HTTP Protocol

COMP4382

Ahmad Hamo

10-3-2025

world wide Web

Invented by Tim Berners-Lee at CERN 1989.
http://home.cern/topics/birth-web

Developed by W3, World Wide Web Consortium.
https://www.w3.org/Consortium

Initially consisted of HTTP and HTML.

https://commons.wikimedia.org/wiki/File:Tim_Berners-Lee_April_2009.jpg

World wide web

Where does the name come from?

HTTP: **HyperText** Transfer Protocol

HTML: **HyperText** Markup Language

HyperText = a link to another webpage (forms a web!).

A webpage

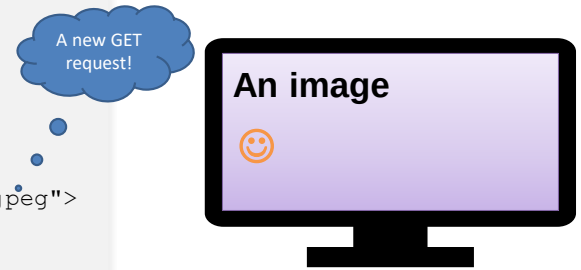
Clients primarily requests HTML files.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hi!</h1>
    <p>Nice to meet you!</p>
  </body>
</html>
```

A webpage

Clients primarily requests HTML files.
But an HTML file usually depends on other files.

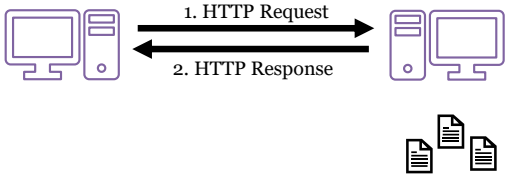
```
<!DOCTYPE html>
<html>
  ...
  <body>
    <h1>An image</h1>
    <img src="smiley.jpeg">
  </body>
</html>
```



Hypertext transfer protocol

Specification: https://tools.ietf.org/html/rfc2616
Built on the client-server model.

- A protocol for distributed, collaborative, hypermedia information systems.
- A **request/response** standard typical of client-server computing.
- Resources accessed by HTTP are identified by **URIs**(Uniform Resource Identifier) (more specifically **URLs**), using the `http` **URI schemes**.
- Format: scheme:[//host]path[?query]



URIs: URLs & URNs

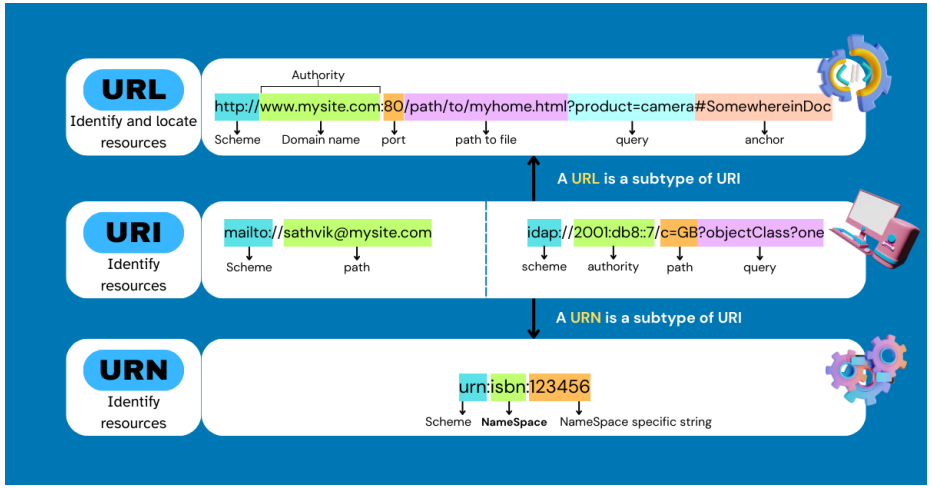
- URIs (Uniform Resource Identifiers) are strings that act as unique identifiers for resources on the internet. They are useful for identifying a variety of items, such as websites, photos, movies, and even actual objects.

- There are two main types of URIs:
  - **URLs (Uniform Resource Locators)**: specify the **location** of a resource, as well as **how to access it**. For example, the URL https://www.example.com/ specifies the location of the homepage of the website example.com.

  - **URNs (Uniform Resource Names): identify a resource by its name** or other characteristics, but they **do not specify its location**. This means that a URN may not always be used to access the resource it identifies. For instance, the book "**The Hitchhiker's Guide to the Galaxy**" by Douglas Adams has the URN **urn:isbn:0-302-38001-8**, but it doesn't say where you may purchase or borrow it.

Source: https://www.linkedin.com/pulse/uris-urls-urns-explained-simply-bejjanki-sathvik-rao-kh2wc/

URIs: URLs & URNs



How URLs work (1/2)?

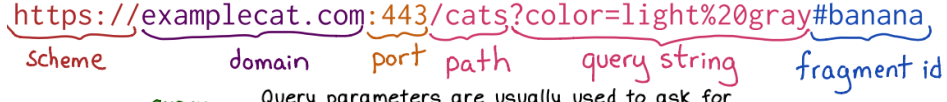
scheme `https://` — Protocol to use for the request. Encrypted (`https`), insecure (`http`), or something else entirely (`ftp`).

domain `examplecat.com` — Where to send the request. For HTTP(s) requests, the Host header gets set to this (`Host: example.com`)

port `:443` — Defaults to 80 for HTTP and 443 for HTTPS.

path `/cats` — Path to ask the server for. The path and the query parameters are combined in the request, like: `GET /cats?color=light%20gray HTTP/1/1`

How URLs work (2/2)?



query parameters `color=light gray` — Query parameters are usually used to ask for a different version of a page ("I want a light gray cat!"). Example:

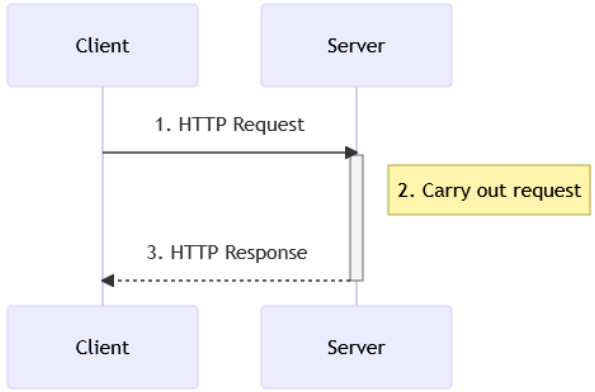
`hair=short&color=black&name=mr%20darcy`

name = value, separated by &

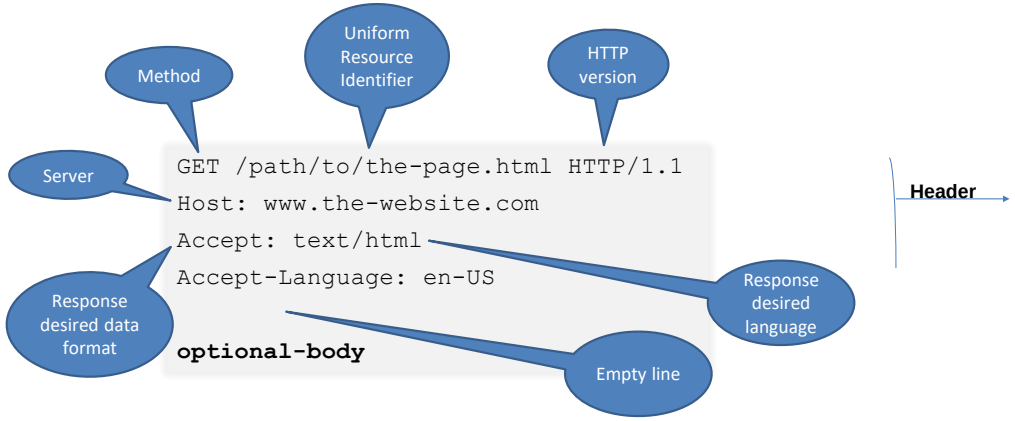
URL encoding `%20` — URLs aren't allowed to have certain special characters like spaces, @, etc. So to put them in a URL you need to percent encode them as % + hex representation of ASCII value. space is `%20`, `%` is `%25`, etc.

fragment id `#banana` — This isn't sent to the server at all. It's used either to jump to an HTML tag (`<a id="banana"..>`) or by Javascript on the page.

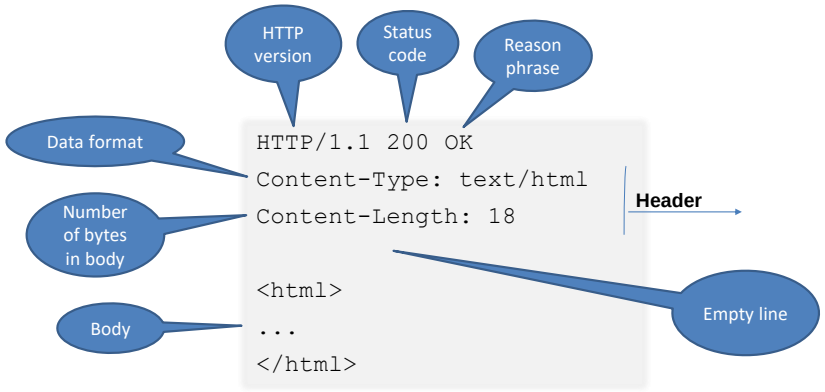
Example of Client-Server communication.



HTTP Request



HTTP Response



HTTP methods - GET

```
GET /path/users/4777 HTTP/1.1
Host: www.the-website.com
Accept: text/html
Accept-Language: en-US
```

GET

- **Purpose:** Used to retrieve data from a server.
- Should not result in changes on the server.
- Data is sent in the **URL parameters** (query string).
- Safe and idempotent, meaning repeated requests should not change the server state.
- Example: GET /users?id=4777 retrieves user data without altering it.

HTTP methods – POST - Example


```
POST /users HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 34
Authorization: Bearer your_token_here
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: application/json
Connection: keep-alive

{
    "name": "John Doe",
    "email": "john@example.com"
}
```

HTTP methods - POST

POST

- **Purpose:** Used to send data to a server to create or update a resource.
- Data can be passed in the body of the request.
- May result in changes on the server.
- Data is sent in the body of the request rather than the URL.
- Not **idempotent**—multiple requests can create multiple resources.
- Typically used for form submissions, file uploads, or API interactions that modify data.
- Example: POST /users with { "name": "John" } in the body creates a new user.

HTML Form: GET, POST

```html
<form action="/search" method="get">
    <label for="query">Search for a Law:</label>
    <input type="text" id="query" name="query">
    <button type="submit">Search</button>
</form>
```

- The data entered in the form is appended to the URL (e.g., **/search?query=contract+law**).
- Best for **search queries** - no data modification

```html
<form action="/submit-question" method="post">
    <label for="question">Ask a Legal Question:</label>
    <input type="text" id="question" name="question">
    <button type="submit">Submit</button>
</form>
```

- The data is sent in the **request body**, not the URL.
- Best for **submitting user input** that changes server data.

HTTP Status codes

1xx Informational
2xx Success
3xx Redirection
4xx Client Error
5xx Server Error

http://www.restapitutorial.com/httpstatuscodes.html

HTTP Common Request methods

Method	Description	Example Use
GET	Retrieve a URI	Retrieve a webpage
HEAD	Retrieve a URI without the response body	Useful for debugging or querying a resource to find metadata such as the size.
POST	Submit data to a resource and create a new entity	Creating a resource.
PUT	Update all data in a resource entity	Updating all fields in an article on a blog (title and body).
PATCH	Update some data in a resource entity	Updating only specific fields in a resource
DELETE	Remove a resource	Remove an article on a blog.

The PUT Verb

- Used to modify (update)resource(s)
- Should contain a message body that specifies the resource to be modified
- Should not contain query string parameters
  - ie *PUT /api/entity?company=15*

The PUT Verb

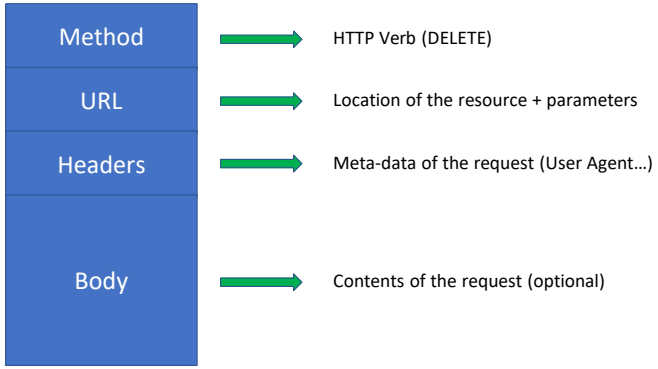
- What happens if the resource to modify does not exist?
- The spec states a new resource should be created
- Use your own discretion (choice)
- PUT is Idempotent

The DELETE Verb

- Used to delete resource(s)
- NEVER use it to add / update / retrieve resources
- Almost always combined with parameters

The DELETE Verb

- Should not include body



HTTP and Sessions

The HTTP protocol is **connectionless**

That is, once the server replies to a request, the server closes the connection with the client, and **forgets** all about the request

In contrast, Unix logins, and JDBC/ODBC connections stay connected until the client disconnects

retaining user authentication and other information

Motivation: **reduces load on server**

operating systems have tight limits on number of open connections on a machine

Information services need session information

E.g., user authentication should be done only once per session

Solution: use a **cookie**

Sessions and Cookies

A **cookie** is a small piece of text containing identifying information

- Sent by server to browser
  - Sent on first interaction, to identify session
- Sent by browser to the server that created the cookie on further interactions
  - **part of the HTTP protocol**
- Server saves information about cookies it issued, and can use it when serving a request
  - E.g., authentication information, and user preferences

Cookies can be stored permanently or for a limited time

```
curl -c cookies.txt https://www.google.com
```

Request Inspection Tools

These tools allow you to inspect HTTP requests and responses:

1.Postman: A popular tool for testing API endpoints and inspecting responses.

2.cURL: A command-line tool for making HTTP requests.

3.Browser Developer Tools: Most modern web browsers include developer tools that allow you to inspect network requests and responses.

HTTP

- Example is relying on **`curl`** which is a command line tool used to transfer data with URL syntax:

Request

```
D:\>curl -v www.google.com
* Host www.google.com:80 was resolved.
* IPv6: (none)
* IPv4: 142.250.200.196
*   Trying 142.250.200.196:80...
* Connected to www.google.com (142.250.200.196) port 80
> GET / HTTP/1.1
> Host: www.google.com
> User-Agent: curl/8.9.1
> Accept: */*
>
```

```
D:\>curl -v https://www.google.com
* Host www.google.com:443 was resolved.
* IPv6: (none)
* IPv4: 142.250.200.196
*   Trying 142.250.200.196:443...
* Connected to www.google.com (142.250.200.196) port 443
* schannel: disabled automatic use of client certificate
```

HTTP

- Example is relying on **`curl`** which is a command line tool used to transfer data with URL syntax:

Response

```
HTTP/1.1 200 OK
Date: Tue, 11 Mar 2025 18:36:09 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Content-Security-Policy-Report-Only: object-src 'none';base-uri 'self';scri
mple' 'unsafe-eval' 'unsafe-inline' https: http:;report-uri https://csp.wit
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: AEC=AVcja2emAG3mDJaYd_kS0dcCoiO5zebCazkki7Lr4rC2yIZoJ8Xt6NEZU7g
om; Secure; HttpOnly; SameSite=lax
Set-Cookie: NID=522=VZyFv9uHbBsb9FT3yOuhOdiCRo3k5fSKsGPwTd1YpB272hXUlGdmiLA
MfWaUdYByv4D5sKljPdWxVntNnaaQnjxp7FgHaAckJjgbMjAloqnuQi21z6_Tfe8kK2mL-nnsva
n=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

doctype html><html dir="rtl" itemscope="" itemtype="http://schema.org/WebPa
```

cURL options

Basic Requests

Option	Description	Example
`-X`	Specify HTTP method	`curl -X POST https://example.com`
`-I`	Perform a **HEAD** request	`curl -I https://example.com`
`-L`	Follow redirects	`curl -L https://example.com`
`-v`	Show detailed request/response info	`curl -v https://example.com`
`-s`	Silent mode (no progress output)	`curl -s https://example.com`

Authentication

Option	Description	Example
`-u`	Basic authentication (`username:password`)	`curl -u user:pass https://example`

Sending Data

Option	Description	Example
`-d`	Send **POST** data	`curl -X POST -d "name=John" https://example.com`

Handling Cookies & Sessions

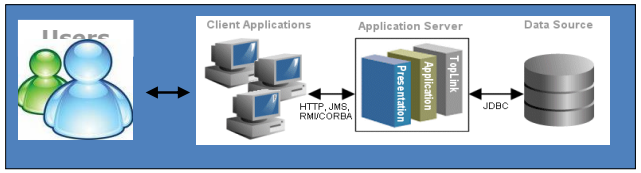
Option	Description	Example
`-b`	Send cookies	`-b "session=12345" https://example.com`
`-c`	Save cookies to a file	`curl -c cookies.txt https://example.com`

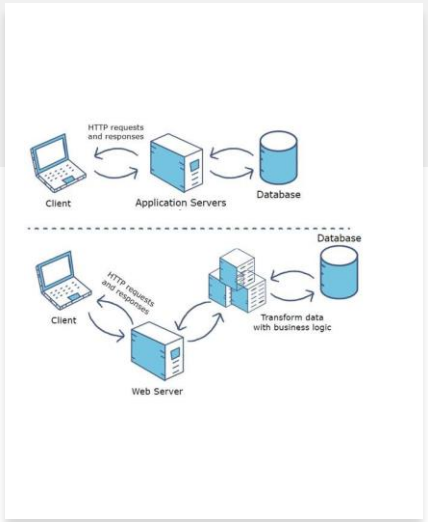
Downloading Files

Option	Description	Example
`-o`	Save output to a file	`curl -o file.html https://example.com`

Architecture of Web Applications

- **Three-layer architecture**





Web server vs Application server

- A **web server** accepts and fulfills requests from clients for static content (i.e., HTML pages, files, images, and videos) from a website. Web servers handle HTTP requests and responses.

- An **application server** exposes *business logic* to the clients, which generates dynamic content. It is a software framework that transforms data to provide the specialized functionality offered by a business, service, or application. Application servers enhance the interactive parts of a website that can appear differently depending on the context of the request.

9

Three-Layer Web Architecture

1. **Presentation Layer**
   - Displays the user interface (e.g., browser or app) and captures user input.
   - Relies on the web server to deliver static files (e.g., HTML) or dynamic content prepared by the Application Layer.
   - Handles communication with the application layer via **HTTP requests**
2. **Application Layer (Business Logic Layer)**
   - Processes **requests** from the client, enforces business rules, and interacts with the data layer.
   - It runs the core functionality of the web application, often using frameworks like FastAPI, Spring Boot,...
   - The **web server** (e.g., Uvicorn in a FastAPI setup) **lives here**, receiving HTTP requests and running the application code (e.g., FastAPI endpoints).
   - It bridges the Presentation Layer (via the web server's **responses**) and the Data Layer.
3. **Data Layer**
   - Stores and retrieves data (SQL or NoSQL) ensuring persistence and security.
   - Interacts with the web server in the Application Layer to provide or receive data as needed.

Tiers in FastAPI vs. Firebase Web Apps

Layer	FastAPI (Three-Tier)	Firebase (Two-Tier)	Firebase (Three-Tier)
Presentation	Client (e.g., Browser, Mobile App)	Client (e.g., Web App w/ SDK)	Client (e.g., Web App w/ SDK)
Application	FastAPI (Uvicorn, Custom Logic)	None (Direct to Data via SDK)	Cloud Functions (Serverless Logic)
Data	Database (e.g., SQLite, PostgreSQL)	Firestore/Realtime Database	Firestore/Realtime Database

Web Services

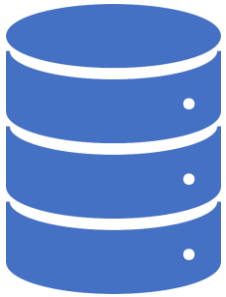
Allow data on Web to be accessed using remote procedure call mechanism

Two approaches are widely used

- **Representation State Transfer (REST)**: allows use of standard HTTP request to a URL to execute a request and return data
  - returned data is encoded either in XML, or in **JavaScript Object Notation (JSON)**
- **Big Web Services:**
  - uses XML representation for sending request data, as well as for returning results
  - standard protocol layer built on top of HTTP

Database

- A **database** is an organized collection of data.
- There are many different strategies for organizing data to facilitate easy access and manipulation.
- A **database management system** (**DBMS**) provides mechanisms for storing, organizing, retrieving and modifying data for many users.



Python's database connectivity

- is defined by the Python Database API Specification v2.0 (PEP 249).
- Instead of a single, universal API like JDBC, Python relies on individual database drivers that adhere to the DB-API specification.
- So, for example, there are specific drivers for PostgreSQL (**psycopg2**), MySQL (**mysql-connector-python**), SQLite (**sqlite3**, which is included in the python standard library), and others.
- Python programs communicate with databases and manipulate their data using these **driver APIs**.

Modern Database Engines with Native REST API Support

- **Firebase Realtime Database and Cloud Firestore**
- **CouchDB:** Fully RESTful, with all operations (CRUD, views, replication) accessible via HTTP methods (e.g., GET /db/doc, PUT /db/doc).
- **MongoDB (via MongoDB Atlas)**: MongoDB Atlas provides a Data API (introduced in 2022) that exposes a REST interface (e.g., POST /v1/action/findOne to query documents)
- **Supabase (PostgreSQL-based)**: Built on PostgreSQL, Supabase auto-generates a REST API for tables and views (e.g., GET /rest/v1/users
- **Exist-DB**: provides a **RESTful API** that allows direct interaction with the database using standard HTTP methods (GET, POST, PUT, DELETE).

- **DynamoDB** (via AWS SDK or HTTP API)
- **SQLite** (with Extensions like ws4sqlite): Native SQLite doesn't have a REST API, but tools like ws4sqlite (a lightweight HTTP server for SQLite) provide a RESTful interface (e.g., GET /db/users).

Python Implementation - Get

```
import requests

response = requests.get('https://api.example.com/data')

params = {'key1': 'value1', 'key2': 'value2'}
response = requests.get('https://api.example.com/data',
params=params)
```

This will construct the URL with the query parameters:
`https://api.example.com/data?key1=value1&key2=value2`

Python Implementation - POST

```
import requests

data = {'username': 'john_doe', 'password': 'secret'}
response = requests.post('https://api.example.com/login',data=
data)
```

You can also send JSON data:

```
import json

json_data = json.dumps({'name': 'John Doe', 'age': 30})
headers = {'Content-Type': 'application/json'}
response = requests.post('https://api.example.com/users',data=
json_data, headers=headers)
```

Advanced Topic (optional):
Handling concurrent HTTP requests

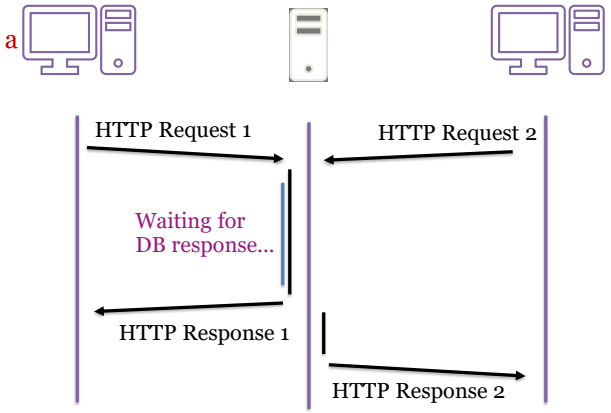
Handling concurrent requests

Attempt 1: Process one request at a time, queue the others.

Bad: Most time wasted on waiting, e.g.:

- Waiting for DB.
- Waiting for reading/writing files.

Very few web applications works this way today.



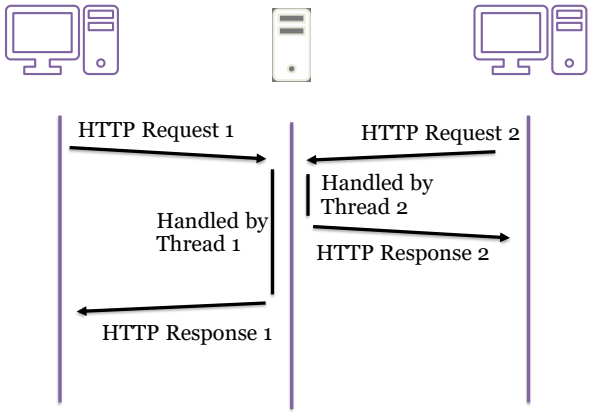
Handling concurrent requests

Attempt 2: Use threads to process requests simultaneously.

Requires us to write thread-safe code.(no deadlocks)

The way many web applications work still today.

Then came Node.js...
And
Python's asyncio library



Handling concurrent requests

Attempt 3: Use a **single** thread with an event loop.

The **event queue** contains tasks to be done.

- Incoming HTTP request are pushed to it.
- Asynchronous operations are pushed to it.

The event loop executes tasks from the event queue.

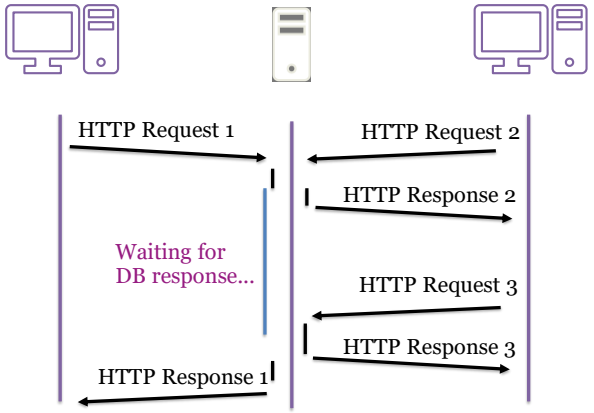
Event Loop/Main code

```
// "pseudocode"
const queue = []
while(true){
  const nextTask =
        queue.unshift()
  nextTask.execute()
}
```

Event Queue

- Do this
- Do that
- Do x
- Do y

Handling concurrent requests



Event Queue

- Handle Request 1
- Handle Request 2
- Handle Request 3
- Handle DB Response

Handling concurrent requests

Attempt 3: Use a single thread with an event loop.

Why is this better than multiple threads?

Context switches (switching thread) are expensive (takes time).
Threads uses a lot of memory.

Any downside?

Asynchronous programming must be used; is a bit harder.

Event Loop

Python's asyncio library

- **Single-Threaded Event Loop**: The asyncio event loop runs in a single thread, managing a queue of tasks (coroutines) that are executed cooperatively. It switches between tasks when they encounter I/O operations (e.g., network requests, file reads) that would otherwise block execution.

```python
from fastapi import FastAPI
import asyncio

app = FastAPI()

@app.get("/slow")
async def slow_request():
    await asyncio.sleep(2)  # Simulates I/O wait
    return {"message": "Done"}

@app.get("/fast")
async def fast_request():
    return {"message": "Quick"}
```

If a client calls /**slow**, the event loop switches to /**fast** during the 2-second sleep, handling both requests concurrently without spawning threads.

asyncio in Python

```python
import asyncio
from time import sleep

from fastapi import FastAPI

app = FastAPI()


@app.get('/sleep/sys')
def nsys_sleep():
    sleep(1)
    return {'error': None}
# system sleep() is blocking
```

```
hey -c 10 -n 10 http://localhost:8000/sleep/sys
```

https://docs.python.org/3/library/asyncio.html

```python
@app.get('/sleep/async-sys')
async def sys_sleep():
    sleep(1)
    return {'error': None}


@app.get('/sleep/async-aio')
async def aio_sleep():
    await asyncio.sleep(1)
    return {'error': None}
```