Getting Started with FastAPI

- Course: Introduction to Web Services
- Instructor: Ahmad Hamo
- Date: 23-4-2025

Introduction to FastAPI

- What is FastAPI?
  - A modern, high-performance Python framework for building REST APIs
  - Core of the FARM stack (FastAPI, React, MongoDB)
  - Acts as the "brain" of the system, managing business logic and data flow
- Why FastAPI?
  - Fast development, clean code, easy debugging
  - Built on Starletta (ASGI) and Pydantic (data validation)
- Topics Covered Today:
  - Overview, setup, Python features, REST API tasks, forms, project structure

Technical Requirements

- Prerequisites:
  - Python 3.11.7 or later (https://www.python.org/downloads/)
  - Virtual environments (using virtualenv)
  - Code editor (e.g., VS Code)
  - REST client (e.g., HTTPie, Postman)
- Checking Python Version:
  - Command: python --version
  - Use pyenv for multiple versions
- Why Virtual Environments?
  - Isolates project dependencies
  - Avoids conflicts between package versions

Setting Up a Virtual Environment

- Command to Create:
  python -m venv venv
- Activation:
  - Windows: venv\Scripts\activate
  - Linux/macOS: source venv/bin/activate
- Key Notes:
  - Deactivate with: deactivate
  - Store in project folder (e.g., venv/)
  - Save dependencies: pip freeze > requirements.txt
- Alternatives: Poetry, virtualenvwrapper, pyenv

pip freeze > requirements

- Lists all the Python packages installed in your current environment (along with their versions) in the format package==version.
- This includes packages installed via pip, whether they were installed directly or as dependencies of other packages.

- The **>** operator redirects the output of pip freeze and writes it to a file named requirements.txt.

Purpose:

- The requirements.txt file is commonly used to document dependencies for a Python project.
- Others can use `pip install -r requirements.txt` to install the exact same versions of the packages in their environment, ensuring consistency.

Tools for Development

- Code Editor: Visual Studio Code
  - Cross-platform, lightweight, plugins (e.g., MongoDB for VS Code)
- Terminal:
  - Linux/macOS: Built-in shell
  - Windows: PowerShell or Cmder (https://cmder.app)
- REST Clients:
  - HTTPie: pip install httpie
  - Postman: GUI-based (https://www.postman.com/)
  - Insomnia: Simple GUI (https://insomnia.rest/)

Installing FastAPI and Uvicorn

- Activate Virtual Environment:
  venv\Scripts\activate
- Install Packages:
  pip install fastapi uvicorn
- What is **Uvicorn**?
  - ASGI server for running FastAPI apps
  - High performance, widely used
- Test HTTPie:
  http GET "http://jsonplaceholder.typicode.com/todos/1"
  - Expected: HTTP/1.1 200 OK

FastAPI in a Nutshell

- Key Features:
  - Fast coding, clean syntax, leverages Python type hints
  - Built on Starlette (ASGI framework) and Pydantic (validation)

- **Starlette:**
  - Provides WebSocket support, middleware, templates (https://www.starlette.io/)

- **Pydantic:**
  - a powerful feature for data validation and conversion.
  - They allow you to define the structure, type, and constraints of the data your application handles, both for incoming requests and outgoing responses

Asynchronous Programming

- Allows applications to handle more requests simultaneously, making them more efficient.
- Asynchronous programming is a style of concurrent programming in which tasks are executed without blocking the execution of other tasks, improving the overall performance of your application.
- To integrate asynchronous programming smoothly, **FastAPI** leverages the **async/await** syntax (https://fastapi.tiangolo.com/async/) and automatically integrates asynchronous functions.

Creating APIs

- **`@app.get, @app.post`**, etc. - Decorators to define HTTP methods and endpoints.
- **`FastAPI`** - The main class to create the API application.
- FastAPI automatically generates interactive API documentation (Swagger UI) at /docs.

Endpoints

- Endpoints are the points at which API interactions happen.
- In FastAPI, an endpoint is created by decorating a function with an HTTP method, such as **@app.get("/").**
- This signifies a GET request to the root of your application

- Unlike Spring Boot's @RestController, FastAPI uses function-based routing with decorators.
- The `async` keyword enables asynchronous request handling, improving performance for I/O-bound operations.

Hello World Example

- Code (chapter4_01.py):

```
from fastapi import FastAPI
 app = FastAPI()

 @app.get("/")
 async def root():
     return {"message": "Hello FastAPI"}
```

- Run with Uvicorn - manually:

```
uvicorn chapter4_01:app –reload
```

- Test with HTTPie:

```
http http://localhost:8000/
```

- Output: {"message": "Hello FastAPI"} (HTTP 200 OK)

Adding a POST Endpoint

- Updated Code (chapter4_01.py):

```
@app.post("/")
 async def post_root():
     return {"message": "Post request success!"}
```

- Test POST:

http POST http://localhost:8000

- Output: {"message": "Post request success!"}

Note: GET and POST handled separately;

If an endpoint is only decorated with @app.post, FastAPI won't register a handler for GET.

Attempting GET / on a POST-only endpoint triggers FastAPI's built-in HTTP method validation.

```
http GET http://localhost:8000/
```

```
HTTP/1.1 405 Method Not Allowed
```

Automatic Documentation

- Feature: Interactive API docs
  - URL: http://localhost:8000/docs
  - Based on OpenAPI and Pydantic
- Benefits:
  - Test endpoints directly in browser
  - Shows inputs, responses, errors
- Comparison:
  - vs. Postman/Insomnia: Built-in, no extra tools needed

Handling HTTP Requests

- Path Parameters
  - Type Hinting with Path Parameters
  - Path Order
  - Path Values Restriction
- Query Parameters
  - Optional Query Parameters
  - Type Hinting with Query Parameters
- Request Headers
- Request Body

Path Parameters

- Basic Example (chapter4_02.py):

```
@app.get("/car/{id}")
 async def root(id):
     return {"car_id": id}
```

- Test:

http http://localhost:8000/car/**1**

http http://localhost:8000/car/**billy**

- Outputs: {"car_id": "1"}, {"car_id": "billy"}

- Note: No type enforcement by default (strings)
- Path parameters extract values from the URL **using Python type hints**.

Type Hinting with **Path Parameters**

- Typed Example (chapter4_02.py):

```
@app.get("/carh/{id}")
async def hinted_car_id(id: int):
    return {"car_id": id}
```

- Test:

/carh/**1** # Works: {"car_id": 1}

/carh/billy # Fails: 422 Unprocessable Entity

- Benefit: Automatic validation and error handling

Path Order Matters

- Incorrect Order (chapter4_03.py):

```
@app.get("/user/{id}")
async def user(id: int):
    return {"user_id": id}

@app.get("/user/me")
async def me_user():
    return {"user_id": "This is me!"}
```

- Test:

/user/me # Fails: 422 (tries to parse "me" as int)

- Fix: Define /user/me before /user/{id}

Restricting Path Values

how it limits the path to a specific set of values?

- Enum Example (chapter4_04.py):

```
from enum import Enum
from fastapi import FastAPI, Path
app = FastAPI()

 class AccountType(str, Enum):
     FREE = "free"
     PRO = "pro"

 @app.get("/account/{acc_type}/{months}")
 async def account(acc_type: AccountType, months: int =
Path(..., ge=3, le=12)):
     return {"message": "Account created", "account_type":
acc_type, "months": months}
```

- Test: Only **free** or **pro** for acc_type, **3-12** for months
- Path(...) is used to define metadata and validation rules for path parameters in an API endpoint.
- This path parameter is **required** (because of ...)

Optional Path Parameter

```
from fastapi import FastAPI, Path
from typing import Optional

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: Optional[int] =
Path(None, ge=3, le=12)):
    if item_id is None:
        return {"message": "No item_id provided"}
    return {"item_id": item_id}
```

Testing - 1

GET /account/free/3

Response (200 OK):

```
{
  "message": "Account created",
  "account_type": "free",
  "months": 3
}
```

Testing - 2

GET /account/pro/12

Response (200 OK):

```
{
  "message": "Account created",
  "account_type": "pro",
  "months": 12
}
```

Testing - 3

GET /account/premium/6

Response (422 Unprocessable Entity):


```
{
  "detail": [
    {
      "type": "enum",
      "loc": ["path", "acc_type"],
      "msg": "Input should be 'free' or 'pro'",
      "input": "premium",
      "ctx": {"expected": "'free' or 'pro'"}
    }
  ]
}
```


Testing - 4

GET /account/pro/2

Response (422 Unprocessable Entity):


```
{
  "detail": [
    {
      "type": "greater_than_equal",
      "loc": ["path", "months"],
      "msg": "Input should be greater than or equal to 3",
      "input": 2,
      "ctx": {"ge": 3}
    }
  ]
}
```

Query Parameters

- Extract query parameters with **optional defaults and validation**.

- Example (chapter4_05.py):

```
 @app.get("/cars/price")
   async def cars_by_price(min_price: int = 0,
max_price: int = 100000):
       return {"message": f"Listing cars with prices
between {min_price} and {max_price}"}
```

- Test:

/cars/price?**min_price=2000&max_price=4000**

- Output: {"message": "Listing cars with prices between 2000 and 4000"}

- Note: Defaults to 0 and 100000 if omitted

Testing 1

GET /cars/price

Response (200 OK):

```
{
  "message": "Listing cars with
prices between 0 and 100000"
}
```

Testing 2

GET /cars/price?min_price=5000&max_price=20000

Response (200 OK):


```
{
  "message": "Listing cars with prices between 5000 and 20000"
}
```


Testing 3

GET /cars/price?min_price=hello&max_price=20000

Response (422 Unprocessable Entity):


```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": ["query", "min_price"],
      "msg": "Input should be a valid integer, unable to parse string as an integer",
      "input": "hello"
    }
  ]
}
```

Testing 4

GET /cars/price?min_price=30000&max_price=10000

Response (200 OK):

```
{
  "message": "Listing cars with prices between 30000
and 10000"
}
```

FastAPI won't reject this because it doesn't validate logical relationships between parameters.

To fix this, add manual validation in the function:

```
    if min_price > max_price:
        raise HTTPException(status_code=400,
detail="min_price cannot exceed max_price")
```

Query Parameters – example2

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/products")
async def get_products(category_id: int | None = None,
sort_by: str = "name"):
    return {"category_id": category_id, "sort_by": sort_by}
```

This endpoint supports two optional query parameters:

- **category_id**: int or None with None as default value
- **sort_by**: str with default value "name"

- 20.py

Test Calls

Purpose	Command	Response
Fetch all products	/products	{"category_id": null, "sort_by": "name"}
Fetch products by category ID	/products?category_id=5	{"category_id": 5, "sort_by": "name"}
Fetch products sorted by price	/product? sort_by=="price"	{"category_id": null, "sort_by": "price"}
Fetch products by category ID and sort by “raing”	/products?category_id==10 &sort_by="rating"	{"category_id": 10, "sort_by": "rating"}
Invalid query parameter	/products?category_id==3 &invalid_param="test"	{"category_id": 3, "sort_by": "name"} **(ignores invalid_param)**
same parameters twice	/products?category_id=5&category_id=6	{"category_id":6,"sort_by":"name"}

Request Body

- Post
- Post with Type Validation
- Control Required/Optional Fields
- Complex Request Body

Request Body

- Basic Example (chapter4_06.py):

```
from typing import Dict
from fastapi import FastAPI, Body
app = FastAPI()

@app.post("/cars")
async def new_car(data: Dict = Body(...)):
    return {"message": data}
```

data: Dict = Body(...) means:

- The endpoint expects a JSON body (automatically parsed by FastAPI).
- ... (Ellipsis) indicates the body is **required** (cannot be empty).
- The parsed JSON will be passed as a dictionary (Dict) to the data parameter.

Test 1

POST /cars
Content-Type: application/json

```
{
 "make": "Toyota",
 "model": "Camry",
 "year": 2022
}
```

Response (200 OK):

```
{
  "message": {
    "make": "Toyota",
    "model": "Camry",
    "year": 2022
  }
}
```

Test 2

POST /cars
Content-Type: application/json

```
{
 "make": "Tesla",
 "model": "Model S",
 "specs": {
  "range": "370 miles",
  "autopilot": true
 }
}
```

Response (200 OK):

```
{
  "message": {
    "make": "Tesla",
    "model": "Model S",
    "specs": {
      "range": "370 miles",
      "autopilot": true
    }
  }
}
```

Test 3

POST /cars
Content-Type: application/json

{} --- nothing

Response (422 Unprocessable Entity):

```
{
  "detail": [
    {
      "type": "missing",
      "loc": ["body"],
      "msg": "Field required",
      "input": null
    }
  ]
}
```

Key points

- Requires JSON Body: The endpoint expects a JSON payload (enforced by Body(...)).
- **No Schema Validation**: Your current implementation accepts any dictionary (even invalid car data).
- To validate fields, use a **Pydantic** mode

Pydantic Models for Request Body

- Typed Example (chapter4_07.py):

```
from pydantic import BaseModel

class InsertCar(BaseModel):
    brand: str
    model: str
    year: int

@app.post("/cars")
async def new_car(data: InsertCar):
    return {"message": data}
```

- Test: Enforces structure; extra fields ignored, invalid types rejected

Test 4 – with validation

**curl -X POST http://localhost:8000/cars **
**-H "Content-Type: application/json" **
-d '{"brand": "Honda", "model": "Civic", "year": 2023}'

Response (200 OK):

```
{
  "message": {
    "brand": "Honda",
    "model": "Civic",
    "year": 2023
  }
}
```

Test 5– with validation

**curl -X POST http://localhost:8000/cars **
**-H "Content-Type: application/json" **
-d '{"brand": "Honda", "model": "Civic"}'

422 Unprocessable Entity

```
{
  "detail": [
    {
      "type": "missing",
      "loc": ["body", "year"],
      "msg": "Field required",
      "input": {"make": "Honda", "model": "Civic"}
    }
  ]
}
```

Control Required/Optional Fields

Field Definition	Required?	Behaviour
year: int	✅ Required	Fails if missing in request.
year: Optional[int]	❌ Optional	Defaults to None if missing.
year: int = 2023	❌ Optional	Defaults to 2023 if missing.

Example:

```
class Car(BaseModel):
    make: str
    model: str
    year: Optional[int] = None   # Optional field (defaults to None)
```

Complex GET Request

- Path parameters can be used to express relation between entities

```
from fastapi import FastAPI, HTTPException, Path
from typing import Dict

app = FastAPI()

# Mock data
users = {
    1: {"name": "Alice"},
    2: {"name": "Bob"}
}

cars = {
    1: [{"car_id": 101, "model": "Toyota"}, {"car_id": 102, "model": "Tesla"}],
    2: [{"car_id": 201, "model": "BMW"}]
}
```

Complex GET Request – cont.

```python
@app.get("/users/{user_id}/cars/{car_id}")
def get_user_car(
    user_id: int = Path(..., gt=0),
    car_id: int = Path(..., gt=0)
):
    if user_id not in users:
        raise HTTPException(status_code=404, detail="User not found")

    user_cars = cars.get(user_id, [])
    for car in user_cars:
        if car["car_id"] == car_id:
            return {"user": users[user_id], "car": car}
    raise HTTPException(status_code=404, detail="Car not found for this
user")
```

Test 1 – Existing user and car

- Request: GET /users/1/cars/101
- Response: 200 OK

{ "user": {"name": "Alice"},
"car": {"car_id": 101, "model": "Toyota"}}

Test2: Car does not belong to the user

- Request: GET /users/1/cars/201
- Response: 404 Not Found

```
{
  "detail": "Car not found for this user"
}
```

Complex POST Request Body

- Example (chapter4_08.py):

```
  class UserModel(BaseModel):
      username: str
      name: str

  @app.post("/car/user")
  async def new_car_model(car: InsertCar, user:
UserModel, code: int = Body(None)):
      return {"car": car, "user": user, "code":
code}
```

- Uses **Pydantic models** (InsertCar, UserModel) for structured data validation.
- code is optional (defaults to None if not provided).
- Test: Use GUI client (e.g., Insomnia) for complex JSON
- Extra Fields in JSON Body will be Ignored by FastAPI

Test1 - **Successful Request (All Fields Provided)**

- http://127.0.0.1:8000/car/user

```
{
    "car": {
        "brand": "Toyota",
        "model": "Corolla",
        "year": 2022
    },
    "user": {
        "username": "john_doe",
        "name": "John Doe"
    },
    "code": 1001
}
```

Cookies Headers (4) Test Results 200 OK

JSON Preview Visualize

```
{
    "car": {
        "brand": "Toyota",
        "model": "Corolla",
        "year": 2022
    },
    "user": {
        "username": "john_doe",
        "name": "John Doe"
    },
    "code": 1001
}
```

Test2 - **Successful Request (Optional code Omitted)**

- http://127.0.0.1:8000/car/user

```
{
    "car": {
        "brand": "Toyota",
        "model": "Corolla",
        "year": 2022
    },
    "user": {
        "username": "john_doe",
        "name": "John Doe"
    }
}
```

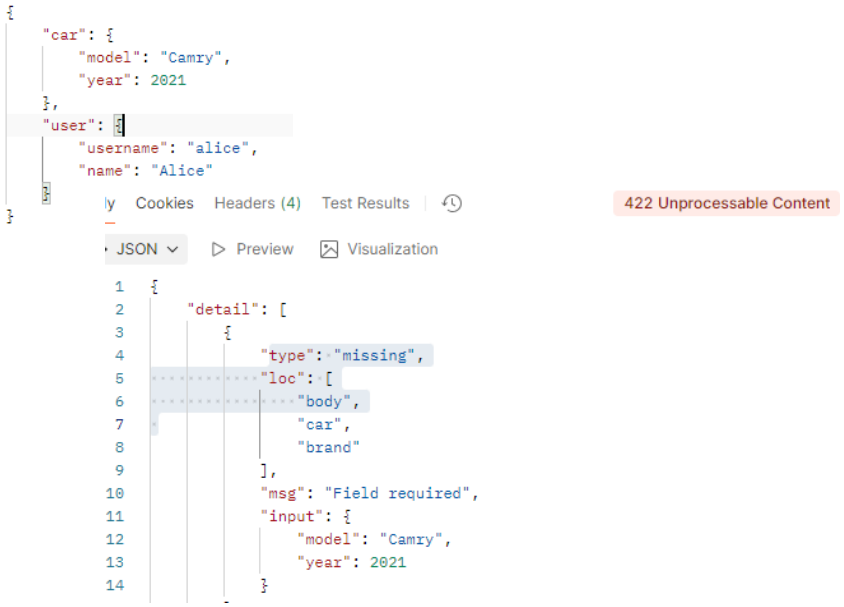
Cookies Headers (4) Test Results 200 OK

JSON Preview Visualization

```
{
    "car": {
        "brand": "Toyota",
        "model": "Corolla",
        "year": 2022
    },
    "user": {
        "username": "john_doe",
        "name": "John Doe"
    },
    "code": null
}
```

Test3 - **Missing** Required Field (**brand** in car)

- http://127.0.0.1:8000/car/user



Raw Request Object

- Example (chapter4_09.py):

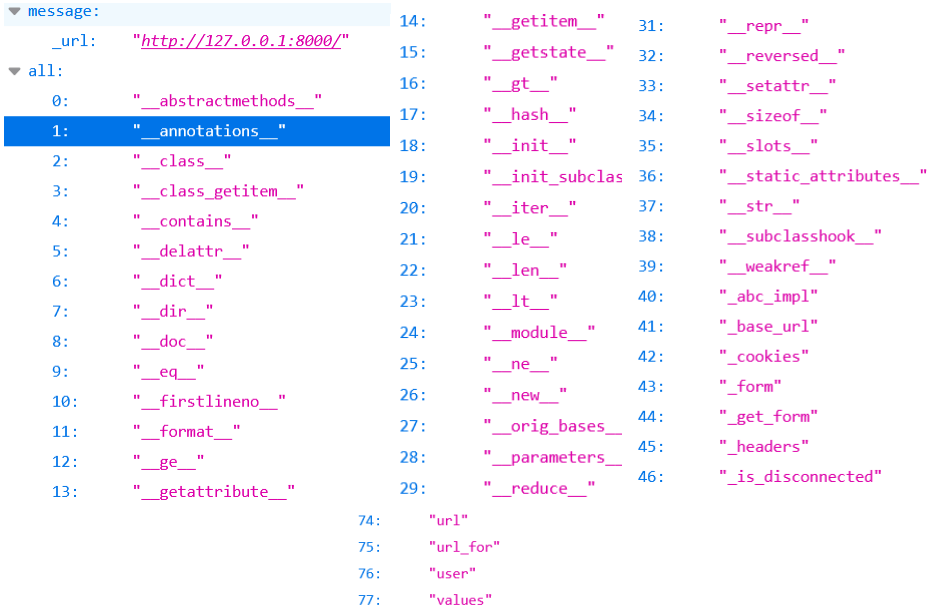
```
from fastapi import FastAPI, Request
app = FastAPI()

 @app.get("/cars")
 async def raw_request(request: Request):
     return {"message": str(request.base_url), "all":
dir(request)}
```

This FastAPI endpoint (/cars) is a GET endpoint that:

- Takes a Request object as a parameter (provided by FastAPI)
- Returns a JSON response containing:
  - The **base URL** of the request (request.base_url)
  - **All attributes/methods available on the Request object** (dir(request))

Test1 - GET http://localhost:8000/cars



Raw Request Object – be careful

- From the all list, these are particularly useful:
  - **request.headers** - Access request headers
  - **request.cookies** - Access cookies
  - **request.query_params** - Access query parameters
  - **request.client** - Get client (host, port) information
  - **request.method** - Get HTTP method used

- Using the **raw request** bypasses FastAPI's key features like Pydantic parsing, validation, and self-documentation
- but may be necessary in specific cases (debugging, ...)

Request Headers 1 - Access headers using the Header parameter

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/products")
async def get_products(x_auth_token: str = Header()):
    return {"token": x_auth_token}
```

- **Purpose** : The endpoint retrieves the value of the **x-auth-token** header and returns it in the response.
- **Key Behaviour** :
  1. If the x-auth-token header is not provided, FastAPI will automatically return a **422 Unprocessable Entity error** because the Header() parameter is required by default.
  2. If the header is provided, its value is returned in the response.

Examples: Request Headers

- **Valid header**:

```
curl -X GET "http://localhost:8000/products" -H "x-auth-token:"my-secret-token"
```

```
{"token": "my-secret-token"}
```

- **Missing header:** `curl -X GET "http://localhost:8000/products`

```
{"detail": [{"loc": ["header", "x-auth-token"], "msg": "field required", ...}]}
```

- **Empty header**: `curl -X GET "http://localhost:8000/products" -H "x-auth-token:""`

```
{"token": ""}
```

- **Incorrect header** name: `curl -X GET "http://localhost:8000/products" -H “auth-token:"my-secret-token"`

```
{"detail": [{"loc": ["header", "x-auth-token"], "msg": "field required", ...}]}
```

Request Headers 2- **header parameter**

```python
from typing import Annotated
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/info")
async def read_headers(user_agent: Annotated[str | None,
Header()] = None):
    return {"User-Agent": user_agent}
```

- **Annotated[...]:** A way to attach extra information (like metadata) to the type hint — here it's saying: "this parameter is a string or None, and it comes from a header".
- **Header()** :Tells FastAPI: "This value should be taken from an HTTP header", not from a path, query, or body.

Test 1 – With User-Agent header

curl -X GET "http://localhost:8000/info" -H "User-Agent: MyCustomAgent/1.0“

```
{
  "user_agent": "MyCustomAgent/1.0"
}
```

Test 2: Without User-Agent header

curl -X GET http://localhost:8000/info


```
{
  "user_agent": null
}
```


Dependency Injection Overview

- FastAPI has a built-in dependency injection system
- Helps in reusing code and handling authentication, logging, DB connections
- How it works?

Dependencies are declared using the Depends() function, allowing FastAPI to manage and *inject external logic or objects into your path operations.*

What is Dependency Injection?

- A design pattern where dependencies (e.g., classes, functions, objects) are provided to a component rather than being created by the component itself.
- Goal: Improve modularity, reusability, and testability.
- In FastAPI: You can declare dependencies in path operation functions, and FastAPI will automatically resolve and inject them.
- **Example**: we want inject the following function:

```
def get_db():
    db = "Database session"
    return db
```

Using Dependencies in FastAPI

```
from fastapi import FastAPI, Depends

app = FastAPI()

@app.get("/items/")
def read_items(db=Depends(get_db)):
    return {"db": db}
```

- Inject via Depends() in function parameters.
- FastAPI calls get_db(), gets the return value, and passes it to read_items().

Dependency Injection - common

```
from fastapi import Depends

def common_parameters(q: str | None = None):
   return {"q": q}

@app.get("/items/")
def read_items(commons:
dict=Depends(common_parameters)):
   return commons
```

Dependency Injection - Pagination

- Example (chapter4_15.py):

```
from typing import Annotated
from fastapi import Depends, FastAPI
async def pagination(q: str | None = None, skip: int = 0,
limit: int =100):
     return {"q": q, "skip": skip, "limit": limit}

@app.get("/cars/")
async def read_item(commons: Annotated[dict,
Depends(pagination)]):
     return commons
```

- The code showing a pagination injected

Notice: (q: str | None = None) means:
q can be either a string (str) or None (i.e., optional).

1. Basic call with default pagination

- curl -X GET "http://localhost:8000/cars/"
- Response:

{"q": null, "skip": 0, "limit": 100}

2. Call with search query and custom pagination

- curl -X GET "/cars/?q=toyota&skip=10&limit=5"
- Response:

{"q": "toyota", "skip": 10, "limit": 5}

3. Call with just a search query (using default pagination)

- curl -X GET "http://localhost:8000/cars/?q=ford"
- Response:

{"q": "ford", "skip": 0, "limit": 100}

4. Invalid skip parameter

- curl -X GET "http://localhost:8000/cars/?skip=ten"
- Expected error response (422 Unprocessable Entity):


```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": ["query", "skip"],
      "msg": "Input should be a valid integer, unable to parse string as an
integer",
      "input": "ten",
      "url": "https://errors.pydantic.dev/2.5/v/int_parsing"
    }
  ]
}
```

Routers

- When we need to handle multiple endpoints that are in different files, we can benefit from using routers.
- Routers assist us in **grouping** our endpoints into different modules, which makes our code base easier to maintain and understand.
- For example, we could use one router for operations related to users and another for operations related to products.

Routers

- **Main File (chapter4_16.py):**

```
from fastapi import FastAPI
from routers.cars import router as cars_router
from routers.user import router as users_router
app = FastAPI()
app.include_router(cars_router, prefix="/cars", tags=["cars"])
app.include_router(users_router, prefix="/users", tags=["users"])
```

- **cars.py:**

```
from fastapi import APIRouter
router = APIRouter()
@router.get("/")
async def get_cars():
    return {"message": "All cars here"}
```

-

```
project/
├── main.py             # Main FastAPI app (entry point)
└── routers/
    ├── __init__.py     # Optional: Makes 'routers' a Python package
    ├── cars.py         # Defines the cars_router
    └── users.py        # Defines the users_router
```

curl -X GET "http://localhost:8000/**cars**/"
Response: {"message": "All cars here"}

app.include_router(cars_router, prefix="/cars", tags=["cars"])

a) cars_router
- Defined in cars.py as router = APIRouter().
- Contains route definitions (e.g., @router.get("/")).

b) prefix="/cars"
- Prepends /cars to all routes in cars_router.
- Example: The route @router.get("/") becomes /cars/ in the main app.
- Without prefix, the endpoint would just be /.

c) tags=["cars"]

Groups routes under the "cars" tag in FastAPI's Swagger/OpenAPI docs.

Form Data and File Upload

- Install: pip install python-multipart
- Example (chapter4_11.py):

```
from fastapi import FastAPI, Form, File, UploadFile
 app = FastAPI()

 @app.post("/upload")
 async def upload(file: UploadFile = File(...),
brand: str = Form(...), model: str = Form(...)):
     return {"brand": brand, "model": model,
"file_name": file.filename}
```

- Test:

http -f POST localhost:8000/upload brand="Ferrari" model="Testarossa" file@car.jpeg

Saving Uploaded Files

- Example (chapter4_12.py):

```
 import shutil
  @app.post("/upload")
  async def upload(picture: UploadFile = File(...),
brand: str = Form(...), model: str = Form(...)):
      with open("saved_file.png", "wb") as buffer:
          shutil.copyfileobj(picture.file, buffer)
      return {"brand": brand, "model": model,
"file_name": picture.filename}
```

- Note: Overwrites file; use UUID for unique names

Customizing Responses

- Status Code Example (chapter4_13.py):

```
 from fastapi import FastAPI, status
 @app.get("/",
status_code=status.HTTP_208_ALREADY_REPOR
TED)
 async def raw_fa_response():
    return {"message": "fastapi response"}
```

- Test: Returns 208 Already Reported

HTTP Errors

- Example (chapter4_14.py):

```
from fastapi import FastAPI, HTTPException, status
@app.post("/carsmodel")
async def new_car_model(car: InsertCar):
    if car.year > 2022:
        raise
HTTPException(status.HTTP_406_NOT_ACCEPTABLE
, detail="The car doesn't exist yet!")
    return {"message": car}
```

- Test: year=2023 → 406 Not Acceptable

Summary

- Key Takeaways:
  - FastAPI: Fast, modern, Pythonic REST API framework
  - Features: Type hints, async, Pydantic, auto-docs
  - Tools: Virtual env, Uvicorn, HTTPie, VS Code
  - Topics: Paths, queries, bodies, forms, routers, middleware

References

- https://fastapi.tiangolo.com/
- https://docs.pydantic.dev/
- https://www.uvicorn.org/

Q&A

- Questions?
- Let’s discuss your first FastAPI apps!