

Module 4 - Week 2

Web Deployment using FastAPI

TimeSeries Team

Ngày 19 tháng 9 năm 2025

I. Overview?

1.1. API là gì

API (Application Programming Interface) là cầu nối giao tiếp giữa giao diện người dùng (Frontend), phần logic xử lý (Backend) và cơ sở dữ liệu (Database).

Dựa trên sơ đồ minh họa:




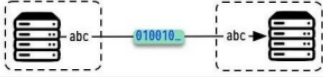
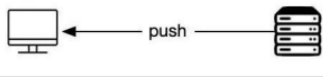
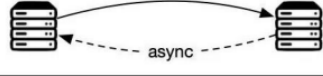
❖ What is API?



- Người dùng (**User**) tương tác với **UI – Frontend**, nơi hiển thị thông tin và nhận dữ liệu đầu vào.
- Giao diện này sẽ **gửi yêu cầu** đến **Backend**, nơi chứa các đoạn mã logic để xử lý nghiệp vụ.
- Khi cần thiết, Backend sẽ **truy vấn dữ liệu** từ **Database**, nơi lưu trữ và cung cấp thông tin.
- Cơ sở dữ liệu phản hồi và gửi lại **dữ liệu trả về** cho Backend, sau đó Backend xử lý và tạo ra kết quả phù hợp.
- Cuối cùng, kết quả này được gửi về Frontend và **hiển thị** cho người dùng.

Nói một cách trừu tượng, ta có thể hiểu API như một người bồi chuyển yêu cầu đến nhà bếp. Trong đó, menu đóng vai trò là phần frontend, đầu bếp chính là backend. Việc bồi bàn chuyển yêu cầu gọi món của ta đến với đầu bếp chính là hành động send request bằng API

Tóm lại, API giúp các thành phần phần mềm giao tiếp với nhau một cách mạch lạc, tạo điều kiện để người dùng tương tác dễ dàng với ứng dụng.

Style	Illustration	Use Cases
SOAP		XML-based for enterprise applications
RESTful		Resource-based for web servers
GraphQL		Query language reduce network load
gRPC		High performance for microservices
WebSocket		Bi-directional for low-latency data exchange
Webhook		Asynchronous for event-driven application

Hiện nay có nhiều phong cách (style) API được sử dụng trong phát triển phần mềm, mỗi loại có đặc điểm và tình huống ứng dụng riêng biệt:

- **SOAP**: Sử dụng định dạng **XML**, thường được áp dụng trong các hệ thống doanh nghiệp lớn. Đây là chuẩn cũ, chặt chẽ và có tính bảo mật cao nhưng phức tạp.
- **RESTful**: Dựa trên tài nguyên (**Resource-based**), phổ biến cho các máy chủ web nhờ tính đơn giản, dễ mở rộng và sử dụng các phương thức HTTP chuẩn (GET, POST, PUT, DELETE).
- **GraphQL**: Cung cấp ngôn ngữ truy vấn linh hoạt, cho phép lấy đúng dữ liệu cần thiết và giảm tải băng thông mạng. Thường dùng trong các ứng dụng có giao diện động, yêu cầu nhiều kiểu dữ liệu khác nhau.
- **gRPC**: Giao tiếp hiệu năng cao, đặc biệt hữu ích cho **microservices**. Dùng giao thức truyền dữ liệu nhị phân và hỗ trợ nhiều ngôn ngữ lập trình.
- **WebSocket**: Hỗ trợ kết nối hai chiều (**bi-directional**) với độ trễ thấp, phù hợp cho các ứng dụng thời gian thực như chat, game online, hay hệ thống giám sát.
- **Webhook**: Cơ chế bất đồng bộ (**asynchronous**), cho phép hệ thống gửi sự kiện trực tiếp đến ứng dụng khác khi có thay đổi. Thường dùng trong các ứng dụng hướng sự kiện (event-driven).

Tóm lại, việc lựa chọn kiểu API nào phụ thuộc vào mục tiêu của hệ thống, yêu cầu về hiệu năng, khả năng mở rộng và đặc thù bài toán.

1.2. Uvicorn, ASGI Server là gì? Trong phát triển ứng dụng web Python, **WSGI (Web Server Gateway Interface)** và **ASGI (Asynchronous Server Gateway Interface)** là hai chuẩn giao tiếp giữa web server và ứng dụng.

- **Gunicorn** là một server WSGI, thường dùng cho các framework đồng bộ như **Flask** hay **Django**.
- **Uvicorn** là một server ASGI, hỗ trợ bất đồng bộ, phù hợp với các framework hiện đại như **FastAPI** hay **Starlette**.

- Quá trình hoạt động:
 1. **Client** gửi **request**.
 2. Server trung gian (**Gunicorn/Uvicorn**) tiếp nhận và chuyển tiếp đến framework.
 3. Framework (Flask/Django hoặc FastAPI/Starlette) xử lý request bằng **business logic code** (handler, router).
 4. Kết quả xử lý được trả về qua server và gửi lại cho **Client**.
- **Điểm khác biệt:**
 - WSGI: chỉ hỗ trợ lập trình đồng bộ (synchronous).
 - ASGI: hỗ trợ cả đồng bộ và bất đồng bộ (asynchronous), giúp tối ưu hiệu năng cho ứng dụng thời gian thực.

Tóm lại, Uvicorn (ASGI) thường được dùng cho các ứng dụng cần tốc độ và khả năng xử lý song song, trong khi Gunicorn (WSGI) phù hợp với các ứng dụng truyền thống.

Lưu ý: Như vậy, API sẽ là ứng dụng logic mà ta viết để xử lý nghiệp vụ, còn uvicorn sẽ là server để chạy ứng dụng đó và giao tiếp với thế giới bên ngoài. Nói cách khác API sẽ phụ trách phần "*làm gì*" khi có request đến và uvicorn phụ trách phần "*làm thế nào*" để request và response đi qua

*Các khái niệm cần biết: Synchronous, Asynchronous, Concurrent và Parallel

a. Synchronous vs Asynchronous

- **Synchronous (Đồng bộ):** Một tiến trình (Process A) phải **chờ** tiến trình khác (Process B) hoàn tất và trả về kết quả thì mới tiếp tục. Điều này dễ triển khai nhưng có thể gây chậm trễ nếu tác vụ mất nhiều thời gian.
- **Asynchronous (Bất đồng bộ):** Tiến trình (Process A) **không cần chờ**, mà có thể tiếp tục làm việc khác trong khi chờ kết quả từ tiến trình (Process B). Khi có kết quả, hệ thống sẽ xử lý trả về. Cách này giúp tận dụng tài nguyên tốt hơn và cải thiện hiệu năng trong các tác vụ I/O.

b. Concurrent vs Parallel

- **Not Concurrent, Not Parallel:** Một CPU core chỉ xử lý tuần tự từng tác vụ một (Task 1 xong rồi mới đến Task 2).
- **Concurrent, Not Parallel:** Một CPU core chia nhỏ thời gian cho nhiều tác vụ (Task 1 và Task 2) xen kẽ nhau, giúp chúng hoàn thành gần như cùng lúc, nhưng thực chất vẫn chạy **một tác vụ tại một thời điểm**.
- **Not Concurrent, Parallel:** Nhiều CPU core xử lý các tác vụ khác nhau nhưng theo trình tự tuần tự trong từng core. Ví dụ, Task 1 chạy ở Core 1, Task 2 chạy ở Core 2.
- **Concurrent, Parallel:** Nhiều CPU core xử lý đồng thời nhiều tác vụ, và mỗi core còn có thể chạy nhiều tác vụ xen kẽ. Đây là mô hình hiệu quả nhất, vừa tận dụng đa nhân vừa tăng tốc độ xử lý.

Tóm lại:

- **Synchronous vs Asynchronous** liên quan đến cách xử lý tác vụ **có chờ hay không chờ kết quả**.
- **Concurrent vs Parallel** liên quan đến **cách tổ chức và phân bổ CPU core** để chạy nhiều tác vụ.

b. Async/Await Example

Ví dụ cho Synchronous, khi ta thực hiện nghiên cứu một bài báo, ta cần phải đợi cho model huấn luyện xong thì mới viết phần kết quả thực nghiệm được, rồi mới viết sang phần các phương pháp nghiên cứu. Quá trình tuyến tính này có khoảng dừng giữa các bước nên sẽ có sự lãng phí. Áp dụng Asynchronous, trong thời gian chờ kết quả từ model, ta sẽ viết mô tả cho các phương pháp nghiên cứu trước. Trong quá trình đó nếu thu được kết quả từ mô hình, ta sẽ chuyển sang viết phần kết quả thực nghiệm rồi lại tiếp tục huấn luyện mô hình và lặp lại các bước trên. Điều này sẽ giúp tối ưu hóa quá trình làm việc.

Với `async/await` (Concurrent, chạy bất đồng bộ)

```
1 import asyncio
2 import time
3
4 async def fetch_data():
5     print("Hi")
6     await asyncio.sleep(3)
7     print("Bye")
8
9 async def main():
10     start_time = time.perf_counter() # Start the timer
11
12     # Create a list of tasks
13     tasks = [fetch_data() for _ in range(2)]
14
15     # Run tasks concurrently
16     await asyncio.gather(*tasks)
17
18     end_time = time.perf_counter() # End the timer
19     print(f"Total time taken: {end_time - start_time:.2f} seconds")
20
21 # Run the main function
22 asyncio.run(main())
```

Kết quả: In ra "Hi" hai lần, "Bye" hai lần, và tổng thời gian chỉ khoảng **3 giây** vì các tác vụ chạy song song (concurrent).

Với không có `async/await` (Sequential, chạy tuần tự)

```
1 import time
2
3 def fetch_data():
4     print("Hi")
5     time.sleep(3)
6     print("Bye")
7
8 def main():
9     start_time = time.perf_counter() # Start the timer
10
11     # Execute tasks sequentially
12     for _ in range(2):
13         fetch_data()
14
15     end_time = time.perf_counter() # End the timer
16     print(f"Total time taken: {end_time - start_time:.2f} seconds")
17
18 # Run the main function
19 if __name__ == "__main__":
20     main()
```



Kết quả: In ra "Hi" và "Bye" xen kẽ, mất khoảng **6 giây** vì mỗi tác vụ phải chờ xong mới đến tác vụ tiếp theo.

- Với `async/await`, các tác vụ I/O được chạy đồng thời \Rightarrow tiết kiệm thời gian.
- Không có `async/await`, các tác vụ chạy tuần tự \Rightarrow tổng thời gian lâu hơn.

Quay trở lại FastAPI, lí do vì sao chúng ta nên sử dụng FastAPI là vì:

- Dễ hiểu, dễ triển khai bởi cú pháp và source code ngắn
- Nhanh chóng bởi framework xử lí phía sau được xây dựng rất tốt nên hiệu suất của FastAPI khá cao.

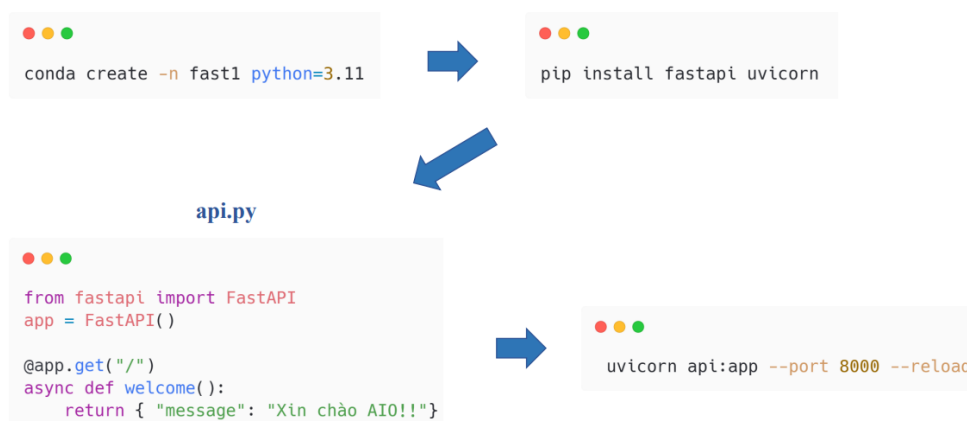
Đây là bảng so sánh giữa ba Framework phổ biến: *django*, *Flask* và *FastAPI*

	django	 Flask	 FastAPI
Performance speed	Normal	Faster than Django	The fastest out there
Async support	YES with restricted latency	NO needs Asyncio	YES native async support
Packages	Plenty for robust web apps	Less than Django for minimalistic apps	The least of all for building web apps faster
Popularity	The most popular	The second popular	Relatively new
Learning	Hard to learn	Easier to learn	The easiest to learn

II. Simple CRUD App

2.1. APIRouter

Cách khởi tạo môi trường để bước đầu xây dựng một app FastAPI:



Khi chạy ứng dụng bằng `uvicorn`, nếu gặp lỗi `ModuleNotFoundError` do không truy cập đúng đường dẫn, ta cần cấu hình lại biến môi trường `PYTHONPATH`.



```
uvicorn api:app --port 8000 --reload
```

```
# Handle Module Not Found
Linux: export PYTHONPATH=$PYTHONPATH:Parent_folder_name
cmd: set PYTHONPATH=%PYTHONPATH%;Parent_folder_name
powershell: $env:PYTHONPATH = "$env:PYTHONPATH;Parent_folder_name"
```

```
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [7564] using StatReload
INFO: Started server process [26092]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:57442 - "GET / HTTP/1.1" 200 OK
INFO: 127.0.0.1:57442 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:57465 - "GET / HTTP/1.1" 200 OK
```

```
{
  "message": "Xin chào AI0!!"
}
```

*APIRouter

Hãy hình dung ứng dụng (**app**) giống như một ngôi nhà, còn **APIRouter** chính là những căn phòng bên trong. Mỗi router có nhiệm vụ xác định xem yêu cầu (request) từ bên ngoài muốn gì và sẽ dẫn người dùng đến đúng “phòng” tương ứng. Ví dụ, nếu có người muốn đi vệ sinh thì router sẽ đưa họ đến phòng tắm.

Bản thân **app** cũng có thể trực tiếp tạo ra các *endpoint*, nhưng nếu số lượng endpoint ngày càng nhiều thì sẽ khó quản lý. Lúc này, việc sử dụng **APIRouter** giúp chia nhỏ, tổ chức các endpoint rõ ràng và gọn gàng hơn.

Một điểm quan trọng cần nhớ: sau khi định nghĩa các endpoint trong router, chúng ta phải gọi `app.include_router(router)` để “đưa các phòng vào trong ngôi nhà”, tức là tích hợp router vào trong ứng dụng chính. Nếu bỏ qua bước này, các endpoint trong router sẽ không được ứng dụng nhận diện.

2.2. HTTP Request Methods

Trong lập trình web, các phương thức HTTP thường được ánh xạ trực tiếp với các thao tác CRUD (Create, Retrieve, Update, Delete) trên dữ liệu:

- **POST (Create):** Thêm dữ liệu mới vào trong hệ thống lưu trữ.
- **GET (Retrieve):** Lấy hoặc truy vấn dữ liệu đã có từ hệ thống.
- **PUT (Update):** Cập nhật và thay thế toàn bộ dữ liệu hiện tại.
- **PATCH (Update):** Chỉ sửa đổi một phần dữ liệu, không thay thế toàn bộ.
- **DELETE (Delete):** Xóa dữ liệu khỏi hệ thống.

Tóm lại:

- Các phương thức HTTP chính là cầu nối để client tương tác với server.
- Chúng giúp chuẩn hóa cách thức thao tác dữ liệu, làm cho API dễ hiểu, dễ bảo trì và tuân thủ quy tắc RESTful.

2.3. Pydantic Model

Sau khi tìm hiểu về **APIRouter**, giờ chúng ta sẽ tạo một router để quản lý thông tin sách trong hệ thống.

model.py

```

from pydantic import BaseModel

class Book(BaseModel):
    id:int
    Name:str
    Author:str
    Publishers:str

curl -X 'POST' \
  'http://localhost:8080/book' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{"id":1,"item":"Mastery"}

```

book.py

```

from fastapi import APIRouter
from model import Book
book_router = APIRouter()

shelf = []

@book_router.get("/book")
async def retrieve_book():
    return {"books": shelf}

@book_router.post("/book")
async def add_book(book:Book):
    shelf.append(book)
    return {"message": "Add successfully!"}

```

Đầu tiên, ta định nghĩa một **Book model**(`model.py`), trong đó mỗi cuốn sách có các thuộc tính: `id`, `Name`, `Author`, `Publishers`. Tiếp theo, ta xây dựng **router sách** (`book.py`):

- **GET /book**: Lấy toàn bộ danh sách sách hiện có (`retrieve_book`).
- **POST /book**: Thêm một cuốn sách mới vào kệ sách (`add_book`).

Các request có thể được gửi đến endpoint này bằng công cụ như `curl` với command line hoặc Postman (UI documentation).

Ví dụ: gửi một POST request để thêm sách mới với thông tin dạng JSON.

Nhờ `APIRouter`, ta có thể dễ dàng tách biệt và tổ chức code quản lý sách, đồng thời có thể mở rộng thêm nhiều router khác trong ứng dụng.

Từ đây, **Pydantic Model** sẽ được sử dụng để validate lại schema của chúng ta, đảm bảo rằng dữ liệu truyền vào đúng định dạng và bao gồm đầy đủ các trường đã định nghĩa (`id`, `Name`, `Author`, `Publishers`).

Nếu dữ liệu nhập vào không hợp lệ (ví dụ thiếu trường, sai kiểu dữ liệu), FastAPI sẽ tự động trả về lỗi 422 cùng với thông báo chi tiết. Điều này giúp giảm thiểu việc viết tay logic kiểm tra, và đảm bảo API luôn nhận và xử lý dữ liệu một cách an toàn.

Lưu ý: Ta có thể nhập **thừa field** và model sẽ tự cắt bớt để phù hợp với khối validation, tuy nhiên ta không thể nhập thiếu field và chương trình sẽ báo lỗi.

❖ Pydantic Model

model.py

```

from pydantic import BaseModel

class Book(BaseModel):
    id:int
    Name:str
    Author:str
    Publishers:str

curl -X 'POST' \
  'http://localhost:8080/book' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{"id":1,"item":"Mastery"}

```

book.py

```

from fastapi import APIRouter
from model import Book
book_router = APIRouter()

shelf = []

@book_router.get("/book")
async def retrieve_book():
    return {"books": shelf}

@book_router.post("/book")
async def add_book(book:Book):
    shelf.append(book)
    return {"message": "Add successfully!"}

```

Trong ví dụ:

- Khi gọi POST /book với JSON hợp lệ, dữ liệu sẽ được lưu vào **shelf** và trả về thông báo "Add successfully!". So với ví dụ trên, lần này **book** trong hàm **add_book** đã được trỏ đến object **Book** tạo bởi Pydantic model để thực hiện việc validate format
- Khi gọi GET /book, ta sẽ nhận về toàn bộ danh sách sách đã được thêm.

2.4. Nested Model

Khi sử dụng **Nested Model** trong Pydantic, chúng ta có thể lồng một **BaseModel** bên trong một **BaseModel** khác. Ví dụ: **Book_nested** chứa trường **item**, và **item** lại là một đối tượng **Item**.

❖ Nested Model

model.py

```
# Nested
class Item(BaseModel):
    Name:str
    Author:str
    Publishers:str

class Book_nested(BaseModel):
    id:int
    item:Item

{
  "id": 1,
  "item": {
    "Name": "Mastery",
    "Author": "Robert Greene",
    "Publishers": "Penguin Books"
  }
}
```

book.py

```
from fastapi import APIRouter
from model import Book, Book_nested
book_router = APIRouter()

shelf = []

@book_router.get("/book")
async def retrieve_book():
    return {"books": shelf}

@book_router.post("/book")
async def add_book(book:Book_nested):
    shelf.append(book)
    return {"message": "Add successfully!"}
```

Điều này khá giống với khái niệm **Composition** trong lập trình hướng đối tượng (OOP), khi một lớp có thể được xây dựng bằng cách kết hợp nhiều lớp nhỏ hơn.

Cách làm này mang lại lợi ích:

- Giúp cấu trúc dữ liệu rõ ràng, có thể tái sử dụng từng model riêng lẻ.
- Dễ dàng mở rộng khi schema trở nên phức tạp (ví dụ: một sách có thể chứa nhiều chi tiết khác nhau).
- Tăng tính trừu tượng: thay vì xử lý toàn bộ dữ liệu dạng phẳng, ta chia thành các thành phần nhỏ và đóng gói logic kiểm tra (validation) trong từng model.

2.5. Path Params

Đến đây, ta mong muốn rằng mỗi quyển sách đều có một **path** riêng. Điều này cũng giống như khi ta chỉ cần xem thông tin của một đối tượng cụ thể thì chỉ nên truy cập đúng đối tượng đó, thay vì phải hiển thị toàn bộ dữ liệu. Cách làm này giúp tránh tình trạng *overfetching*, đồng thời tăng cường bảo mật vì không để lộ các dữ liệu không cần thiết.

Để giải quyết vấn đề này, ta sử dụng **Path Parameters**. Cụ thể, mỗi cuốn sách sẽ có một **id** duy nhất, và ta định nghĩa endpoint dạng: /book/{book_id}. Khi client gửi request kèm **book_id**, hệ thống sẽ chỉ tìm và trả về đúng cuốn sách có id tương ứng.

❖ Path Params

How to see information of particular book?

```

from fastapi import APIRouter, Path
from model import Book, Book_nested
book_router = APIRouter()

shelf = []

@book_router.get("/book")
async def retrieve_book():
    return {"books": shelf}

@book_router.post("/book")
async def add_book(book: Book_nested):
    shelf.append(book)
    return {"message": "Add successfully!"}

```

```

@book_router.get("/book/{book_id}")
async def get_single_book(book_id: int = Path(...)):
    for book in shelf:
        if book.id == book_id:
            return {
                "book": book
            }
    return {
        "message": "Book ID doesn't exist."
    }

```

book.py

Đến đây, ta đã xây dựng đầy đủ các thao tác CRUD cho đối tượng **Book**:

❖ Full CRUD

```

from fastapi import APIRouter, Path
from model import Book, Book_nested
book_router = APIRouter()

shelf = []

@book_router.get("/book")
async def retrieve_book():
    return {"books": shelf}

@book_router.post("/book")
async def add_book(book: Book_nested):
    shelf.append(book)
    return {"message": "Add successfully!"}

```

```

@book_router.get("/book/{book_id}")
async def get_single_book(book_id: int = Path(...)):
    for book in shelf:
        if book.id == book_id:
            return {
                "book": book
            }
    return {
        "message": "Book ID doesn't exist."
    }

```

book.py

❖ Full CRUD

```

@book_router.put("/book/{book_id}")
async def update_book(book_data: Book_nested, book_id: int = Path(...)):
    for book in shelf:
        if book.id == book_id:
            book.item = book_data.item
            return {
                "message": "book updated successfully."
            }
    return {
        "message": "book with supplied ID doesn't exist."
    }

```

book.py

```

@book_router.delete("/book/{book_id}")
async def delete_single_book(book_id: int):
    for index in range(len(shelf)):
        book = shelf[index]
        if book.id == book_id:
            shelf.pop(index)
            return {
                "message": "book deleted successfully."
            }
    return {
        "message": "book with supplied ID doesn't exist."
    }

@book_router.delete("/book")
async def delete_all_book() -> dict:
    shelf.clear()
    return {
        "message": "books deleted successfully."
    }

```

- **Create (POST /book)**: thêm một cuốn sách mới vào trong **shelf**.
- **Read (GET /book, GET /book/{id})**: lấy toàn bộ danh sách hoặc chỉ một cuốn sách cụ thể dựa trên id.
- **Update (PUT /book/{id})**: cập nhật lại thông tin của một cuốn sách đã có. Trong hàm

`update_book`, ta sử dụng `Book_nested` làm dữ liệu đầu vào. Điều này có nghĩa là khi muốn cập nhật một cuốn sách, request body phải chứa đầy đủ cấu trúc của `Book_nested`

- **Delete (DELETE /book/{id}, DELETE /book):** xóa một cuốn sách theo id hoặc xóa toàn bộ dữ liệu sách.

III: Response, Templating

3.1. Response Model Custom Response Model cho phép chúng ta kiểm soát dữ liệu trả về:

chọn những trường nào được hiển thị và định dạng output ra sao.

Ví dụ trong trường hợp này, id của sách là dữ liệu nhạy cảm, nên ta ẩn nó đi bằng cách chỉ định `response_model`. Khi đó, client chỉ nhận được thông tin cần thiết như `Name`, `Author`, `Publishers`, mà không truy cập được vào id.

❖ Response Model Custom (body)

```
@book_router.get("/book",
                  response_model=BookItems)
async def retrieve_book():
    return {"books": shelf}
```

We want to hide book's ids

```
localhost:8000/book
Pretty-print
{
  "books": [
    {
      "item": {
        "Name": "mastery",
        "Author": "robert",
        "Publishers": "aio"
      }
    },
    {
      "item": {
        "Name": "mastery1",
        "Author": "robert1",
        "Publishers": "aio1"
      }
    }
  ]
}
```

```
class BookItem(BaseModel):
    item: Item

    class Config:
        schema_extra = {
            "example": {
                "item": {
                    "Name": "mastery",
                    "Author": "robert"
                }
            }
        }
```

```
class BookItems(BaseModel):
    books: List[BookItem]

    class Config:
        schema_extra = {
            "example": {
                "books": [
                    {
                        "item": {
                            "Name": "mastery",
                            "Author": "robert",
                            "Publishers": "Publishers"
                        }
                    }
                ]
            }
        }
```

3.2. Response Status

Mỗi request đến API đều được server phản hồi bằng một **status code**, cho biết kết quả xử lý thành công hay thất bại. Ta xem bảng sau:

Code	Category	Meaning	Typical Use Case
200	✔ Success	OK	Standard success response for GET/POST
201	✔ Success	Created	Resource successfully created (e.g., user signup)
204	✔ Success	No Content	Success, no response body (e.g., DELETE request)
301	🟡 Redirect	Moved Permanently	URL permanently moved
302	🟡 Redirect	Found	Temporary redirect
304	🟡 Redirect	Not Modified	Resource cached, no need to re-download
400	🔴 Client Error	Bad Request	Invalid request (wrong JSON, params, etc.)
401	🔴 Client Error	Unauthorized	Authentication required/invalid token
403	🔴 Client Error	Forbidden	User authenticated but not allowed
404	🔴 Client Error	Not Found	Resource doesn't exist
5xx	🔴 Server Error	-	-

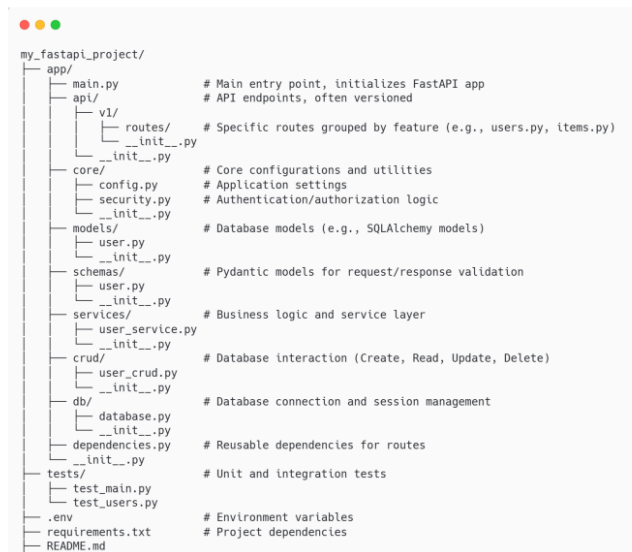
- **2xx – Success:** Request thành công
 - 200: OK (GET/POST thành công).

- 201: Created (tài nguyên mới đã được tạo, ví dụ đăng ký user).
- 204: No Content (thành công nhưng không trả về dữ liệu, ví dụ DELETE).
- **3xx – Redirect:** Yêu cầu được chuyển hướng
 - 301: Moved Permanently – URL đã đổi vĩnh viễn.
 - 302: Found – chuyển hướng tạm thời.
 - 304: Not Modified – dữ liệu chưa thay đổi, dùng cache.
- **4xx – Client Error:** Lỗi từ phía client
 - 400: Bad Request – request sai định dạng (JSON, params...).
 - 401: Unauthorized – chưa xác thực hoặc token không hợp lệ.
 - 403: Forbidden – đã xác thực nhưng không có quyền.
 - 404: Not Found – tài nguyên không tồn tại.
- **5xx – Server Error:** Lỗi từ phía server khi xử lý request.

3.3. FastAPI Structure

Cấu trúc thư mục trong một dự án FastAPI thường được tổ chức thành nhiều phần riêng biệt, giúp mã nguồn rõ ràng, dễ bảo trì và dễ mở rộng. Ta xem hình sau:

❖ FastAPI Structure



- **main.py:** Điểm khởi tạo chính, tạo instance FastAPI và include các router.
- **api/:** Chứa các endpoint, thường được chia theo version (ví dụ **v1/**) hoặc theo module (users, items...).
- **core/:** Các thành phần lõi như **config.py** (cấu hình ứng dụng) **security.py** (xác thực/ủy quyền, ví dụ sinh và kiểm tra JWT token. Ta có thể hiểu nó sẽ có một key bảo mật mà chỉ frontend và backend biết để truy cập).
- **models/:** Định nghĩa các model cho cơ sở dữ liệu (ORM như SQLAlchemy).
- **schemas/:** Pydantic models dùng để validate dữ liệu request/response.

- **services/**: Business logic, nơi xử lý các nghiệp vụ chính của ứng dụng.
- **crud/**: Các thao tác với cơ sở dữ liệu (Create, Read, Update, Delete).
- **db/**: Thiết lập kết nối và session với cơ sở dữ liệu.
- **dependencies.py**: Các dependency có thể tái sử dụng, ví dụ `get_db()`.
- **tests/**: Chứa unit test và integration test để đảm bảo chất lượng hệ thống.

Ví dụ minh họa

Giả sử ta có một module `users`:

- Trong `schemas/user.py` định nghĩa `UserCreate`, `UserResponse`.
- Trong `crud/user_crud.py` viết hàm `create_user(db, user)` để lưu user vào DB.
- Trong `services/user_service.py` thêm logic như kiểm tra trùng email.
- Trong `api/v1/routes/users.py` khai báo endpoint:

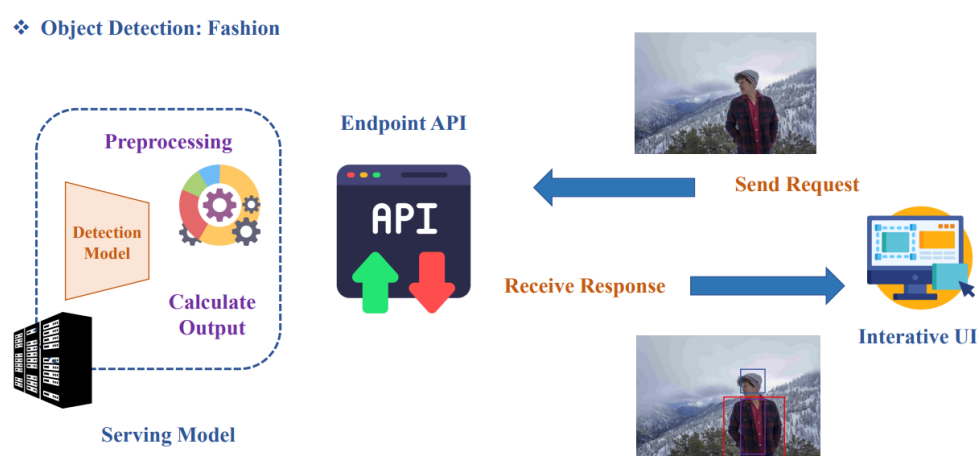
```
@router.post("/users")
async def create_user(user: UserCreate, db: Session = Depends(get_db)):
    return user_service.create_user(db, user)
```

Kết luận: Cách tổ chức này giúp tách biệt rõ ràng giữa *API layer* (endpoint), *schemas* (validation), *services* (business logic), và *crud/db* (cơ sở dữ liệu). Điều này làm dự án dễ mở rộng, bảo trì, và kiểm thử khi hệ thống ngày càng lớn.

Lưu ý: FastAPI ưu tiên access đến end point hơn là path parameters

IV: Case Study: Fashion Detection App (Optional - Xem thêm)

Trong project này, chúng ta xây dựng một hệ thống phát hiện đối tượng thời trang (dựa trên hình ảnh người dùng upload). Quy trình hoạt động bao gồm:



1. Người dùng tải ảnh từ giao diện (**Interactive UI**) và gửi request đến API.
2. Request được chuyển tới **Serving Model**, nơi ảnh được xử lý:
 - **Preprocessing**: Chuẩn hoá ảnh đầu vào (resize, normalize).

- **Detection Model:** Áp dụng mô hình học máy (ví dụ YOLO, Faster R-CNN) để phát hiện các item thời trang.
 - **Calculate Output:** Tính toán bounding box và nhãn đối tượng.
3. Kết quả (bounding box + label) được trả về thông qua **Endpoint API**.
 4. Giao diện người dùng nhận response và hiển thị trực quan: ví dụ áo khoác, mũ, áo sơ mi được khoanh vùng trong ảnh.