Introduction to Jinja and FastAPI

Lab-10(Aaditya Narain, Dileep Adari)

Prerequisites:

1. Basic Python
2. Restful APIs/HTTP
3. JSON format (NoSQL)
4. Async/Await
5. Basic HTML/CSS/JS

What we will cover in this Lab:

1. What is FastAPI and why is it becoming increasing relevant
2. Recap of Concurrency and Parallelism
3. How to install FastAPI
4. Decorators, Routers, Pydantic Models
5. Swagger Docs
6. Jinja Templates
7. Basic Introduction to Cookies, Ports, WebSockets

Additional Resources and Links:

1. https://en.wikipedia.org/wiki/Client%E2%80%93server_model
2. https://fastapi.tiangolo.com/
3. https://www.starlette.io/
4. https://docs.pydantic.dev/latest/#pydantic-examples
5. https://www.geeksforgeeks.org/open-systems-interconnection-model-osi/
6. https://jinja.palletsprojects.com/en/stable/
7. https://www.geeksforgeeks.org/getting-started-with-jinja-template/
8.

How to install fastapi

1. Make sure you have python installed and it's version is greater than 3.6
   a. python3 --version
2. If it's less than 3.6
   a. sudo apt update
   b. sudo apt install -y python3
3. Now install fastapi
   a. pip install fastapi
   b. pip install uvicorn
   c. Pip install jinja2

Do not install unnecessary dependencies!!!

TEST if the install is a success(IN main.py) write:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read_root():

        return {"message": "Hello, FastAPI!"}
```

Run it with: uvicorn main:app

But what is FastAPI actually ???

FastAPI is a tool for building websites and apps that let computers talk to each other. In one sense it is basically a Server Side Software.

Features- Fast, Fewer Bugs, Intuitive, Easy, Short, Standard Based, ML models can run easily, GraphQL and microservices based.

Companies that use FastAPI- Uber, Netflix, Microsoft

College websites that use FastAPI- sports.iiit.ac.in, researchfest.iiit.ac.in

Dependencies- It is dependent on Pydantic and Starlette

But why not Django, Flask, Express, Next or anything else ???

Answer is simple…

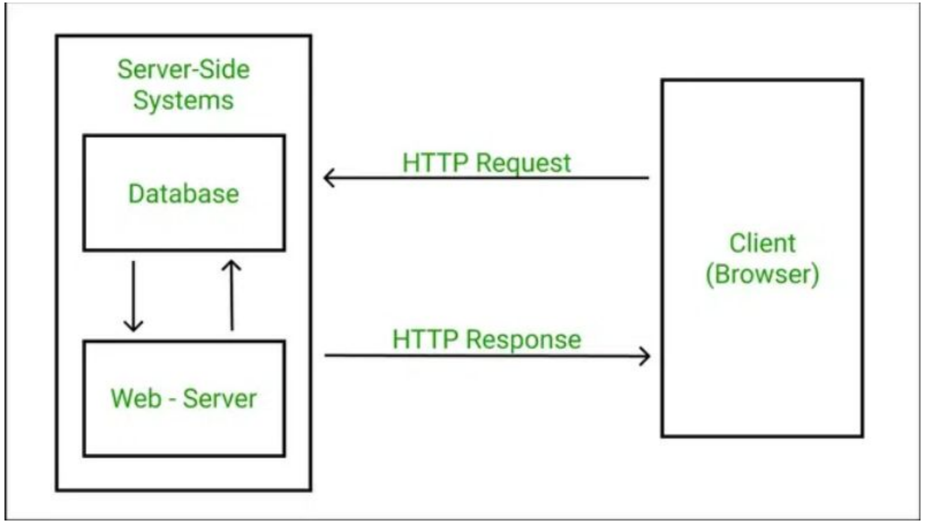
FastAPI is fast, easily understandable.

It has auto-generated documentation and has async support.

You can also run ML models on web easily.

What is HTTP?

- **HTTP (Hypertext Transfer Protocol) is a fundamental protocol** of the Internet, enabling the transfer of data between a client and a server. It is the foundation of data communication for the World Wide Web.
- HTTP provides a standard between a web browser and a web server to establish communication. It is a set of rules for transferring data from one computer to another.
- **HTTPS** is just a secure layer build on top of it.
- HTTP/HTTPS works in the application layer of the OSI model



What are HTTP Methods?

`GET` → Retrieve data

`POST` → Create new data

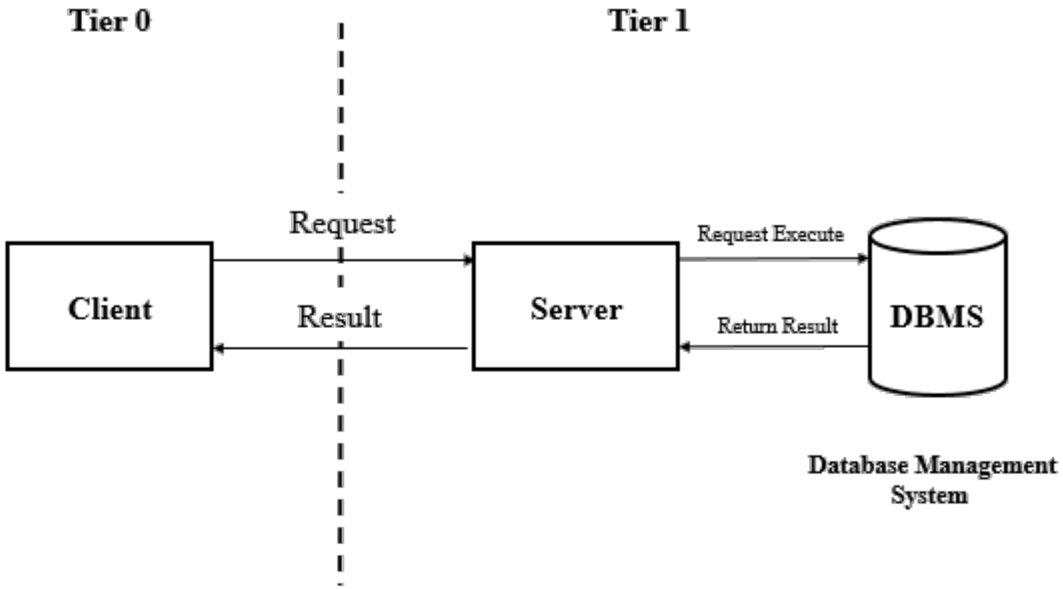
`PUT` → Update existing data

`PATCH` → Partially update data

`DELETE` → Remove data

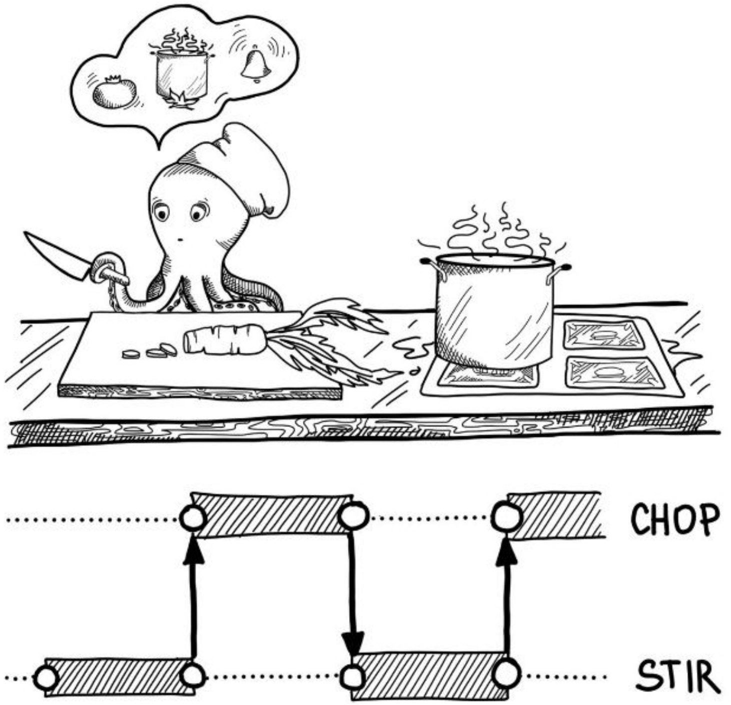
These are nothing but actions that can be performed on resources in a RESTful API. They are crucial in web development, particularly when interacting with APIs.

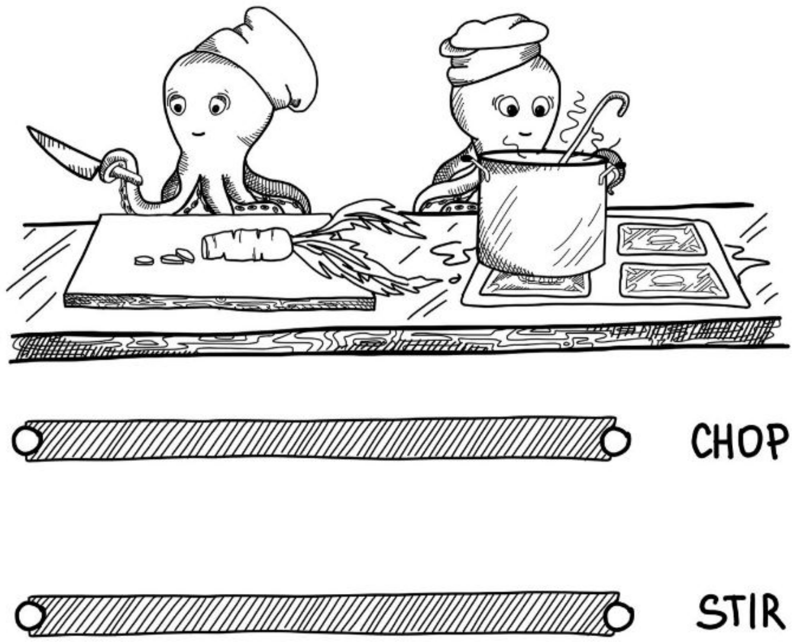
Simple Representation of Client-Server Architecture



Concurrency and Parallelism

What is what?





Source: https://freecontent.manning.com/concurrency-vs-parallelism/

Concurrency Vs Parallelism

Concurrency is about dealing with lot of things at once while parallelism is doing lot of things at once

- **Concurrency:** Running multiple threads/processes at the same time, even on a single CPU by interleaving their executions
- **Parallelism:** Running multiple threads/processes in parallel over different CPU cores
- Concurrent computations can be parallelized without changing correctness of result
- Concurrency by itself does not imply parallelism and vice versa
- Parallelism can be thought of as subclass of concurrency

Some Handy terminal commands

```
lsof -i :<PORT_NUMBER>
```

In the case of the code snippets provided, it will be 8000.

```
kill -9 <PID>
```

To kill the process running on the specific port

DO NOT kill -9 1

Run the code called activity1

What are pydantic Models ???

Prompt given to chatgpt: **if I had to explain to a kid what pydantics are, what would I say**

Pydantic is like your lemonade stand's super-smart helper.

- It makes sure every order (data) is **correct**, **makes sense**, and is **ready to use**.
- So when someone says, "Here's some data," Pydantic checks it, fixes small mistakes, and tells you if something's wrong.

Imagine you're running a lemonade stand, and your job is to make sure everything works perfectly.

- You have a checklist for each order:
    1. How many cups of lemonade? (It must be a number, like `1` or `2`, not a word like "one.")
    2. What flavor? (It must be a word, like `lemon` or `strawberry`.)
    3. Is it iced? (It must be `yes` or `no`.)

Now, when someone gives you an order, your job is to:

1. **Check if the order makes sense**:
    - If someone says "two cups of lemon with ice," that's fine.
    - But if they say "a hundred lemonades with rocks," you say, "That doesn't make sense!"
2. **Fix little mistakes**:
    - If someone says "2 lemonades," but writes the number as a word, you can quietly change "two" into `2` so your lemonade machine understands it.
3. **Remember everything perfectly**:
    - Once an order is checked and fixed, you keep it in your notebook exactly as it should be.

An example

```
class User(BaseModel):
    id : int
    name : str
    age : Optional[int] = None
```

Run the code called activity2

What are decorators?

In Python, **decorators** are special functions that modify or enhance the behavior of other functions or classes without changing their actual code. Think of them as "wrappers" around a function that add extra functionality before or after the function runs.

A Simple Analogy

Imagine you’re baking a cake.

- The cake is your function — it does the main job.
- A decorator is like adding frosting or sprinkles to the cake — it makes the cake look or taste better without changing the actual cake.

An example

```
@app.get(“/”)

def func():

        #do something
```

```
@decorator_function

def actual_function():

        #do something
```

Run the code called activity3

What is Swagger Docs ???

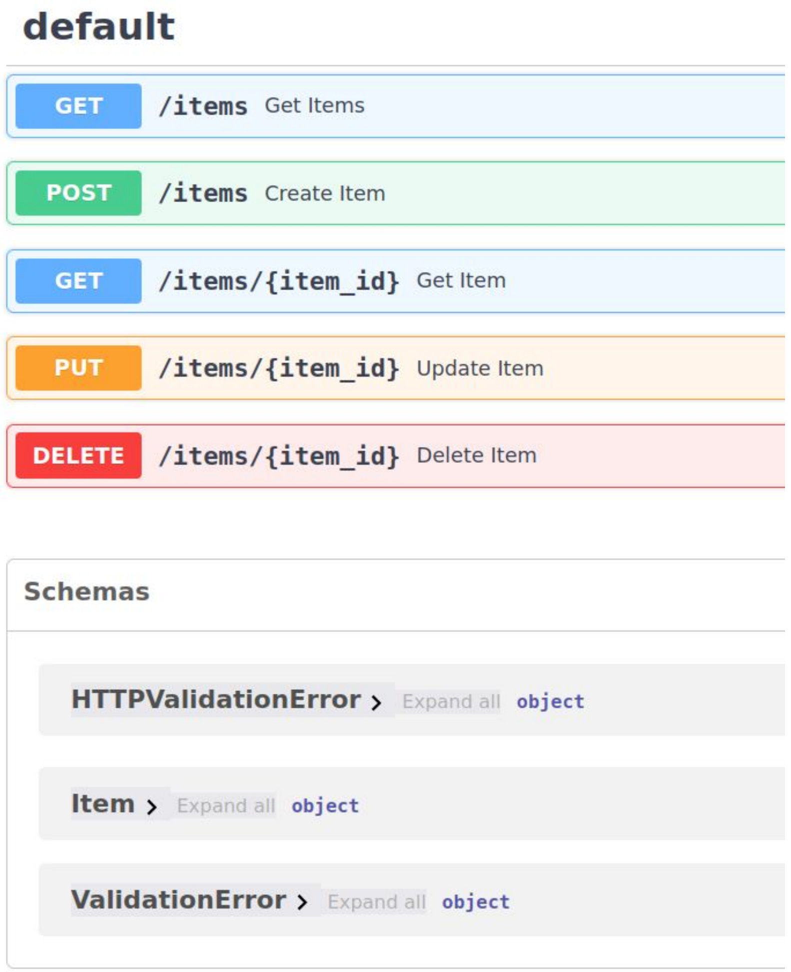
Swagger Docs (also known as **Swagger UI**) are an automatically generated, interactive user interface for APIs. They allow developers and users to:**Understand the A**

PI: See what endpoints are available, their inputs, outputs, and responses.

1. **Test the API**: Interact with the API directly from the browser without needing additional tools like Postman.

`http://127.0.0.1:8000/docs#/`

Accessible from the above URL if you are running on localhost



default

GET /items Get Items

POST /items Create Item

GET /items/{item_id} Get Item

PUT /items/{item_id} Update Item

DELETE /items/{item_id} Delete Item

Schemas

HTTPValidationError › Expand all object

Item › Expand all object

ValidationError › Expand all object

Run the code called activity4

Look at example1

What are FastAPI routes ???

In FastAPI, **routes** are like **directions** that tell the web server how to handle different requests coming from users. Each route is a specific **path** or **endpoint** that the user can visit (like a webpage), and it's associated with a **function** that will run when someone visits that path.

Think of a route like a **delivery address** for the server. The server has a bunch of addresses (routes), and when you tell it where to go (make a request), it delivers the right **response** based on that address.

A simple example of route would be post office addresses.

Endpoint examples:

/api

/api/get-all

/xyz/a

/xyz/b

What are path parameters ???

Path parameters are placeholders in a URL that allow you to capture values from the URL and use them in your application. These values are dynamic and typically represent specific resources or identifiers.

```
@app.delete("/todos/{todo_id}")

def delete_todo(todo_id: int):

        pass
```

Do activity5

Look at example2

Surprise Quiz

(take out your notebook)

~~Surprise Quiz~~

(Just Kidding)

Jinja Templates

What is Jinja???

Jinja is a **templating engine** for Python, commonly used with web frameworks like **FastAPI**, **Flask**, and **Django** (via its template layer). It enables the creation of **dynamic HTML pages** by embedding Python-like syntax within HTML.

Insert **variables** into your HTML

Use **control structures** like loops and conditionals

Implement **filters** and **macros** for advanced functionality

Manage **template inheritance** for improved modularity

Basic things you need for Jinja Templates

{{ … }}	Outputs a **variable** or **expression** result
{% … %}	Execute a **Control statement** (for , if)
{# … #}	Used for **Comments**

Base Template with blocks

Jinja has this cool feature where you can use the base template and add blocks inside it which you can extend it on other files.

Check out base.html and inheritance.html in the given jinja example.

Run the main.py in jinja template

app.mount() & Static Files

```
app.mount("/static", StaticFiles(directory="static"),
name="static")
```

`app.mount()` → This method is used in **FastAPI** to mount a **separate application** or a **static file directory** under a specific path.

`StaticFiles` is a FastAPI utility that serves static files like CSS, JS, images, etc.

Jinja2Templates

```
templates = Jinja2Templates(directory="templates")
```

`Jinja2Templates` → This is a FastAPI utility for integrating **Jinja2**, a powerful templating engine.

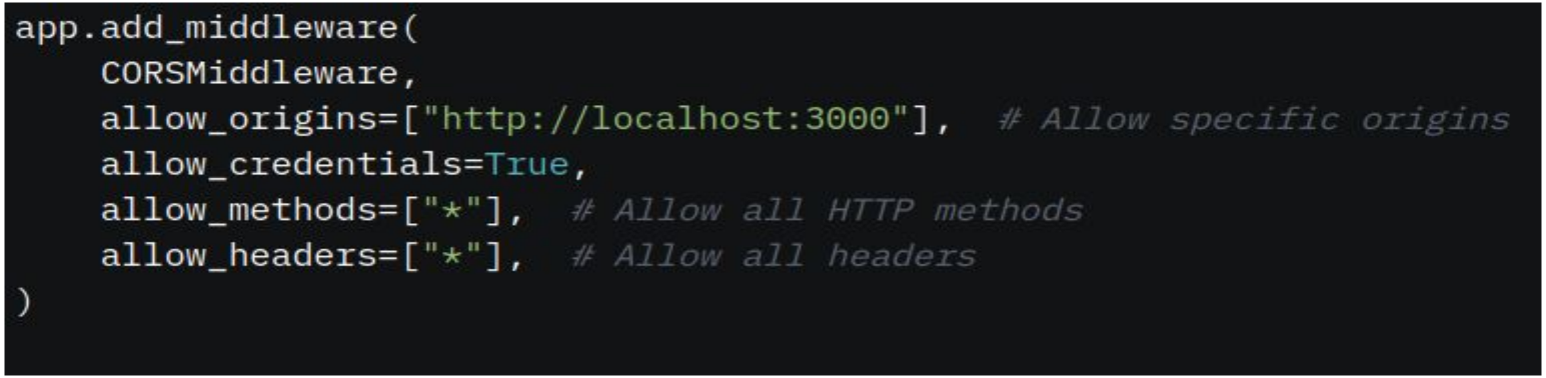
`directory="templates"` → Specifies the folder where your **`.html`** template files are stored.

Keep a Note of directory structure in the given example

CORS

CORS is a policy enforced by web browsers to control which domains (origins) can access resources (e.g., APIs, files) from your server. It comes into play when:

- A frontend (e.g., JavaScript running in a browser) makes requests to a backend server.
- The frontend and backend are hosted on **different origins** (e.g., http://frontend.com vs. http://api.backend.com).
- from fastapi.middleware.cors import CORSMiddleware



```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],  # Allow specific origins
    allow_credentials=True,
    allow_methods=["*"],  # Allow all HTTP methods
    allow_headers=["*"],  # Allow all headers
)
```

JWT Tokens (pip install PyJWT python-decouple)

JSON Web Token (JWT) is a secure way to transmit authentication data.

JWTs are used for authentication and session management.

Consist of three parts: Header, Payload, and Signature.

Header: Contains metadata about the token.

Payload: Contains claims (e.g., user ID, roles).

Signature: Ensures the token integrity.

JWT Implementation in FastAPI

```
import time
from typing import Dict
import jwt
from decouple import config

JWT_SECRET = config("secret")
JWT_ALGORITHM = config("algorithm")

def sign_jwt(user_id: str) -> Dict[str, str]:
        payload = {
                "user_id": user_id,
                "expires": time.time() + 600
        }
        token = jwt.encode(payload, JWT_SECRET, algorithm=JWT_ALGORITHM)
        return token_response(token)
```

JWT Implementation in FastAPI

```
import time
import jwt
from decouple import config

JWT_SECRET = config("secret")
JWT_ALGORITHM = config("algorithm")

def decode_jwt(token: str) -> dict:
        try:
              decoded_token = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
              return decoded_token if decoded_token["expires"] >= time.time() else None
        except:
              return {}

def verify_jwt(self, jwtoken: str) -> bool:
        isTokenValid: bool = False
        try:
            payload = decode_jwt(jwtoken)
        except:
            payload = None
        if payload:
            isTokenValid = True
        return isTokenValid
```

Using JWT for Authentication

```
from fastapi import Request, HTTPException
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

from .auth_handler import decode_jwt, verify_jwt

class JWTBearer(HTTPBearer):
    def __init__(self, auto_error: bool = True):
        super(JWTBearer, self).__init__(auto_error=auto_error)

    async def __call__(self, request: Request):
        credentials: HTTPAuthorizationCredentials = await super(JWTBearer, self).__call__(request)
        if credentials:
            if not credentials.scheme == "Bearer":
                raise HTTPException(status_code=403, detail="Invalid authentication scheme.")
            if not verify_jwt(credentials.credentials):
                raise HTTPException(status_code=403, detail="Invalid token or expired token.")
            return credentials.credentials
        else:
            raise HTTPException(status_code=403, detail="Invalid authorization code.")
```

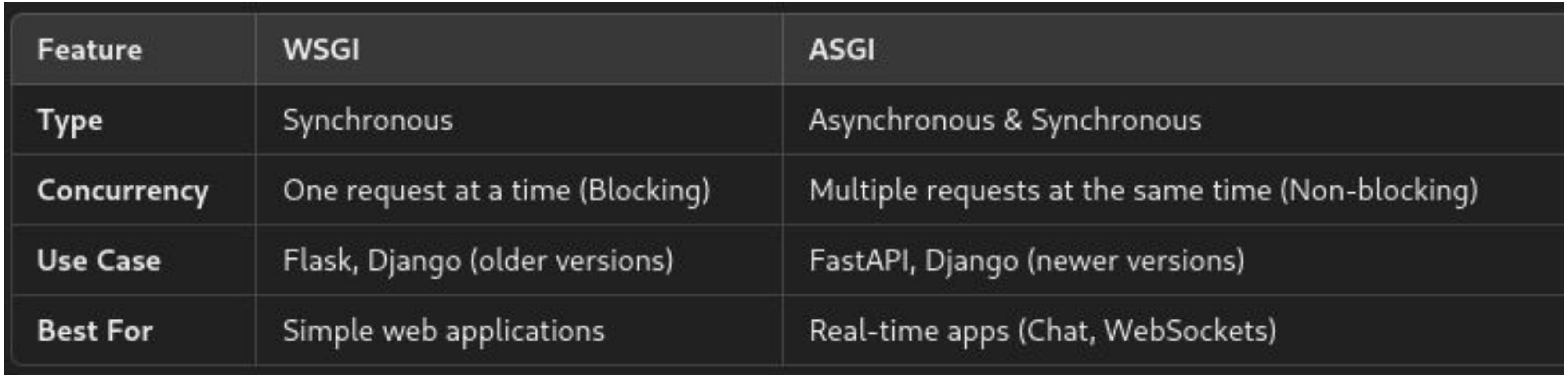
Using JWT for protecting routes

```
from fastapi import FastAPI, Body, Depends


from app.auth.auth_bearer import JWTBearer
from app.auth.auth_handler import sign_jwt
from app.model import PostSchema, UserSchema, UserLoginSchema


@app.post("/posts", dependencies=[Depends(JWTBearer())], tags=["posts"])
async def add_post(post: PostSchema) -> dict:
    post.id = len(posts) + 1
    posts.append(post.dict())
    return {
        "data": "post added"
    }
```

What are WSGI and ASGI

WSGI (**Web Server Gateway Interface**) and ASGI (**Asynchronous Server Gateway Interface**) are **protocols** that allow Python web frameworks (like FastAPI and Flask) to communicate with web servers (like Gunicorn and Uvicorn).

Think of them as **middlemen** between your **Python web application** and the **server** that serves your web requests.



Feature	WSGI	ASGI
Type	Synchronous	Asynchronous & Synchronous
Concurrency	One request at a time (Blocking)	Multiple requests at the same time (Non-blocking)
Use Case	Flask, Django (older versions)	FastAPI, Django (newer versions)
Best For	Simple web applications	Real-time apps (Chat, WebSockets)

WSGI (Web Server Gateway Interface)

WSGI follows a **synchronous (blocking)** request-handling model:

1. The client (browser) makes a request.
2. The server (**Gunicorn** or **uWSGI**) receives it.
3. WSGI waits for a response before handling the next request (one at a time).
4. Once processed, the response is sent back to the client

```
import time
from flask import Flask

app = Flask(__name__)

@app.route("/slow")
def slow():
        time.sleep(5)  # Blocks for 5 seconds
        return "Done"
```

ASGI (Asynchronous Server Gateway Interface)

ASGI supports both **synchronous and asynchronous** operations:

1. The client sends a request.
2. ASGI server (**Uvicorn**) processes multiple requests at the same time.
3. Supports **WebSockets**, background tasks, and async operations.

```
import asyncio
from fastapi import FastAPI

app = FastAPI()

@app.get("/fast")
async def fast():
    await asyncio.sleep(5)  # Doesn't block other
requests
    return {"message": "Done"}
```