NAVYA NARAYAN PANICKER

WEEK 3

DAY 1: 06/10/25 [MONDAY]

TASK : Setup Flask project; define folder structure (app folder, routes, static etc.); basic "Hello world" route, Implement GET /tasks and POST /tasks with dummy in-memory storage; test via Postman or curl

QUESTIONS/REFLECTIONS:

1. What is the flow when a HTTP GET request comes to Flask → how Flask handles routing → response.
    - First the client sends a request to the flask server.
    - Flask runs on a web server called Werkzeug.
    - When the request reaches Flask, it checks:
    - The **HTTP method** (GET, POST, PUT, DELETE, etc.)
    - The **URL path** (`/tasks`, `/users`, `/`, etc.)
    - Flask will compare the request url and the route, it matches it will execute the function otherwise it will send a 404 not found error.
    - Once it finds the correct route then it will execute that function.
    - The python code is converted into http response.
    - Then finally the response is sent back to the client.

2. What is WSGI? How Flask sits on WSGI server (development vs production).
    - Stands for web server gateway interface
    - Flask on the whole is an wsgi application.
    - from flask import Flask
      app = Flask(__name__)
      its creating a wsgi application object.

      Development:
      When you run:
      python app.py
    - Flask starts its **built in wsgi**

      production:
      when you run:
      gunicorn app:app
    - a real wsgi server is used.

3. How you'd structure a medium size API project (folders, modules).

```
my_api_project/
|
 ├── app/
|    ├── __init__.py
|   ├── config.py
|   ├── models/
|   |   ├── __init__.py
|   |   └── user.py
|   ├── routes/
|   |   ├── __init__.py
|   |   ├── auth.py
|   |   └── tasks.py
|   ├── services/
|   |   ├── __init__.py
|   |   └── task_service.py
|   ├── schemas/
|   |   ├── __init__.py
|   |   └── task_schema.py
|   ├── utils/
|   |   ├── __init__.py
|   |   └── helpers.py
|   └── extensions.py
|
├── tests/
|   ├── __init__.py
|   ├── test_auth.py
|   └── test_tasks.py
|
├── migrations/
|
├── requirements.txt
├── run.py
└── README.md
```

- **app/** → Core application code
- **__init__.py** → Initialize Flask app, register blueprints, and load config.
- **config.py** → Holds configuration (dev, test, prod).
- **models/** → ORM models (like SQLAlchemy models for Users, Tasks, etc.).
- **routes/** → Flask Blueprints for endpoints grouped by feature.
- **services/** → Business logic separate from routes.
- **schemas/** → Validation/serialization schemas (e.g., Marshmallow or Pydantic).
- **utils/** → Helper functions, common utilities.
- **extensions.py** → Extensions like DB, JWT, CORS, etc.
- **tests/** → Unit and integration tests.
- **migrations/** → Database migrations (if using Alembic/Flask-Migrate).
- **requirements.txt** → Dependencies.
- **run.py** → Entry point to run the app.

4. What is REST: what makes an endpoint RESTful?
    - **REST** stands for **Representational State Transfer.**

- used to design network applications like web API.

  Endpoint:
- use proper url.
- Use proper http methods(put/post/delete etc)
- Returns standard https status codes.
- Use json for request and response.

5. What status codes should be returned and when?
    - **3 digit number** returned by a server in response to a client's HTTP request.
    - 400- bad request- invalid request data, missing parameters, or malformed json
    - 404- not found- resource not found
    - 405- method not allowed-http method not allowed for the end point, eg: POST on /users/123
    - 409- conflict- resource conflict , eg:duplicate entry

6. How to validate input and handle bad JSON or missing fields.
    - If the client sends a bad json the server should be able to find the error.
    - Make sure that all the fields are present.
    - The format should also be checked.
    - Use appropriate libraries
    - Use error handling techniques like 404 etc.
    - Never trust the clients input always check for the server.