AEROASPIRE INTERN ASSESSMENT-01
KARTHIK TL

1. What is the flow when a HTTP GET request comes to Flask → how Flask handles routing → response.

When a browser or client sends a GET request, Flask's WSGI server receives it first. Flask then checks its routing map — built from all the @app.route() decorators — to match the request path to a view function. Once a match is found, Flask calls that function.

If the route defines query parameters, they can be accessed with request.args.get.The view function then constructs and returns a response object (often a string, template, or JSON). Finally, Flask packages it into a proper HTTP response and sends it back to the client.

The flow is essentially: client → WSGI server → Flask router → view function → response → client. GET requests are idempotent, meaning they only retrieve data without changing server state.

2. What is WSGI? How Flask sits on WSGI server (development vs production).

WSGI (Web Server Gateway Interface) is the standard interface between a Python web framework and the web server. It acts as a bridge so the web server can talk to the Flask app.

- Development environment: Flask's built-in server (flask run --debug) uses Werkzeug's lightweight WSGI server — convenient for local testing but not meant for production.
- Production environment: You plug Flask into a full WSGI server (for example, Gunicorn) that handles concurrency, scaling, and HTTP connections. The WSGI server receives the request, translates it into a WSGI environment, calls the Flask app, gets the response, and passes it back to the client.

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

A clean structure keeps code maintainable and testable. I'd usually organize it like this:

text

project/

├── app/

| ├── __init__.py # initializes Flask app and registers blueprints

| ├── config.py # config variables

| ├── models/ # database models

| ├── routes/ # route controllers (organized by feature)

| ├── schemas/ # for serialization/validation (e.g., Marshmallow)

```
|   ├── services/        # business logic layer
|   └── tests/           # unit tests
├── run.py               # app entry point
└── requirements.txt     # dependencies
```

Each feature (like users or posts) gets its own route, model, and schema module connected via blueprints, helping modularize the app.

4. What is REST: what makes an endpoint RESTful?

A RESTful endpoint follows certain principles:

- Uses resources (nouns), not actions; e.g., /api/users instead of /api/getUsers.
- Uses HTTP methods to describe the action, e.g., GET to read, POST to create, PUT/PATCH to update, DELETE to remove.
- Is stateless — each request carries all the information needed (no sessions).
- Uses consistent URIs and returns clear HTTP status codes.
  Following these makes your API predictable and easy for clients to integrate with.

5. What status codes should be returned and when?

Success responses (2xx)

- 200 OK – The standard for successful requests like GET or successful PUT/PATCH updates.
- 201 Created – Used when something new was created through a POST request. I usually include a Location header or return the created resource.
- 202 Accepted – For long-running asynchronous jobs; it means I've accepted the request but haven't finished processing it.
- 204 No Content – I return this when the operation succeeded but there's nothing to send back (for example, DELETE requests).

Client errors (4xx)

- 400 Bad Request – The client sent invalid data or malformed JSON. This is my default for syntax or validation errors.
- 401 Unauthorized – The request needs valid authentication (like a missing or invalid token).
- 403 Forbidden – The user is authenticated but doesn't have permission to do the action.

- 404 Not Found – The requested resource doesn't exist or the route is wrong.
- 409 Conflict – There's a resource conflict, like trying to create a duplicate record.
- 422 Unprocessable Entity – I use this when the request format is correct but some fields fail validation (for example, an invalid email format).

Server errors (5xx)

- 500 Internal Server Error – Something failed on the server that wasn't the client's fault.
- 503 Service Unavailable – The service is down temporarily or overloaded

6. How to validate input and handle bad JSON or missing fields.

When I build a Flask API and expect JSON input, the first thing I do is try to parse the JSON safely. If the client sends bad or missing JSON, Flask will throw an error, so I catch that and respond with a 400 error telling them their JSON is invalid or missing.

Then, I check that the required fields are actually there. For simple cases, I do quick checks like "Is this field present? Is it the right type?" If something's missing or wrong, I send back a 400 error explaining exactly what's missing or invalid.

For bigger projects, I use libraries like Marshmallow or Pydantic that let me define a schema with the expected fields and their types. These libraries do the validation automatically and give me clear error messages when something's off.

Overall, I focus on failing fast and giving helpful error messages so clients know exactly what to fix. It helps keep the API stable and saves time debugging.