**🧠 PHASE 1: Foundation (5–7 days max)**

**✅ Step 1: Web API Theory (You're doing now)**

**Topics:**

* What is an API
* What is a REST API
* HTTP methods + status codes
* Client-server model
* Request-response cycle

**Output:**  
Write your own summary in a web\_api\_notes.md file in your repo. Don't move forward until you can explain the basics without stuttering.

**✅ Step 2: requests Module (2–3 days)**

**Topics:**

* requests.get, .post, .put, .delete
* Sending headers, query params, JSON body
* Reading response.status\_code, .json(), .text

**Output:**  
Create a Python script that:

* Fetches data from a public API (e.g., https://jsonplaceholder.typicode.com/posts)
* Posts some dummy data
* Updates it
* Deletes it

Push this to GitHub under /requests-practice

**✅ Step 3: os + json Modules (2–3 days)**

**Topics:**

* os.path, os.getcwd(), os.environ, os.remove
* Reading and writing JSON files with json.load, json.dump

**Output:**

* Script that loads data from a .json file, modifies it, and saves it
* Use .env file + os.environ.get() to simulate secret handling

Push under /os-json-practice

**🔧 PHASE 2: Flask & Backend (2–3 weeks)**

**✅ Step 4: Flask Basics**

**Topics:**

* Flask, request, jsonify
* Routes (@app.route), HTTP methods
* Returning JSON, handling errors

**Output:**

* A Flask app with 3–4 routes (GET, POST, PUT, DELETE)
* Use in-memory dict or list as fake DB

Push to /flask-api-basics

**✅ Step 5: MongoDB Integration**

**Topics:**

* pymongo, MongoClient
* CRUD operations with MongoDB
* Connecting Flask to Mongo

**Output:**

* Convert the above Flask app to use MongoDB
* Store real data from client

Push to /flask-mongo-api

**✅ Step 6: Authentication (JWT or token-based)**

**Topics:**

* Flask-JWT-Extended or token system
* Login/signup routes
* Token generation, protected routes

**Output:**

* Add user system to your API
* Implement @jwt\_required on protected routes

Push to /flask-auth-api

**🧪 PHASE 3: Real Project + Deployment (2 weeks)**

**✅ Step 7: Build a Complete REST API Project**

**Example Ideas:**

* Todo app with users and auth
* Notes app
* Expense tracker

**Requirements:**

* Flask + Mongo
* JWT Auth
* Full CRUD
* .env used properly
* Code modularized (Blueprints, services, routes)

Push to /your-final-api-project

**✅ Step 8: Deployment**

**Platforms:**

* Railway / Render / Fly.io

**Output:**

* API must be live on a public URL
* README with:
  + Base URL
  + Endpoints
  + Example usage

Completed topic notes

**Core API Concepts – Full Summary**

*(Your Theory Notes for Day 1)*

**1️⃣ What is an API?**

🔹 **API (Application Programming Interface)** is a set of rules that allows two software systems to talk to each other.  
🌐 In web dev, APIs let a **client** (browser/app) communicate with a **server** using HTTP and data formats like **JSON**.

📦 Think of it as a **menu** — you don’t know how the kitchen works, but you can order food through the menu. That’s your API.

**2️⃣ What is a REST API?**

🔹 **REST (Representational State Transfer)** is a style of API design that follows these rules:

✅ Uses **HTTP** methods (GET, POST, PUT, DELETE)  
✅ Works with **resources** (like /users/1, /notes)  
✅ Communicates using **JSON**  
🚫 Is **stateless** — the server does not remember previous requests  
📂 Allows clean, predictable access to data via URLs

**3️⃣ HTTP Methods & Status Codes**

**🔧 Common HTTP Methods:**

| **Method** | **Use Case** | **Description** |
| --- | --- | --- |
| GET | 📥 Read | Retrieve data |
| POST | 📝 Create | Add new data |
| PUT | 🛠️ Update (Replace) | Update full resource |
| DELETE | 🗑️ Delete | Remove a resource |

**📟 Common HTTP Status Codes:**

| **Code** | **Meaning** | **When It Happens** |
| --- | --- | --- |
| 200 | ✅ OK | Request succeeded |
| 201 | 🆕 Created | POST request succeeded |
| 400 | ❌ Bad Request | Client sent invalid or incomplete data |
| 401 | 🔒 Unauthorized | Token missing/invalid |
| 403 | 🚫 Forbidden | Valid token, but no access rights |
| 404 | ❓ Not Found | Resource doesn’t exist or wrong URL |
| 500 | 💥 Server Error | Crash/bug on server |

**4️⃣ Client-Server Model**

The **client-server model** separates the system into two roles:

🖥️ **Client** = sends HTTP requests (browser, app, frontend)  
🧠 **Server** = receives request, processes it, and responds with data (usually from a DB)

🔗 They are **independent** systems connected only via **HTTP communication**.  
This separation makes systems **modular**, **reusable**, and **secure**.

**5️⃣ Request-Response Cycle**

The **request-response cycle** is how a client and server interact every time a request is made.

**🔁 Full Flow:**

1. 📨 **Client sends request** (with method, headers, URL, body)
2. 🧠 **Server receives and routes** it to the correct function
3. 📂 **Server runs logic**, talks to the DB if needed
4. 📤 **Server sends back a response** (status + data)
5. 📲 **Client receives the response**, uses it to update UI or logic

**🔚 Example (Python requests):**

🎯 This sends a GET request, receives a 200 response, and prints the returned data.

**✅ requests Module – Day 2 Notes (05 Aug 2025)**

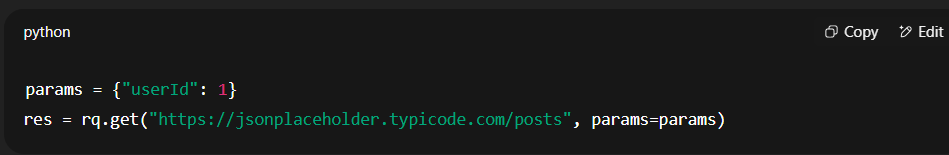
**🔹 Overview:**

Continued learning the Python requests library to understand how to interact with APIs using HTTP methods.

**🧠 Topics Covered Today:**

**✅ GET Request with Query Parameters:**

* Used params to send query parameters.



* Parsed .json() response and used indexing to access specific post data.
* Confirmed status code, URL, and number of posts received.

**✅ POST Request:**

* Sent JSON data using json= keyword.



* Observed that the data isn't persisted — because it's a mock API.
* POST returns a dummy object with an id.

**✅ PUT Request:**

* Updated data at a specific resource ID using .put() method.



* Sent full replacement of data (unlike PATCH).
* Realized PUT works only for existing IDs (1–100) in jsonplaceholder.
* PUT to /posts/1000 returns a 500 error → due to API limitation.

**✅ Error Handling & Debugging:**

* Faced JSONDecodeError when server returned no content.
* Used .text and .headers to inspect raw response.
* Understood the difference between client-side error (400), not found (404), and mock server limitations (500).

**🧪 Key Learnings:**

* Only use .json() when you're sure the response is valid JSON.
* Mock APIs like jsonplaceholder have limits — they don’t persist or validate deeply.
* Use try-except around .json() to handle bad responses.
* PUT updates full resources; invalid IDs on mock APIs may cause fake server errors.