

Day 5 - Session 1:

API Programming with Python

This notebook introduces API (Application Programming Interface) programming using Python. APIs are fundamental for enabling different software systems to communicate and exchange data. In engineering and scientific domains, APIs are crucial for integrating with external data sources, controlling remote instruments, or building distributed systems. We will focus on REST APIs and use the popular requests library to *consume* APIs, and then explore how to *create* your own simple APIs using Flask and FastAPI. We will also include curl commands for testing these APIs from the command line.

1. Introduction to APIs: What is an API?

Imagine you want to order food from a restaurant. You don't go into the kitchen and cook it yourself. Instead, you use a menu (interface) to tell the waiter (the messenger) what you want, and the kitchen (the system) prepares it and sends it back to you.

An API (Application Programming Interface) works similarly in software:

- It's a set of rules and definitions that allows one software application to talk to another.
- It defines how software components should interact.
- It hides the complex internal workings of a system, just showing you what you can request and what you'll get back.

Why are APIs important in Engineering/Science?

- **Data Integration:** Get real-time weather data, map data, satellite orbital data from external services.
- **Device Control:** Send commands to remote instruments or embedded systems (though sometimes specialized protocols are used for this, APIs can sit on top).
- **Automated Workflows:** Connect different tools and services in your data pipeline (e.g., pulling data from a storage service, sending it to an analysis service).
- **Building Applications:** Powering mobile apps, web dashboards, or desktop tools that rely on data from various sources.

2. REST API Overview: The Web's Common Language

REST (Representational State Transfer) is a set of architectural principles for designing networked applications. It's the most common style of API used on the web today. REST APIs use standard web technologies like HTTP and JSON.

2.1 Key REST Principles

- **Client-Server:** Your Python script (the client) sends requests to a web server (the API provider), and the server sends responses back.
- **Stateless:** Each request from your script to the server must contain all the information the server needs to understand it. The server doesn't "remember" past requests from your script.
- **Resources and URLs:** Everything you interact with in a REST API is treated as a **resource**, identified by a unique web address called a **URL**.
 - Example: https://api.example.com/sensors/temp_001 (a specific temperature sensor).
- **HTTP Verbs (Methods): The Actions You Can Take**
HTTP methods (like GET, POST) tell the server what kind of action you want to perform on a resource.

Table: Common HTTP Verbs for REST APIs

HTTP Verb	Purpose	Analogy Example
GET	Retrieve data. (Read-only, doesn't change data on server).	"Give me the current weather report."
POST	Create new data. (Send data to server to make something new).	"Submit a new sensor reading log."
PUT	Update existing data (full replacement). (Completely replaces a resource).	"Replace the entire configuration file."
DELETE	Remove data. (Deletes a specific resource).	"Delete this old log entry."

2.2 JSON: The Data Format

JSON (JavaScript Object Notation) is the most common format for sending and receiving data with REST APIs. It's easy for humans to read and for computers (like Python) to understand. It directly translates to Python dictionaries and lists.

- **Objects:** Like Python dictionaries, represented by `{}`.
`{"sensor_id": "T001", "value": 25.5, "unit": "Celsius"}`
- **Arrays:** Like Python lists, represented by `[]`.
`[
 {"id": "T001", "value": 25},
 {"id": "P002", "value": 10}
]`

3. Consuming APIs with Python's requests Library

The requests library is the best tool in Python for making web requests.

Installation: If you haven't installed it, open your terminal/command prompt and run:

```
Shell  
pip install requests
```

3.1 Basic GET Requests (Getting Data)

GET requests are used to fetch data from a server.

requests.get(): How to Send a GET Request

- **Purpose:** Retrieves data from a website or API.
- **Syntax:** `requests.get(url, params=None, headers=None)`
 - `url`: The web address you want to get data from.
 - `params`: (Optional) A dictionary of extra information to send in the URL (like filters).

Response Object: What You Get Back

When `requests.get()` finishes, it gives you a response object. Key parts of this object are:

- `response.status_code`: A number that tells you if the request was successful (e.g., 200 means OK, 404 means Not Found).
- `response.text`: The raw data from the website/API as a string.
- `response.json()`: If the data is in JSON format, this turns it into a Python dictionary or list.

```
Python
import requests
import json # Used to pretty-print JSON

print("--- Example 1: Basic GET Request ---")

# Example 1: Get a single "todo" item from a public test API
url_todo_item = "https://jsonplaceholder.typicode.com/todos/1"
print(f"Trying to get data from: {url_todo_item}")

try:
    response = requests.get(url_todo_item) # Send GET request
    print(f"Status Code: {response.status_code}") # 200 means success

    # Convert JSON response to a Python dictionary
    todo_item = response.json()
    print("\nResponse as Python Dictionary:\n")
    print(json.dumps(todo_item, indent=2)) # Pretty print JSON

    print(f"\nAccessing specific data: Title = {todo_item['title']}")

except requests.exceptions.RequestException as e:
    print(f"An error occurred while making the request: {e}")
except json.JSONDecodeError:
    print("Error: The response was not valid JSON.")

print("\n--- Example 2: GET Request with Parameters ---")

# Example 2: GET request with query parameters (e.g., filter todos by userId
```

```

and completed status)
url_filter_todos = "https://jsonplaceholder.typicode.com/todos"
my_parameters = {
    "userId": 2,
    "completed": True # Only get completed tasks for userId 2
}

print(f"Trying to get data from: {url_filter_todos} with filters:
{my_parameters}")

try:
    response_filtered = requests.get(url_filter_todos, params=my_parameters)
    print(f"Status Code: {response_filtered.status_code}")

    filtered_todos = response_filtered.json()
    print(f"\nFound {len(filtered_todos)} completed todos for userId 2.")

    if filtered_todos:
        print("First completed todo title:", filtered_todos[0]['title'])

except requests.exceptions.RequestException as e:
    print(f"An error occurred while making the filtered request: {e}")
except json.JSONDecodeError:
    print("Error: The filtered response was not valid JSON.")

```

3.2 Basic POST Requests (Sending Data to Create Something)

POST requests are used to send new data to the server, often to create a new record or resource. The data you send is usually in the "body" of the request.

requests.post(): How to Send a POST Request

- **Purpose:** Sends data to create a new resource on the server.
- **Syntax:** `requests.post(url, json=data_to_send, headers=None)`
 - `url`: The address where you want to send data.
 - `json`: (Optional) A Python dictionary that will be converted to JSON and sent as the request body. This is preferred for most REST APIs.

Python

```
import requests
import json

print("\n--- Basic POST Requests ---")

# URL for the test API (JSONPlaceholder)
url_create_post = "https://jsonplaceholder.typicode.com/posts"

# Data you want to send (like creating a new blog post)
new_data_for_post = {
    "title": "New Sensor Data Log",
    "body": "Daily temperature readings recorded.",
    "userId": 10
}

print(f"Sending new data to create a post at: {url_create_post}")
print(f>Data to send: {new_data_for_post}")

try:
    # Send POST request (json= handles automatic conversion to JSON string)
    response_new_post = requests.post(url_create_post, json=new_data_for_post)

    print(f>Status Code: {response_new_post.status_code}") # 201 = Created

    # Convert the JSON response back to Python dict and pretty print
    created_item = response_new_post.json()
    print(f"\nResponse from server (created item):\n{json.dumps(created_item,
        indent=2)}")

    # Accessing a specific part of the response
    print(f"\n✅ The new post was assigned ID: {created_item['id']}")

except requests.exceptions.RequestException as e:
    print(f>An error occurred during the POST request: {e}")
```

3.3 Simplified Lightweight Data Puller Function

Let's create a simple, reusable function to fetch data from an API. We'll simplify the Open-Meteo weather example significantly.

Conceptual Approach:

A data puller function should:

- Take minimal, essential inputs (like latitude and longitude).
- Construct the API request.
- Handle basic success/failure.
- Return the fetched data in a usable format.

Python

```
import requests
import json
import pandas as pd

print("\n--- Simplified Lightweight Data Puller Function with CSV Save ---")

def get_simple_weather(latitude: float, longitude: float, location_name: str)
-> pd.DataFrame or None:
    """
    Fetches current weather and saves it as a CSV file.

    Args:
        latitude: Latitude of the location
        longitude: Longitude of the location
        location_name: Name used to save the CSV file

    Returns:
        A DataFrame with weather info or None on error
    """
    api_url = "https://api.open-meteo.com/v1/forecast"
    params = {
        "latitude": latitude,
        "longitude": longitude,
        "current_weather": True,
```

```

        "temperature_unit": "celsius",
        "timezone": "auto"
    }

    print(f"\nPulling weather for {location_name} (Lat: {latitude}, Lon:
{longitude})...")
    try:
        response = requests.get(api_url, params=params, timeout=5)
        response.raise_for_status()

        weather_data = response.json()

        if "current_weather" in weather_data:
            current = weather_data["current_weather"]
            df_weather = pd.DataFrame({
                "Time": [current['time']],
                "Temperature_C": [current['temperature']],
                "WindSpeed_kmh": [current['windspeed']]
            })
            # Save to CSV
            filename = f"{location_name.lower().replace(' ', '_')}_weather.csv"
            df_weather.to_csv(filename, index=False)
            print(f"✅ Weather data for {location_name} saved to '{filename}'")
            return df_weather
        else:
            print("⚠️ 'current_weather' not found in API response.")
            return None

    except requests.exceptions.RequestException as e:
        print(f"❌ Request failed: {e}")
        return None
    except json.JSONDecodeError:
        print(f"❌ Invalid JSON received from API.")
        return None

# Bengaluru
bengaluru_weather = get_simple_weather(12.9716, 77.5946, "Bengaluru")
if bengaluru_weather is not None:
    print("\nBengaluru Weather:\n", bengaluru_weather)

# Sriharikota
sriharikota_weather = get_simple_weather(13.6589, 80.2297, "Sriharikota")

```



```
if sriharikota_weather is not None:  
    print("\nSriharikota Weather:\n", sriharikota_weather)
```

4. Building Your Own APIs with Flask

So far, we've learned how to *use* APIs. Now, let's learn how to *create* very simple ones! Flask is a popular "micro" web framework in Python, meaning it's lightweight and gives you a lot of flexibility. It's great for building simple web services and APIs.

Installation: Open your terminal/command prompt and run:

```
Python  
pip install -U flask --no-cache
```

4.1 Basic Flask API: "Hello World" & Simple GET/POST

Conceptual Approach:

You create a Flask app object, and then use `@app.route()` to define what happens when someone visits a specific URL.

How to Run (Important!):

The Flask code below needs to be saved as a .py file (e.g., `flask_app.py`) and run from your terminal. You cannot run Flask directly within a Jupyter notebook cell as a live server.

File: flask_app.py

Python

```
# Save this file as flask_app.py
from flask import Flask, request, jsonify # Flask essentials

app = Flask(__name__) # Create the Flask app instance

# --- ROUTES ---

# Root Route: Simple hello message
# Access: http://127.0.0.1:5000/
@app.route('/')
def home():
    return "Hello from Flask API! Try /hello or /echo_name?name=Alice"

# Simple GET route returning JSON
# Access: http://127.0.0.1:5000/hello
@app.route('/hello')
def hello_world():
    return jsonify(message="Hello, Engineers!")

# Route with query parameter (GET)
# Access: http://127.0.0.1:5000/echo_name?name=Alice
@app.route('/echo_name')
def echo_name():
    name = request.args.get('name', 'Guest')
    return jsonify(greeting=f"Hello, {name}!")

# Route for POST request with JSON payload
# Access: http://127.0.0.1:5000/sensor_data
@app.route('/sensor_data', methods=['POST'])
def receive_sensor_data():
    if request.is_json:
        sensor_data = request.json
        sensor_id = sensor_data.get('id', 'N/A')
        value = sensor_data.get('value', 'N/A')
        print(f"Server received sensor data: ID={sensor_id}, Value={value}")
        return jsonify(status="success", received_data=sensor_data), 200
    else:
        return jsonify(status="error", message="Request must be JSON"), 400

# --- RUNNING THE APP ---
```

```
if __name__ == '__main__':  
    app.run(debug=True)
```

Shell

```
# --- GET request to home route ---  
curl -X GET http://127.0.0.1:5000/  
  
# --- GET request to /hello ---  
curl -X GET http://127.0.0.1:5000/hello  
  
# --- GET request with query parameter to /echo_name ---  
curl -X GET "http://127.0.0.1:5000/echo_name?name=Rajath"  
  
# --- POST request with JSON data to /sensor_data ---  
curl -X POST http://127.0.0.1:5000/sensor_data \  
    -H "Content-Type: application/json" \  
    -d '{"id": "TEMP_001", "value": 28.7}'
```

Bash Commands to Run Flask (flask_app.py):

1. Save the code above as flask_app.py.
2. Open your terminal or Anaconda Prompt.
3. Navigate to the directory where you saved flask_app.py.
4. Run the Flask app using:

Python

```
# Step 1: Install Flask (if not already installed)  
pip install flask  
  
# Step 2: Run the Flask app  
python flask_app.py
```

You will see output like * Running on <http://127.0.0.1:5000/> (Press CTRL+C to quit).

4.2 Flask API: Serving an HTML Form and Processing Input

This example shows how Flask can display a simple web page with a form, take input from that form, and process it on the server.

Conceptual Approach:

- Create a route for a GET request that shows the HTML form (`render_template_string`).
- Create a route for a POST request that receives and processes the form data, which Flask makes available in `request.form`.

File: `flask_form_app.py`

Python

```
# flask_form_app.py
# A simple Flask app to accept and display sensor data via a styled HTML form

from flask import Flask, request, render_template_string

# Create an instance of the Flask application
app = Flask(__name__)

# ----- EMBEDDED HTML FORM TEMPLATE -----
# This HTML is styled using Tailwind CSS and rendered directly from a string.
# In real-world Flask projects, you would place HTML in a separate .html file
# inside a "templates" folder.

HTML_FORM = """
<!DOCTYPE html>
<html>
<head>
    <title>Process Sensor Data</title>
    <!-- Load Tailwind CSS for fast, modern styling -->
    <link
href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css"
rel="stylesheet">
</head>

```

```

<body class="bg-gray-100 flex items-center justify-center min-h-screen">
  <div class="bg-white p-8 rounded-lg shadow-md w-full max-w-sm">
    <h2 class="text-2xl font-bold mb-6 text-gray-800 text-center">Sensor
    Data Input (Flask)</h2>

    <!-- Form to accept sensor data. It uses POST method to send data
    securely -->
    <form action="/process_sensor_form" method="POST">

      <!-- Sensor ID Input Field -->
      <div class="mb-4">
        <label for="sensor_id" class="block text-gray-700 text-sm
        font-bold mb-2">Sensor ID:</label>
        <input type="text" id="sensor_id" name="sensor_id"
        class="shadow appearance-none border rounded w-full py-2
        px-3 text-gray-700"
        required>
      </div>

      <!-- Temperature Input Field -->
      <div class="mb-4">
        <label for="temperature" class="block text-gray-700 text-sm
        font-bold mb-2">Temperature (°C):</label>
        <input type="number" step="0.1" id="temperature"
        name="temperature"
        class="shadow appearance-none border rounded w-full py-2
        px-3 text-gray-700"
        required>
      </div>

      <!-- Pressure Input Field (Optional) -->
      <div class="mb-6">
        <label for="pressure" class="block text-gray-700 text-sm
        font-bold mb-2">Pressure (kPa):</label>
        <input type="number" step="0.1" id="pressure" name="pressure"
        class="shadow appearance-none border rounded w-full py-2
        px-3 text-gray-700">
      </div>

      <!-- Submit Button -->
      <div class="flex items-center justify-between">
        <button type="submit"
        class="bg-blue-500 hover:bg-blue-700 text-white

```

```

font-bold py-2 px-4 rounded w-full">
    Submit Data
</button>
</div>
</form>

<!-- Section to display success message after form is submitted -->
{% if result %}
<div class="mt-6 p-4 bg-green-100 border border-green-400
text-green-700 rounded relative">
    <strong class="font-bold">Success!</strong>
    <span class="block sm:inline">Data Received: {{ result }}</span>
</div>
{% endif %}
</div>
</body>
</html>
"""

# ----- FLASK ROUTES -----

@app.route('/input_form')
def show_input_form():
    """
    Route to display the HTML form.
    This function is triggered when user visits:
    http://127.0.0.1:5001/input_form
    """
    return render_template_string(HTML_FORM, result=None)

@app.route('/process_sensor_form', methods=['POST'])
def process_sensor_form():
    """
    Route to handle form submission (POST).
    Extracts sensor_id, temperature, and pressure values from form,
    performs basic validation and returns the same form with a result message.
    """
    # Get values from form fields (submitted via POST)
    sensor_id = request.form['sensor_id']
    temperature = request.form['temperature']
    pressure = request.form.get('pressure', 'N/A') # Optional field with

```

default value

```
try:
    # Convert temperature and pressure to float
    temp_float = float(temperature)
    pressure_float = float(pressure) if pressure != 'N/A' else None

    # Build a formatted string with the submitted values
    processed_message = (
        f"Sensor ID: {sensor_id}, Temp: {temp_float}°C, "
        f"Pressure: {pressure_float if pressure_float is not None else
'N/A'} kPa"
    )

    # Print message to the terminal (good for debugging or logging)
    print(f"Server received and processed: {processed_message}")

    # Render the same form again with the result displayed
    return render_template_string(HTML_FORM, result=processed_message)

except ValueError:
    # If conversion fails (e.g. non-numeric input), show error message
    return render_template_string(HTML_FORM, result="Error: Temperature and
Pressure must be valid numbers.")

# ----- MAIN APPLICATION ENTRY -----

if __name__ == '__main__':
    # Run the Flask app locally on port 5001 with debug mode enabled
    app.run(debug=True, port=5001)
```

Bash Commands to Run Flask Form App (flask_form_app.py):

1. Save the code above as flask_form_app.py.
2. Open your terminal or Anaconda Prompt.
3. Navigate to the directory where you saved flask_form_app.py.
4. Run the Flask app using:

Shell

```
python flask_form_app.py
```

You will see output indicating it's running on <http://127.0.0.1:5001/>

Open your web browser and visit <http://127.0.0.1:5001/> . Fill the form and submit.

5. Building Your Own APIs with FastAPI

FastAPI is a modern, fast (high performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. It's known for its speed, automatic data validation, and built-in interactive API documentation.

Installation: Open your terminal/command prompt and run:

Shell

```
pip install fastapi uvicorn (uvicorn is the server that runs FastAPI apps)
```

5.1 Basic FastAPI API: "Hello World" & Simple GET/POST

Conceptual Approach:

Similar to Flask, you create a FastAPI app object. Routes are defined using decorators like `@app.get()`, `@app.post()`. FastAPI leverages Python type hints to automatically validate incoming data and generate documentation.

How to Run (Important!):

FastAPI code needs to be saved as a `.py` file (e.g., `fastapi_app.py`) and run using `uvicorn` from your terminal. You cannot run FastAPI directly within a Jupyter notebook cell as a live server.

File: fastapi_app.py

Python

```
# Save this code as fastapi_app.py

from fastapi import FastAPI                                # FastAPI is a high-performance web
framework for building APIs                                # Used to define and validate data
from pydantic import BaseModel                              # Lets us mark fields as optional
models for input                                           # Lets us mark fields as optional
from typing import Optional                                # Lets us mark fields as optional
(e.g., pressure can be None)

# Create a FastAPI application instance
app = FastAPI()

# -----
# Define a data model (schema) for incoming POST requests using Pydantic
# This is how you tell FastAPI what kind of data to expect
# -----
class SensorData(BaseModel):
    sensor_id: str                                           # Required: Sensor ID (e.g.,
"TEMP_001")
    temperature: float                                       # Required: Temperature reading
    pressure: Optional[float] = None                        # Optional: Pressure value, can be
left out

# -----
# ROUTES / ENDPOINTS
# -----

# Basic root endpoint to verify API is running
# Access: http://127.0.0.1:8000/
@app.get("/")
async def read_root():
    return {"message": "Hello from FastAPI!"}

# Health check or status endpoint
# Access: http://127.0.0.1:8000/status
@app.get("/status")
async def get_status():
    return {"system_status": "Operational", "uptime_hours": 125.5}
```

```

# Endpoint to fetch details of a sensor by its ID (path parameter)
# Access: http://127.0.0.1:8000/sensor/TEMP_001
@app.get("/sensor/{sensor_id}")
async def get_sensor_info(sensor_id: str):
    """
    Returns mock metadata for a sensor ID.
    If 'TEMP_001' is requested, it returns hardcoded sensor details.
    """
    if sensor_id == "TEMP_001":
        return {
            "sensor_id": sensor_id,
            "type": "Temperature",
            "location": "Engine Bay"
        }
    else:
        # Return a 404 status with a custom message
        return {"sensor_id": sensor_id, "message": "Sensor not found"}, 404


# Endpoint to accept new sensor data using a POST request
# Access: http://127.0.0.1:8000/new_sensor_reading/
@app.post("/new_sensor_reading/")
async def create_sensor_reading(data: SensorData):
    """
    Accepts and parses JSON data into a SensorData object automatically.
    FastAPI validates the data format and types.
    """
    # You could store this data into a database or log it
    print(f"Received new reading: ID={data.sensor_id}, Temp={data.temperature}, Pres={data.pressure}")

    return {
        "status": "Reading received successfully",
        "data_received": data
    }


# -----
# This block is needed only if you're running this file directly with Python
# Normally, you run FastAPI using: `uvicorn fastapi_app:app --reload`
# -----
if __name__ == '__main__':

```

```
import uvicorn
uvicorn.run(app, host="0.0.0.0", port=8000)
```

Bash Commands to Run FastAPI (fastapi_app.py):

1. Save the code above as fastapi_app.py.
2. Open your terminal or Anaconda Prompt.
3. Navigate to the directory where you saved fastapi_app.py.
4. Run the FastAPI app using uvicorn:
uvicorn fastapi_app:app --reload --port 8000
 - o fastapi_app: is the name of your Python file.
 - o app: is the variable inside that file that holds your FastAPI() instance.
 - o --reload: (Optional) Restarts the server automatically when you save changes to your code.
 - o --port 8000: (Optional) Specifies the port.

You will see output like Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit).

cURL Commands to Test fastapi_app.py (Run in a NEW terminal window while FastAPI app is running):

- GET /status:
curl http://127.0.0.1:8000/status

Expected output: {"system_status":"Operational","uptime_hours":125.5}

- GET /sensor/TEMP_001:
curl http://127.0.0.1:8000/sensor/TEMP_001

Expected output:

{"sensor_id":"TEMP_001","type":"Temperature","location":"Engine Bay"}

- GET /sensor/UNKNOWN_SENSOR:

```
curl http://127.0.0.1:8000/sensor/UNKNOWN_SENSOR
```

Expected output: {"sensor_id":"UNKNOWN_SENSOR","message":"Sensor not found"} (with HTTP status 404)

- POST /new_sensor_reading/ (sending JSON with all fields):

```
curl -X POST -H "Content-Type: application/json" -d '{"sensor_id": "Pres002",
"temperature": 99.8, "pressure": 101.2}'
http://127.0.0.1:8000/new_sensor_reading/
```

Expected output: {"status":"Reading received successfully","data_received":{"sensor_id":"Pres002","temperature":99.8,"pressure":101.2}}

(You should also see "Received new reading..." in your FastAPI app's terminal)

- POST /new_sensor_reading/ (sending JSON with optional field missing):

```
curl -X POST -H "Content-Type: application/json" -d '{"sensor_id": "Flow003",
"temperature": 15.0}' http://127.0.0.1:8000/new_sensor_reading/
```

Expected output: {"status":"Reading received successfully","data_received":{"sensor_id":"Flow003","temperature":15.0,"pressure":null}}

- POST /new_sensor_reading/ (sending INVALID JSON - e.g., missing required field):

```
curl -X POST -H "Content-Type: application/json" -d '{"sensor_id": "Invalid"}'
http://127.0.0.1:8000/new_sensor_reading/
```

Expected output (FastAPI will return a validation error automatically with status 422):

```
{"detail":[{"loc":["body","temperature"],"msg":"field
required","type":"value_error.missing"]]}
```

5.3 FastAPI: Automatic API Documentation

One of FastAPI's standout features is its automatic generation of interactive API documentation based on your code and type hints.

- **Swagger UI:** Visit /docs (e.g., <http://127.0.0.1:8000/docs>)
- **ReDoc:** Visit /redoc (e.g., <http://127.0.0.1:8000/redoc>)

These interfaces allow you to explore your API endpoints, see their expected parameters, try out requests directly in the browser, and understand the data models. This is incredibly useful for developers consuming your API.

Conceptual Preview (what you'd see in your browser at /docs):

This is a textual representation of what the Swagger UI documentation would look like:

GET /

Returns a welcome message from the API.

GET /status

Returns the current system operational status.

GET /sensor/{sensor_id}

sensor_id: string (path parameter)

Returns information about a specific sensor.

POST /new_sensor_reading/

Request body:

SensorData (model definition: sensor_id: string, temperature: float, pressure: float or null)

Receives and processes a new sensor reading.

Example Value:

{

"sensor_id": "string",

"temperature": 0,

"pressure": 0

}

5.4 FastAPI API: Serving an HTML Form and Processing Input

FastAPI can also serve static HTML files (like the form HTML) and process traditional HTML form data.

Conceptual Approach:

- You need a templates folder for your HTML files and Jinja2Templates to render them.
- Use `@app.get(..., response_class=HTMLResponse)` for routes that return HTML.
- For handling form data (POST requests from HTML forms), use `Form()` as a dependency for each form field in your function's parameters.

Setup for `fastapi_form_app.py`:

1. Create a folder named `templates` in the same directory as your Python file.
2. Save the HTML content from `HTML_FORM_FILE_CONTENT` below into a file named `sensor_form.html` *inside the templates folder*.
3. Save the Python code below as `fastapi_form_app.py` in the main directory.

File: `fastapi_form_app.py`

Python

Save this code as `fastapi_form_app.py`

```
from fastapi import FastAPI, Request, Form                # Import FastAPI
core and tools for form handling                           # Used to indicate
from fastapi.responses import HTMLResponse                # For rendering
HTML output                                                 # To mark pressure
from fastapi.templating import Jinja2Templates
Jinja2 HTML templates
from typing import Optional
as optional

# Create the FastAPI app instance
app = FastAPI()

# Tell FastAPI where to find your HTML templates (create a folder named
'templates')
```

```

templates = Jinja2Templates(directory="templates")

# =====
# ROUTE 1: Display HTML form
# =====
# Access this route at: http://127.0.0.1:8001/sensor_form
@app.get("/sensor_form", response_class=HTMLResponse)
async def get_sensor_form(request: Request, result: Optional[str] = None):
    """
    Display the sensor input form. The `result` (if any) is passed to the
    template to show a message after form submission.
    """
    return templates.TemplateResponse("sensor_form.html", {
        "request": request,
        "result": result
    })

# =====
# ROUTE 2: Process submitted form
# =====
# This route is hit when the user clicks "Submit" on the form
@app.post("/process_form", response_class=HTMLResponse)
async def post_sensor_form(
    request: Request,                                # Required to re-render
    the form with result                               # Required text input
    sensor_id: str = Form(...),                        # Required number input,
    temperature: float = Form(...),                    # Required number input,
    auto-converted to float                             # Optional number input,
    pressure: Optional[float] = Form(None)              # Optional number input,
    will be None if left blank
):
    # Process received data
    processed_message = (
        f"Sensor ID: {sensor_id}, Temp: {temperature}°C, "
        f"Pressure: {pressure if pressure is not None else 'N/A'} kPa"
    )

    # Print for server logs
    print("✅ Received:", processed_message)

    # Re-render the form page with result
    return templates.TemplateResponse("sensor_form.html", {

```

```

        "request": request,
        "result": processed_message
    })

# =====
# HOW TO RUN THIS APP:
# =====
# Run the server with:
#   uvicorn fastapi_form_app:app --reload --port 8001
#
# Make sure your folder has this structure:
#   fastapi_form_app.py
#   templates/
#       └─ sensor_form.html
#
# The HTML file content is provided below.

```

HTML

```

<!DOCTYPE html>
<html>
<head>
    <title>Process Sensor Data (FastAPI)</title>
    <link
href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css"
rel="stylesheet">
</head>
<body class="bg-gray-100 flex items-center justify-center min-h-screen">
    <div class="bg-white p-8 rounded-lg shadow-md w-full max-w-sm">
        <h2 class="text-2xl font-bold mb-6 text-gray-800 text-center">FastAPI
Sensor Data Input</h2>
        <form action="/process_form" method="POST">
            <div class="mb-4">
                <label for="sensor_id" class="block text-gray-700 text-sm
font-bold mb-2">Sensor ID:</label>
                <input type="text" id="sensor_id" name="sensor_id" required
class="shadow appearance-none border rounded w-full py-2
px-3 text-gray-700">
            </div>
            <div class="mb-4">

```



```

        <label for="temperature" class="block text-gray-700 text-sm
font-bold mb-2">Temperature (°C):</label>
        <input type="number" step="0.1" id="temperature"
name="temperature" required
        class="shadow appearance-none border rounded w-full py-2
px-3 text-gray-700">
    </div>
    <div class="mb-6">
        <label for="pressure" class="block text-gray-700 text-sm
font-bold mb-2">Pressure (kPa):</label>
        <input type="number" step="0.1" id="pressure" name="pressure"
        class="shadow appearance-none border rounded w-full py-2
px-3 text-gray-700">
    </div>
    <div>
        <button type="submit"
        class="w-full bg-green-500 hover:bg-green-700
text-white font-bold py-2 px-4 rounded">
            Submit Data
        </button>
    </div>
</form>

{% if result %}
    <div class="mt-6 p-4 bg-blue-100 border border-blue-400 text-blue-700
rounded relative">
        <strong class="font-bold">Result:</strong>
        <span class="block">{{ result }}</span>
    </div>
{% endif %}
</div>
</body>
</html>

```

Bash Commands to Run FastAPI Form App (fastapi_form_app.py):

1. **Create a folder** named templates in the same directory as your Python file.
2. **Save the HTML content** from HTML_FORM_FILE_CONTENT above into a file named sensor_form.html *inside the templates folder*.
3. **Save the Python code** above as fastapi_form_app.py in the main directory.

4. Open your **terminal** or **Anaconda Prompt**.
5. Navigate to the directory where you saved `fastapi_form_app.py`.
6. Run the FastAPI app using `uvicorn`:

```
uvicorn fastapi_form_app:app --reload --port 8001
```

You will see output indicating it's running on `http://127.0.0.1:8001/`.

Open your web browser and visit `http://127.0.0.1:8001/sensor_form`. Fill the form and submit.

6. Key Takeaways from Day 5, Session 1

This session has provided you with a comprehensive introduction to API programming in Python, covering both consuming existing APIs and building your own simple web services.




- **API Fundamentals:** Understood what APIs are, why they are crucial for software communication, and their importance in engineering and scientific contexts.
- **REST Principles:** Gained a solid understanding of REST architectural principles (Client-Server, Stateless, Resources/URLs, HTTP Verbs, JSON).
- **Consuming APIs (requests):**
 - Mastered `requests.get()` for retrieving data (with parameters).
 - Learned `requests.post()` for creating new data (sending JSON).
 - Understood how to interpret `response.status_code` and parse JSON responses with `response.json()`.
 - Implemented a **simplified lightweight data puller function** for practical data fetching.
- **Building APIs (Flask):**
 - Introduced Flask as a micro-web framework for quick API development.
 - Created basic GET and POST API endpoints using `@app.route()`.
 - Learned to serve a simple HTML form and process its input using `request.form`.
 - Understood how to run Flask applications from the terminal using `python your_app.py`.
 - Tested Flask APIs using `curl` commands.

- **Building APIs (FastAPI):**

- Introduced FastAPI as a modern, high-performance web framework for robust API development.
- Created GET and POST API endpoints using `@app.get()`, `@app.post()`, and **Pydantic models** for automatic data validation.
- Discovered FastAPI's **automatic interactive API documentation** (`/docs` and `/redoc`), which simplifies API exploration and testing.
- Learned to serve HTML forms and process input using Form dependencies.
- Understood how to run FastAPI applications using uvicorn from the terminal (`uvicorn your_app:app --reload`).
- **Tested FastAPI APIs using curl commands.**

By mastering API programming with Python, you are now empowered to build more connected and data-rich applications, enabling seamless integration with a vast ecosystem of online services and data sources, and even create your own simple web services for your engineering and scientific projects.

ISRO URSC – Python Training | Analog Data | Rajath Kumar

 For queries: rajath@analogdata.ai |  [\(+91\) 96633 53992](tel:+919663353992) |  <https://analogdata.ai>

This material is part of the ISRO URSC Python Training Program conducted by Analog Data (June 2025). For educational use only. © 2025 Analog Data.
