Lecture 22 – Building a Text Processing API with FastAPI

In this lecture, we focus on building and executing a FastAPI application designed for **text processing**. This application demonstrates the flexibility of FastAPI by allowing us to perform string manipulations, validate user input, and return structured responses. The project highlights how custom endpoints are defined, how input data is validated with **Pydantic**, and how FastAPI handles errors effectively while providing automatic documentation.

Key Concepts

1. API Endpoints

Endpoints in FastAPI define the routes through which users can send requests to the server. These endpoints receive input, process it, and return a response. Developers can easily customize endpoints with different parameters and validation rules.

---

2. GET vs POST Methods

- **GET Method**
  Used to retrieve data from the server. In previous examples, GET requests were employed for simple tasks like displaying a welcome message.
- **POST Method**
  Used to send data to the server for processing. In this lecture, the POST method processes user input text, manipulates it according to given parameters, and returns the processed text along with its length.

---

3. Validation with Pydantic

FastAPI integrates with **Pydantic** to handle input validation. By defining input models with Pydantic, we can enforce data types, mark fields as required or optional, and ensure data integrity. If invalid data is received, FastAPI automatically returns descriptive error messages.

Building the Text Processing API

Step 1: API Setup

We begin by importing necessary packages:

- **FastAPI** for creating the API
- **HTTPException** for handling errors
- **Pydantic** for input validation

An application instance is then created with metadata such as title, description, and version.

Step 2: Defining the Text Processing Endpoint

The main functionality of the API is implemented in a **POST endpoint**.

Input Parameters:

- `text` (string, required): The text to be processed.
- `uppercase` (boolean, optional): If `true`, the text is converted to uppercase.

Output Response:

- `processedText`: The final processed string.
- `textLength`: The number of characters in the processed text.

Validation:

- The input string cannot be empty. If it is, an **HTTP 400 Bad Request** error is raised with a clear message.

Step 3: Example Execution

- **Case 1**
  Input: `"Pakistan"`, `uppercase = false`
  Output: `"Pakistan"`, length = 8
- **Case 2**
  Input: `"I love my country"`, `uppercase = true`
  Output: `"I LOVE MY COUNTRY"`, length = 18

Step 4: Response Handling

The API returns structured responses that include both the processed text and its length. If invalid input is provided, appropriate error messages are returned. This ensures reliability and smooth user experience.

Automatic Documentation with Swagger

FastAPI automatically generates documentation for the API, accessible at:

- **Swagger UI**: `http://127.0.0.1:8000/docs`
- **ReDoc**: `http://127.0.0.1:8000/redoc`

The documentation includes:

- API title, description, and version
- Available endpoints with input/output details
- Interactive testing options directly in the browser

This feature reduces the need for manual documentation and accelerates development.

Error Handling

FastAPI provides robust error-handling mechanisms using **HTTPException**.

Examples:

- If the input text is empty → `400 Bad Request` with message: *"Text cannot be empty."*
- If input data types are invalid → descriptive validation errors returned automatically.

This ensures that users receive meaningful feedback when their requests are invalid.

Code Implementation Overview

1. **FastAPI Setup** – Import FastAPI, create an app instance, and configure metadata.
2. **Pydantic Validation** – Define a `BaseModel` to validate the input (`text` and `uppercase`).
3. **API Logic** – The endpoint processes text, applies transformations (uppercase if requested), and calculates length.

4. **Error Handling** – Ensure text is non-empty; raise HTTP errors for invalid input.

Conclusion

In this lecture, we explored the development of a **text processing API** with FastAPI. We saw how to:

- Define custom endpoints with POST requests
- Validate input using **Pydantic**
- Handle errors gracefully with **HTTP exceptions**
- Use FastAPI's **automatic documentation** for testing and exploration

This practice-based approach illustrates the efficiency and flexibility of FastAPI for building APIs. The seamless integration of validation, error handling, and documentation greatly improves development speed and reliability.

Final Thoughts

- FastAPI is ideal for building efficient and scalable APIs.
- Automatic documentation significantly reduces development overhead.
- Pydantic and HTTP exceptions provide strong validation and error-handling tools.

In the next lecture, we will explore **more complex endpoint handling** and move toward **project-based applications** to consolidate these skills.