Developing a RESTful API with FastAPI [Part 1]

> **Objective: In this lecture, we will explore FastAPI and its core functionality. While we won't dive too deeply into advanced features, we will cover the basics of writing a simple REST API, running the application locally, and interacting with it via the localhost. Additionally, we'll introduce basic data validation techniques to ensure robust and reliable APIs.**

What is FastAPI?

FastAPI is a modern, fast (high-performance), and easy-to-use Python web framework for building APIs. It leverages Python type hints to provide automatic data validation, serialization, and documentation. FastAPI is built on **Starlette** for web handling and **Pydantic** for data validation and settings management, ensuring both speed and developer productivity. Documentation

Key Features of FastAPI

1. **High Performance**:
    - FastAPI is built on **ASGI (Asynchronous Server Gateway Interface)**, making it highly performant and suitable for building high-concurrency applications.
    - Performance is comparable to Node.js and Go.
2. **Ease of Use**:
    - The framework is designed to reduce developer effort while maintaining high functionality.
    - Its dependency injection system makes managing complex application logic straightforward.
3. **Automatic Interactive Documentation**:
    - FastAPI automatically generates interactive API documentation with **Swagger UI** and **ReDoc**, allowing developers to test endpoints directly from their browser.
    - Accessible at `/docs` (Swagger) and `/redoc`.
4. **Data Validation and Serialization**:
    - Uses **Pydantic** to enforce data validation and serialization based on Python type hints.

    - Provides clear error messages when data doesn't meet the required structure.
5. **Type Hints and Autocompletion**:
    - Takes full advantage of Python type annotations to improve code clarity, reduce bugs, and enable better IDE support for autocompletion.
6. **Asynchronous Support**:
    - Supports asynchronous programming with `async def` for building non-blocking, high-performance APIs.
7. **Built-In Dependency Injection**:
    - Simplifies handling dependencies like database connections, authentication, and services using its powerful dependency injection system.

How Does FastAPI Work?

FastAPI is based on Python's **type hints** to define request and response models. These type hints allow the framework to:

1. Validate request data (e.g., query parameters, JSON payloads).
2. Automatically serialize responses.
3. Generate OpenAPI-compliant documentation.

Use Cases for FastAPI

1. **RESTful APIs**:
    - Build APIs for CRUD operations, integration with frontend applications, or backend services.
2. **Microservices**:
    - Ideal for creating lightweight and fast microservices with high concurrency support.
3. **Real-Time Applications**:
    - Use WebSocket support for real-time features like chat apps or live dashboards.
4. **Machine Learning Model Deployment**:
    - Serve machine learning models with APIs to make predictions.
5. **IoT and Streaming Data**:
    - Handle real-time data streams for IoT applications or data pipelines.

Comparison: FastAPI vs Other Frameworks

Feature	FastAPI	Flask	Django REST Framework
Performance	Very High	Moderate	Moderate
Type Safety	Yes (Pydantic)	No	Partial
Automatic Docs	Yes (Swagger)	No	Partial
Ease of Use	High	High	Moderate
Asynchronous Support	Built-in	No (Requires extensions)	No (Partial)
Best Use Case	APIs and Microservices	Small web apps	Large-scale APIs with integrated web apps

Why Use FastAPI?

FastAPI is an excellent choice if you want:

1. **High performance** for real-time applications.
2. **Easy-to-use features** for building APIs quickly.
3. **Automatic validation and documentation** for robust and reliable APIs.
4. **Scalability** to handle high concurrency workloads.

Installing FastAPI

```
!pip install fastapi uvicorn
```

Creating your first Endpoint

```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def hello_world():
  return {"hello": "world"}
```

In this example, we create a **GET endpoint** at the root path (`/`) that always responds with the message: `{"hello": "world"}`.

To make this work, we first create a **FastAPI app** by creating an object called `app`. This object is like the "brain" of the application, connecting all the API routes.

Next, we define a function that runs when someone accesses our endpoint. This function is called the **path operation function**. FastAPI automatically takes what this function returns (in this case, `{"hello": "world"}`) and sends it back as a proper **JSON response**.

The special part of the code is the line starting with `@`. This is called a **decorator**. It tells FastAPI to connect the function below it with a specific URL path (like `/`) and an HTTP method (like GET). Think of it as a shortcut for telling FastAPI where this function should run.

For example, this line:

```
@app.get("/")
```

Meaning: "When someone sends a GET request to `/`, run the function below this line."

FastAPI has different decorators for different HTTP methods, like `@app.post` for POST requests or `@app.put` for PUT requests.

Finally, when you run this code, FastAPI starts a local server where you can test your API!

Important Things to Keep In Mind

1. The File Name

When you write your FastAPI application, it is saved as a Python file. For example, if your code is saved in a file called `main.py`, this file name is important when running the application.

2. The App Name

In your FastAPI code, you create an instance of the `FastAPI` class, usually named `app`. This instance represents the entire application and is where you define your API routes (like `@app.get("/")`).

3. Running Uvicorn

Uvicorn is a tool that runs your FastAPI app and makes it accessible as a web server. To run the app, you need to provide:

- The **file name** where your FastAPI code is stored.
- The **app instance name** you created.
- Make sure to cd into direction where the .py file is at.
- For example, if your file is named `main.py` and your app instance is called `app`, you would run:

  ```
  uvicorn main:app --reload
  ```

This command means:

- `main`: The file name (without the `.py` extension).
- `app`: The name of the FastAPI app instance in the file.

Why Is This Important?

- **File Name and App Name Must Match**: Uvicorn uses the file name and app name to locate the FastAPI application. If your file is named `myapp.py` and the app instance is called `api`, you would need to run:
  uvicorn myapp:api --reload
- **Flexibility**: You can name your file and app instance anything, as long as you correctly reference them in the `uvicorn` command.

What Does `--reload` Do?

The `--reload` option automatically restarts the server whenever you make changes to your code. It’s useful during development, so you don’t have to manually stop and restart the server every time you edit the file.

Example

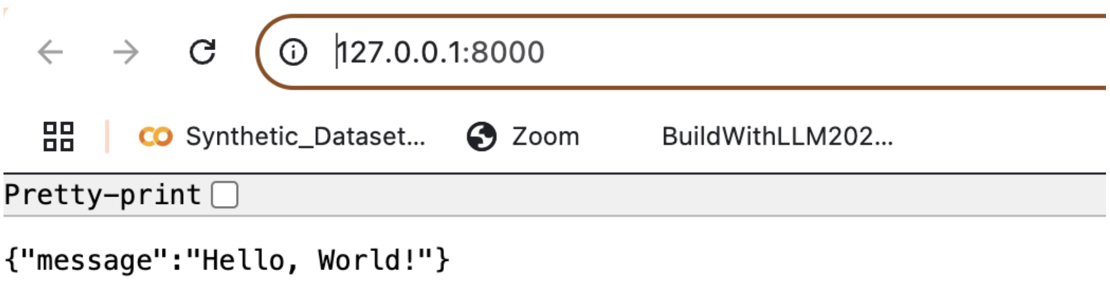
- If you save the following FastAPI code in a file named `server.py`:

  ```python
  from fastapi import FastAPI

  app = FastAPI()

  @app.get("/")
  def read_root():
      return {"message": "Hello, World!"}
  ```
- You would run it with:

  ```
  uvicorn server:app --reload
  ```
  - `server`: The file name (without `.py`).
  - `app`: The name of the FastAPI app instance.

  When you visit `http://127.0.0.1:8000` in your browser, you’ll see the response:

  ```
  {"message": "Hello, World!"}
  ```



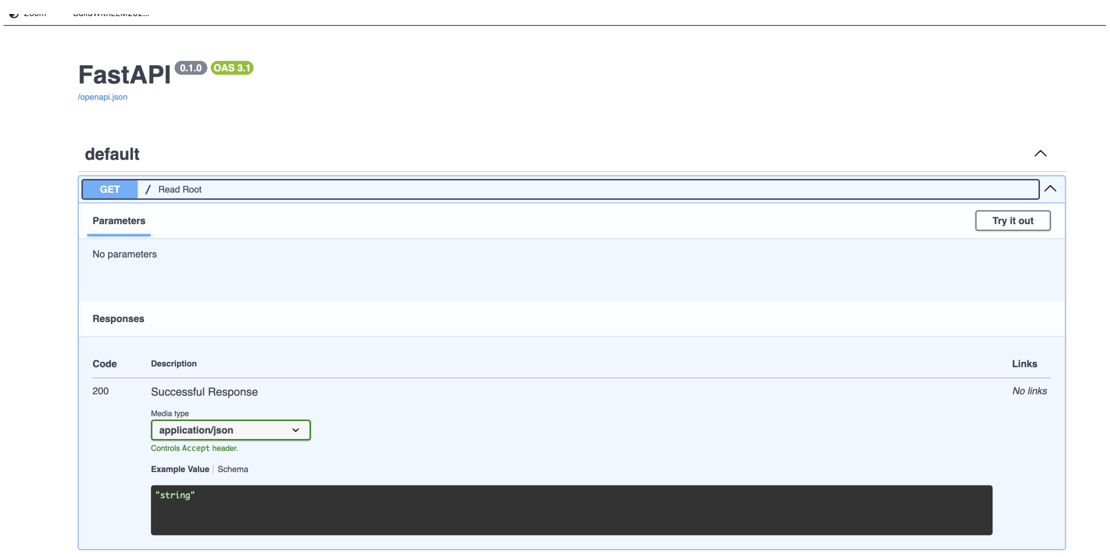
```
{"message":"Hello, World!"}
```

Automatic Documentation

One of the most beloved features of FastAPI is the automatic interactive documentation. When you run the FastAPI app, two interactive documentation pages are automatically available:

- **Swagger UI**: Accessible at `/docs`.
- **ReDoc**: Accessible at `/redoc`.

These provide a clean interface to test your API endpoints and view their specifications.



No parameters

200 Successful Response No links

application/json

Controls Accept header.

```
"string"
```

Handling Request Parameters in FastAPI

A REST API is like a menu in a restaurant—it gives you a structured way to ask for what you want. APIs let you interact with data by sending information like:

- **Path Parameters**: Specific parts of the URL that provide dynamic information (e.g., `/users/123` where `123` is the ID).
- **Query Parameters**: Additional details sent in the URL after a `?` (e.g., `/search?name=John`).
- **Headers**: Additional metadata about the request.

In many frameworks, you'd need to manually extract and validate these inputs. However, **FastAPI** makes this easier by using **type hints** to automatically:

1. Extract parameters from the request.
2. Validate the data based on the expected type.

This feature not only saves time but also ensures your API is more reliable and easier to debug.

Path Parameters

Path parameters are part of the URL itself and are commonly used for dynamic information, such as the ID of a specific user you want to retrieve.

- For example:
  GET /users/123

Here, `123` is the **path parameter** representing the user's ID. Let's see how you can define this in FastAPI.

Example: Handling Path Parameters

path_parameters.py:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, World!"}

@app.get("/users/{id}")
def get_user(id: int):
    return {"id": id}
```

How It Works:

1. The `@app.get("/users/{id}")` line defines an endpoint that expects a **path parameter** (`id`) at the end of the URL.
2. In the `get_user` function, the `id` parameter is automatically retrieved from the URL when a request is made.

3. The `int` type hint ensures that FastAPI:
    - Validates that the `id` is an integer.
    - Returns a clear error if it’s not.

Testing the Path Parameter

Let's run the script via uvicorn first. Then test the following scenarios:

Scenario 1: Missing Path Parameter

If you omit the parameter in the request:

http http://localhost:8000/users
Response:

```
{
  "detail": "Not Found"
}
```

- The API returns a `404 Not Found` error because the URL doesn’t match the expected pattern (`/users/{id}`).

**Scenario 2: Providing a Valid

script_name: path_parameter.py

If you send a request with a valid integer:

http://localhost:8000/users/123
Response:

```
{
  "id": 123
}
```

- The API correctly extracts the `id` value (`123`) from the URL and returns it in the response.

Why Path Parameters Are Useful

Path parameters are great for dynamic endpoints, such as:

- Retrieving a specific resource (e.g., `/products/45` to get product `45`).
- Performing actions on a specific resource (e.g., `/orders/99/cancel` to cancel order `99`).

Hands-On Exercise

Create a script called exercise_1.py containing your solutions and push to the "fastapi-project" folder in you git branch.

1. **Define a Path Parameter for a Product**:
    - Create an endpoint `/products/{product_name}` that accepts a product name as a string and returns it in a JSON response.

    Expected Response:

    ```
    {
        "product_id": "Juice"
    }
    ```
2. **Test Error Handling**:
    - Send a request to `/products` without a `product_name` and observe the error response.
3. **Try Invalid Data**:
    - Send a request to `/products/33` (integer) and see how FastAPI handles it.

Limiting Allowed Values and Advanced Validation in FastAPI

When building APIs, you often want to control what kind of input your users can send. For example:

- A user’s role might need to be limited to "admin" or "standard."
- A numeric value (like an Age) might need to be greater than or equal to 1.
- A string might need to follow a specific format, like a license plate.

With **FastAPI**, you can enforce these rules easily using tools like:

1. **Enumerations (`Enum`)** for predefined values.
2. **Validation functions like `Path`** for numbers and strings.

Using Enumerations to Limit Values

What Are Enumerations (`Enum`)?

An **enumeration** is a way to define a set of allowed values for a parameter. For example, if you only want to allow users with roles `"admin"` or `"standard"`, you can use an `Enum`.

Example: Limiting User Roles

script_name: validation1.py

```python
from enum import Enum
from fastapi import FastAPI

# Define an Enum for user types
class UserType(str, Enum):
    STANDARD = "standard"
    ADMIN = "admin"

# Create the FastAPI app
app = FastAPI()

# Use the Enum as a path parameter type
@app.get("/users/{type}/{id}")
def get_user(type: UserType, id: int):
    return {"type": type, "id": id}
```

How It Works:

1. **Defining the Enum**:
   - `UserType` is an `Enum` that lists the allowed values: `"standard"` and `"admin"`.
   - It inherits from both `str` (so the values behave like strings) and `Enum` (to define the allowed values).
2. **Using the Enum in a Route**:
   - The `type` parameter in `/users/{type}/{id}` must match one of the Enum values.
   - If it doesn't, FastAPI returns an error.

Testing the Endpoint:

1. **Valid Request**:

   ```
   http://localhost:8000/users/admin/123
   ```
   Response:

   ```json
   {
       "type": "admin",
       "id": 123
   }
   ```
2. **Invalid Request**:

   ```
   http://localhost:8000/users/hello/123
   ```
   Response:

   ```json
   {
       "detail": [
           {
               "msg": "value is not a valid enumeration member;
   permitted: 'standard', 'admin'",
               "type": "type_error.enum"
   ```

```
        }
    ]
}
```

Validating Numbers with `Path`

Sometimes, you need to ensure a number meets specific conditions, such as being positive. FastAPI provides the `Path` function for advanced validation.

Example: Restricting an ID to Positive Numbers

script_name:validation2.py

```python
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/users/{id}")
def get_user(id: int = Path(..., ge=1)):
    return {"id": id}
```

How It Works:

1. The `Path` function specifies validation rules for the `id` parameter:
   - `...` means the parameter is required (no default value).
   - `ge=1` ensures the value is **greater than or equal to 1.**
2. If the `id` doesn't meet the condition, FastAPI returns an error.

Testing the Endpoint:

1. **Valid Request**:

   http://localhost:8000/users/10
   Response:

   ```
   {
       "id": 10
   }
   ```
2. **Invalid Request**:

   http://localhost:8000/users/0
   Response:

   ```
   {
       "detail": [
           {
               "msg": "ensure this value is greater than or equal
   to 1",
               "type": "value_error.number.not_ge"
   ```

```
            }
        ]
    }
```

Validating Strings

Example: Restricting String Length

You can validate strings based on their length. For example, if you want to ensure a license plate number has exactly 9 characters:

script_name: validation3.py

```python
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/license-plates/{license}")
def get_license_plate(license: str = Path(..., min_length=9,
max_length=9)):
    return {"license": license}
```

1. **`min_length=9`**: The string must be at least 9 characters.
2. **`max_length=9`**: The string must be no more than 9 characters.

Example: Using Regular Expressions

For more precise validation, like matching a specific pattern (e.g., French license plates `AB-123-CD`), use a **regular expression** (`regex`):

script_name: validation4.py

```python
@app.get("/license-plates/{license}")
def get_license_plate(license: str = Path(..., regex=r"^\w{2}-
\d{3}-\w{2}$")):
    return {"license": license}
```

1. **`regex=r"^\w{2}-\d{3}-\w{2}$"`**:
   - `\w{2}`: Two alphanumeric characters.
   - `\d{3}`: Three digits.
   - `-`: Dashes between sections.
2. **The `r` prefix**: Indicates this is a raw string for regex.

Testing the Endpoint:

1. **Valid Request**:

   ```
   $ http http://localhost:8000/license-plates/AB-123-CD
   ```
   Response:

   ```
   {
       "license": "AB-123-CD"
   ```

```
}
```

2. **Invalid Request**:

   ```
   $ http http://localhost:8000/license-plates/INVALID123
   ```

   Response:

   ```
   {
       "detail": [
           {
               "msg": "string does not match regex '^\\w{2}-
   \\d{3}-\\w{2}$'",
               "type": "value_error.str.regex"
           }
       ]
   }
   ```

Validation Options Recap

1. **Number Validations**:
   - `gt`: Greater than.
   - `ge`: Greater than or equal to.
   - `lt`: Less than.
   - `le`: Less than or equal to.
2. **String Validations**:
   - `min_length`: Minimum number of characters.
   - `max_length`: Maximum number of characters.
   - `regex`: Match a specific pattern.

Hands-On Exercise

Create a script called exercise_2.py containing your solutions and push to the "fastapi-project" folder in you git branch.

1. **Limit User Types**:
   - Create an endpoint `/roles/{type}` that accepts a type of user role, limited to `"admin"`, `"standard"`, and `"guest"`.
2. **Restrict Numeric IDs**:
   - Create an endpoint `/products/{id}` that accepts only IDs greater than `100`.
3. **Serial Number Validator**
   - Create an endpoint `/validate-serial/{serial}` that validates product serial numbers. The serial number must follow this pattern:
     `AAA-1234-BB`

   Rules:

   A. The serial number must: - Start with **three uppercase letters** (e.g., `AAA`). - Follow with a hyphen (`-`) and **four digits** (e.g., `1234`). - End with another hyphen (`-`) and **two uppercase letters** (e.g., `BB`).
   B. If the serial number is valid, return a success message.
   C. If the serial number is invalid, return a detailed error message.

   Example Valid Serial Numbers:

   - `ABC-1234-ZY`
   - `XYZ-5678-AA`

   Example Invalid Serial Numbers:

   - `AB-1234-ZY` (only 2 letters at the start)
   - `XYZ-12-AA` (only 2 digits in the middle)
   - `XYZ-1234-Z` (only 1 letter at the end)

4. **Checkout the documentation page via the `/docs` endpoint**

Important Things To Know

Your FastAPI script should always look like this with the `if __name__ ...` section.

Example:

script_name: validation3.py

```
from fastapi import FastAPI, Path

app = FastAPI()

@app.get("/license-plates/{license}")
def get_license_plate(license: str = Path(..., min_length=9,
max_length=9)):
    return {"license": license}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("script_name:app", host="0.0.0.0", port=8000,
reload=True)
```

Breakdown:

1. **`if __name__ == "__main__":`**

   - The `__name__` variable is a special built-in variable in Python.
   - When a Python script is run directly, `__name__` is set to `"__main__"`.

- When the script is imported as a module into another script, `__name__` is set to the name of the script/module.

This conditional ensures that the code block inside it only runs when the script is the main entry point.

2. **`import uvicorn`**

   - Uvicorn is an ASGI server used to serve FastAPI applications. It's required to run the app and handle HTTP requests.

3. **`uvicorn.run(app, host="0.0.0.0", port=8000, reload=True)`**

   - Starts the Uvicorn server and runs the FastAPI application:
     - `"script_name:app"`: The FastAPI file and app instance to run.
     - `host="0.0.0.0"`: Makes the app accessible from any network interface on the host machine (not limited to `localhost`).
     - `port=8000`: Specifies the port on which the app will be accessible.
     - `reload=True`: a feature that allows the server to automatically restart whenever you make changes to the code. This is particularly useful during development, as it eliminates the need to manually stop and restart the server after every code change.

Purpose:

- It allows you to run the application server directly by executing the script with a command like:
  python script_name.py
- If you import the script into another module, the app won't start automatically, allowing you to reuse its code (e.g., routes or utility functions) without unintended behavior.