Azure API Training <> Revanture

Day 8

Creating Endpoints and Route Handler

Endpoints

An endpoint is a URL path (like /users or /login) that your API exposes, it defines where a client can send an HTTP request and what the server will do in response.

Creating First Endpoint

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/")
def myFirstEndPoint():


    return {"message": "This is my first Endpoint"}
```

Different Types of Endpoints

HTTP Method	FastAPI Decorator	Typical Use
GET	@app.get()	Retrieve data
POST	@app.post()	Create data
PUT	@app.put()	Replace existing data
PATCH	@app.patch()	Update part of data
DELETE	@app.delete()	Delete data

Creating different types of Endpoints

We are going to create different types of endpoints.

Get the source code

Grouping the Endpoints

Creating different types of Endpoints

For bigger apps, organize endpoints by feature (e.g., /users, /products, /orders)

Get the source code

Request/Response models using Pydantic

What is a Pydantic Model?

A Pydantic model is a Python class that defines the structure and types of your data — like a “schema.”

Advantages

- FastAPI automatically validates your input (e.g., wrong email format → error).
- You don’t need manual parsing or validation.
- Automatically documented at /docs and /redoc.

Using Request Model

Using for Request

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr
from typing import Optional

app = FastAPI()

class User(BaseModel):
    name: str
    email: EmailStr
    age: Optional[int] = None

@app.post("/user")
def create_user(user: User):
    return user
```

Using Response Model

Using for Response (1/2)

```
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr
from typing import Optional

app = FastAPI()

class User(BaseModel):
   name: str
   email: EmailStr
   age: Optional[int] = None

class UserResponse(BaseModel):
   statusCode: int
   name: str
   email: EmailStr
   age: Optional[int] = None
   message: str
```

Using for Response (2/2)

```
@app.post("/user", response_model=UserResponse)
def create_user(user: User):
    return UserResponse(
        statusCode=200,
        name=user.name,
        email=user.email,
        age=user.age,
        message="User created successfully"
    )
```

Path and Query Parameter

What Are Path and Query Parameters?

When you make a request like

GET /users/101?active=true&limit=5

Type	Example	Description
Path Parameter	/users/101	Part of the URL **path** (used to identify a resource).
Query Parameter	?active=true&limit=5	Part of the URL **after the "?"** (used for filters, options, etc.). Required with default values

Using Path Parameter

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/user/{user_id}")
def get_user(user_id: int):
    return {"message": f"Fetching user with ID {user_id}"}
```

Query Parameters

Used for optional data that modifies a request

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users")
# /users?active=true&limit=10
def list_users(active: bool = True, limit: int = 10):
    return {
        "message": "Listing users",
        "active_only": active,
        "limit": limit
    }
```

Combine Path + Query Parameters

You can use both together in the same endpoint.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{user_id}")
def list_users(user_id: int, active: bool =  True, limit: int = 10):
   return {
       "message": "Listing users",
       "active_only": active,
       "limit": limit,
       "user_id": user_id
   }
```

Request Body validation

What is Request Validation?

When you Request validation means checking that the data a client sends to your API is complete, correct, and in the right format, before processing it.

FastAPI does this automatically using:

- **Pydantic models** for request bodies
- **Type hints** for path & query parameters

Check this example

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/check-username/{username}")
def check_username(username: str):
    if len(username) < 3:
        return "Username must be at least 3 characters long"
    elif len(username) > 20:
        return "Username must be less than 20 characters long"
    else:
        return "Username is valid"
```

Previous example was valid way of validation?

Correct example: Check string length

```
from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

# Define request body schema
class Username(BaseModel):
    name: str = Field(..., min_length=3, max_length=20)

@app.post("/check-username")
def check_username(username: Username):
    return {"message": "User validated successfully!", "details": username}
```

Using @validator

```
from fastapi import FastAPI
from pydantic import BaseModel, validator

app = FastAPI()

class Username(BaseModel):
    name: str

    @validator("name")
    def validate_name(cls, value):
        if len (value) < 3:
            raise ValueError("Username cannot be less than 3 chars")
        elif len(value) > 20:
            raise ValueError("Username cannot be more than 20 chars")
        return value


@app.post("/check-username")
def check_username(username: Username):
    return {"message": "User validated successfully!," "details": username}
```

Connecting to Database

FastAPI <> Database (Simple Method)

Create a virtual environment in python for better developer workflow (Follow the steps)

1) Install following packages
   a) pip install sqlalchemy pymysql
2) Create a file main.py
3) Add the following code (source code)
4) Create users table and add following dummy data (source code)
5) Run the following command
   a) fastapi dev main.py

Project

Project

Create a FastAPI Application

- You can add new users (username, email, password)
- Update user details (id passed via path parameter)
- Read all users, or limited users passed via query parameter
- Apply all necessary validations
- Delete user