Lab1 - Introduction to FastAPI & Development Setup

Ménissa Cherifa-Luron

2024-10-11

Hello 👋, and welcome to your journey into **FastAPI**, a modern, fast (high-performance) web framework for building APIs with Python!

This live coding is designed to guide you through the initial steps of setting up your development environment and creating your first APIs using FastAPI.

Objectives

- Recall basic commands and syntax for setting up a FastAPI environment.
- Comprehend the purpose and function of FastAPI and its components.
- Break down the structure of FastAPI applications to understand their components.
- Evaluating the correctness and efficiency of CRUD operations implemented in the notebook.
- Develop new FastAPI applications or extend existing ones with additional features.
- Creating new endpoints using Pydantic models to handle complex data validation and processing.

Getting Started

To get the most out of this module, ensure you have access to a terminal or command line interface on your computer. You'll also need Python installed. As you work through the exercises, consider the practical applications of each step and how these skills could be applied to real-world projects.

Let's dive in and start building with FastAPI! 🚀

1. Terminal Setup

- **Create and Activate a Virtual Environment**
  - **Virtual Environment Creation:**
    - To create a virtual environment, use the command:

      ```
      python3 -m venv env
      ```

    - **Activate on Linux and macOS:**

      ```
      source env/bin/activate
      ```

    - **Activate on Windows:**

```
env\Scripts\activate
```

2. Install FastAPI and Uvicorn

- **Installation Commands:**
  - Install FastAPI and Uvicorn using pip:

```
pip install fastapi[standard]
pip install uvicorn[standard]
```

  - **Check Installation:**
    - Verify the installation by checking the FastAPI version:

```
fastapi --version
```

  - **Freeze Requirements:**
    - Save the installed packages to a `requirements.txt` file:

```
pip freeze > requirements.txt
```

3. Write and Test Your First FastAPI "Hello, World!" Endpoint

- **Basic FastAPI Application:**

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

4. Run a Local FastAPI Development Server

- **Using Uvicorn:**
  - Run the server with the command:

```
uvicorn app1:app
```

  - Here, `app1` is the name of the file, and `app` is the FastAPI object.

5. Path Parameters in Queries

- **Adding Parameters:**
  - Define endpoints with path parameters:

```python
@app.get("/text/{message}")
async def read_message(message: str):
    return {"message": message}

@app.get("/number/{number}")
async def read_number(number: int):
    return {"number": number}
```

6. Using Enums and Models

- **Enum Example:**

```python
from enum import Enum

# Using Enums with FastAPI
class PeopleName(str, Enum):
    """Enum for family members' names."""
    brother = "Marc"
    sister = "Marie"
    mother = "Josette"

@app.get("/people/{person_name}")
async def get_person(person_name: PeopleName):
    """Get details based on the family member's name."""
    if person_name == PeopleName.brother:
        return {"person_name": person_name, "message": "He's the best brother!"}
    if person_name == PeopleName.sister:
        return {"person_name": person_name, "message": "She's the best sister!"}
    if person_name == PeopleName.mother:
        return {"person_name": person_name, "message": "She's the best mother!"}

    return {"person_name": person_name, "message": "This person is not in our
family!"}
```

7. Pydantic Models for Data Validation

Here are several examples showcasing how to use Pydantic with FastAPI, demonstrating its capabilities for data validation, serialization, and complex data structures.

1. Basic Model Example

This example shows how to define a simple Pydantic model for a user and validate the data.

```python
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

# Define a Pydantic model for a User
```

```python
class User(BaseModel):
    name: str
    email: EmailStr
    age: int

@app.post("/users/")
async def create_user(user: User):
    return {"message": "User created successfully!", "user": user}
```

2. Nested Models

You can define nested Pydantic models to represent more complex data structures.

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

# Define a model for an Address
class Address(BaseModel):
    street: str
    city: str
    state: str
    zip_code: str

# Define a model for a User with an Address
class UserWithAddress(BaseModel):
    name: str
    email: str
    age: int
    address: Address  # Nesting Address model

@app.post("/users-with-address/")
async def create_user_with_address(user: UserWithAddress):
  return {"message": "User with address created successfully!", "user": user}
```

3. Using Default Values

Pydantic allows you to set default values for model fields.

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# Define a model with default values
class Item(BaseModel):
```

```
    name: str
    price: float
    is_available: bool = True  # Default value

@app.post("/items/")
async def create_item(item: Item):
    return {"message": "Item created successfully!", "item": item}
```

4. Validating Data with Constraints

You can add constraints to model fields using Pydantic's built-in validators.

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

app = FastAPI()

# Define a model with constraints
class Product(BaseModel):
    name: constr(min_length=1, max_length=100)  # Name must be 1-100 characters
    price: float
    quantity: int

@app.post("/products/")
async def create_product(product: Product):
    return {"message": "Product created successfully!", "product": product}
```

5. Using Lists and Optional Fields

Pydantic can handle lists of items and optional fields.

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

app = FastAPI()

# Define a model for an Order
class Order(BaseModel):
    item_name: str
    quantity: int
    notes: Optional[str] = None  # Optional field

# Define a model for a Cart
class Cart(BaseModel):
    user_id: int
    items: List[Order]  # List of Order items
```

```python
@app.post("/carts/")
async def create_cart(cart: Cart):
    return {"message": "Cart created successfully!", "cart": cart}
```

6. Complex Data Types

You can use Pydantic to define more complex types, such as dictionaries.

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Dict

app = FastAPI()

# Define a model for a Configuration
class Configuration(BaseModel):
    setting_name: str
    value: str

# Define a model for a System
class System(BaseModel):
    name: str
    configurations: Dict[str, Configuration]  # Dictionary of configurations

@app.post("/systems/")
async def create_system(system: System):
    return {"message": "System created successfully!", "system": system}
```

8. CRUD Operations with FastAPI

- **Manage Cars Database:**

```python
from pydantic import BaseModel
from datetime import datetime

class Car(BaseModel):
    brand: str
    model: str
    date: datetime
    price: float

cars_db = {}

# Get all cars
@app.get("/cars/", response_model=List[Car])
async def get_all_cars():
    return list(cars_db.values())
```

```python
# Get a car by ID
@app.get("/cars/{car_id}")
async def get_car(car_id: int):
    if car_id not in cars_db:
        raise HTTPException(status_code=404, detail="Car not found")
    return cars_db[car_id]

# Add a new car
@app.post("/cars/")
async def add_car(car: Car):
    car_id = len(cars_db) + 1
    cars_db[car_id] = car.dict()
    return {"message": "Car added successfully", "car": car.dict()}

# Update an existing car
@app.put("/cars/{car_id}")
async def update_car_price(car_id: int, car: Car):
    if car_id not in cars_db:
        raise HTTPException(status_code=404, detail="Car not found")
    car.price *= 1.10  # Augment the price by 10%
    cars_db[car_id] = car.dict()
    return {"message": "Car updated successfully with a 10% price increase",
"car": car.dict()}

# Delete a car
@app.delete("/cars/{car_id}")
async def delete_car(car_id: int):
    if car_id not in cars_db:
        raise HTTPException(status_code=404, detail="Car not found")
    del cars_db[car_id]
    return {"message": "Car deleted successfully"}
```

Customizing FastAPI Documentation

1. Customize the API Metadata

Modify the **title**, **description**, and **version** of the API when initializing the FastAPI instance. This helps in presenting important details about the API on the documentation page.

```python
from fastapi import FastAPI

app = FastAPI(
    title=" Recipe and Movie Collection API",  # Custom API title
    description="An API for managing recipes and movie collections. Manage,
retrieve, and share your favorite items!",  # Custom description
    version="1.0.0",  # Version of your API
)
```

2. Add Tags with Descriptions

Use the `tags_metadata` parameter to categorize endpoints and add descriptions. This allows users to easily understand different parts of your API.

```
tags_metadata = [
    {
        "name": "Introduction",
        "description": "Basic introduction endpoints to get started.",
    },
    {
        "name": "Recipe Management",
          "description": "Endpoints for managing recipes including adding,
retrieving, and deleting recipes.",
    },
    {
        "name": "Movie Collection",
        "description": "Endpoints for managing movie collections, including
adding, retrieving, and deleting movies.",
    },
]

app = FastAPI(
    openapi_tags=tags_metadata  # Apply tags metadata to FastAPI instance
)
```

3. Document Each Endpoint

Add detailed docstrings to each endpoint. This enhances the auto-generated documentation and helps API users understand each endpoint's functionality, parameters, and responses.

```
@app.get("/", tags=["Introduction"])
async def index():
    """
    Returns a welcome message to introduce users to the API.

    **Response:**
    - `200`: A welcome message string.
    """
    return {"message": "Welcome to the Recipe and Movie Collection API!"}

@app.post("/recipes/", tags=["Recipe Management"])
async def add_recipe(recipe: Recipe):
    """
    Add a new recipe to the collection.

    **Request Body:**
    - `title`: (string) The title of the recipe.
    - `ingredients`: (list) The ingredients required.
```

```
    - `instructions`: (string) The steps to prepare the recipe.

    **Response:**
    - `200`: Success message and the new recipe.
    """
    # Recipe handling logic here
    pass
```

4. Customizing OpenAPI Schema

You can customize the OpenAPI schema further by adding terms of service, license, or contact information. This is useful for providing more context about your API, especially for enterprise or public APIs.

```
app = FastAPI(
    title=" Recipe and Movie Collection API",
    description="An API for managing recipes and movie collections.",
    version="1.0.0",
    contact={
        "name": "API Support Team",
        "email": "support@example.com",
        "url": "https://example.com/support"
    },
    license_info={
        "name": "MIT License",
        "url": "https://opensource.org/licenses/MIT",
    },
    terms_of_service="https://example.com/terms/"
)
```

Here's the updated code with enhanced documentation customization:

```
import uvicorn
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List

# Description for the API
description = """
Welcome to the Combined Recipe and Movie Collection API!

## Recipe Management
Manage and share your favorite recipes. Users can add, update, delete, and
retrieve recipes, along with their ingredients and instructions.

## Movie Collection
Keep track of your favorite movies. Users can manage their movie collection,
```

```
including details like the title, director, and release year.

Check out documentation below 👇 for more information on each endpoint.
"""

# Tags metadata for the API documentation
tags_metadata = [
    {
        "name": "Introduction Endpoints",
        "description": "Simple endpoints to try out!",
    },
    {
        "name": "Recipe Management",
        "description": "Manage and share your favorite recipes.",
    },
    {
        "name": "Movie Collection",
        "description": "Keep track of your favorite movies.",
    },
]

app = FastAPI(
    title="📚 Recipe and Movie Collection API",
    description=description,
    version="0.1",
    contact={
        "name": "Ményssa Cherifa-Luron",
        "email": "cmenyssa@live.fr",
        "url": "menyssacherifaluron.com",
    },
    license_info={
        "name": "MIT License",
        "url": "https://opensource.org/licenses/MIT"
    },
    openapi_tags=tags_metadata
)

# Recipe Model
class Recipe(BaseModel):
    title: str
    ingredients: List[str]
    instructions: str
    cook_time: int  # in minutes

# Database simulation
recipes_db = {}

# Movie Model
```

```python
class Movie(BaseModel):
    title: str
    director: str
    year: int
    genre: str

# Movie Database
movies_db = {}

@app.get("/", tags=["Introduction Endpoints"])
async def index():
    """
    Simply returns a welcome message!
    """
    message = "Hello world! This `/` is the most simple and default endpoint.
If you want to learn more, check out documentation of the api at `/docs`"
    return message

# Recipe Endpoints

@app.get("/recipes/", response_model=List[Recipe], tags=["Recipe Management"])
async def get_all_recipes():
    """
    Retrieve a list of all recipes.
    """
    return list(recipes_db.values())

@app.get("/recipes/{recipe_id}", tags=["Recipe Management"])
async def get_recipe(recipe_id: int):
    """
    Retrieve details of a specific recipe by ID.
    """
    if recipe_id not in recipes_db:
        raise HTTPException(status_code=404, detail="Recipe not found")
    return recipes_db[recipe_id]

@app.post("/recipes/", tags=["Recipe Management"])
async def add_recipe(recipe: Recipe):
    """
    Add a new recipe.
    """
    recipe_id = len(recipes_db) + 1
    recipes_db[recipe_id] = recipe.dict()
    return {"message": "Recipe added successfully", "recipe": recipe.dict()}

@app.put("/recipes/{recipe_id}", tags=["Recipe Management"])
async def update_recipe(recipe_id: int, recipe: Recipe):
    """
```

```
    Update an existing recipe by ID.
    """
    if recipe_id not in recipes_db:
        raise HTTPException(status_code=404, detail="Recipe not found")
    recipes_db[recipe_id] = recipe.dict()
    return {"message": "Recipe updated successfully", "recipe": recipe.dict()}

@app.delete("/recipes/{recipe_id}", tags=["Recipe Management"])
async def delete_recipe(recipe_id: int):
    """
    Delete a recipe by ID.
    """
    if recipe_id not in recipes_db:
        raise HTTPException(status_code=404, detail="Recipe not found")
    del recipes_db[recipe_id]
    return {"message": "Recipe deleted successfully"}

# Movie Endpoints

@app.get("/movies/", response_model=List[Movie], tags=["Movie Collection"])
async def get_all_movies():
    """
    Retrieve a list of all movies.
    """
    return list(movies_db.values())

@app.get("/movies/{movie_id}", tags=["Movie Collection"])
async def get_movie(movie_id: int):
    """
    Retrieve details of a specific movie by ID.
    """
    if movie_id not in movies_db:
        raise HTTPException(status_code=404, detail="Movie not found")
    return movies_db[movie_id]

@app.post("/movies/", tags=["Movie Collection"])
async def add_movie(movie: Movie):
    """
    Add a new movie.
    """
    movie_id = len(movies_db) + 1
    movies_db[movie_id] = movie.dict()
    return {"message": "Movie added successfully", "movie": movie.dict()}

@app.put("/movies/{movie_id}", tags=["Movie Collection"])
async def update_movie(movie_id: int, movie: Movie):
    """
    Update an existing movie by ID.
```

```
    """
    if movie_id not in movies_db:
        raise HTTPException(status_code=404, detail="Movie not found")
    movies_db[movie_id] = movie.dict()
    return {"message": "Movie updated successfully", "movie": movie.dict()}

@app.delete("/movies/{movie_id}", tags=["Movie Collection"])
async def delete_movie(movie_id: int):
    """
    Delete a movie by ID.
    """
    if movie_id not in movies_db:
        raise HTTPException(status_code=404, detail="Movie not found")
    del movies_db[movie_id]
    return {"message": "Movie deleted successfully"}

if __name__ == "__main__":
    uvicorn.run(app, host="127.0.0.1", port=8000)
```