Study Notes on FastAPI with Python

Table of Contents

---
1. Introduction to FastAPI
- **Definition**: FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.
- **Key Features**:
  - Fast: One of the fastest Python frameworks available (comparable to Node.js and Go).
  - Easy to use: Designed to be easy to learn and use.
  - Based on Python type hints: Enables auto-completion, validation, and interactive documentation.
---
2. Installation and Setup
- **Environment Preparation**:
  - Python Version: Ensure Python 3.6 or above is installed.
- **Installation**:
  ```bash
  pip install fastapi[all]
  ```
- **Running the application**:
  - Use an ASGI server such as `uvicorn`:
  ```bash
  uvicorn main:app --reload
  ```
---
3. Basic Concepts
ASGI and ASGI Servers
- **ASGI (Asynchronous Server Gateway Interface)**:
  - A specification for Python web servers and applications to communicate with each other.
- **ASGI Servers**:
  - Servers such as Uvicorn or Daphne are capable of handling asynchronous connections.
REST vs RPC

- **REST (Representational State Transfer)**:
  - Architectural style that uses standard HTTP methods.
- **RPC (Remote Procedure Call)**:
  - Calls functions through an HTTP protocol.
---
4. Creating a Basic FastAPI Application
Basic Structure
```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def read_root():
    return {"Hello": "World"}
```
Explanation
- **FastAPI Instance**: `app = FastAPI()` creates an instance of the FastAPI application.
- **Endpoint Definition**: `@app.get("/")` defines a GET endpoint for the root path.
---
5. Path Parameters and Query Parameters
Path Parameters
- Used to capture values from the URL.
```python
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```
Query Parameters
- Optional parameters that can alter the response.
```python
@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}
```
---
6. Request Body and Models
Request Body
- Send data in a structured format. Use Pydantic models for validation.
```python
from pydantic import BaseModel
class Item(BaseModel):
    id: int
    name: str
    price: float
@app.post("/items/")
async def create_item(item: Item):
    return item
```
Pydantic Models
- **Definition**: Pydantic is a data validation and settings management using Python type annotations.
- **Features**:
  - Data validation.
  - Serialization of data.

---
7. Dependency Injection
- **Definition**: A technique to define dependencies that are reused across paths.
- **Example**:
```python
from fastapi import Depends
def get_query(param: str = None):
    return param
@app.get("/items/")
async def read_item(query_param: str = Depends(get_query)):
    return {"query": query_param}
```
---
8. Error Handling in FastAPI
- FastAPI provides built-in error handling for common exceptions.
- **Example**:
```python
from fastapi import HTTPException
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id not in data:
        raise HTTPException(status_code=404, detail="Item not found")
    return data[item_id]
```
---
9. Security Features
OAuth2
- A protocol that allows secure authorization from third-party applications.
- FastAPI supports OAuth2 authentication out of the box.
API Key
- Simple way to secure your API by requiring a key.
```python
from fastapi import Security, Depends
api_key: str = "your_api_key"
@app.get("/secure-data/")
async def read_secure_data(api_key: str = Depends(get_api_key)):
    return {"data": "Secure Data"}
```
---
10. Middleware
- **Definition**: Middleware is a function that is executed for every request before reaching the request path.
- **Example**:
```python
from fastapi import FastAPI
app = FastAPI()
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    response = await call_next(request)
    response.headers["X-Process-Time"] = str(process_time)
    return response
```
---

11. Background Tasks
- Useful for non-blocking tasks.
- **Example**:
```python
from fastapi import BackgroundTasks
def write_log(message: str):
    with open("log.txt", mode="a") as log:
        log.write(message)
@app.post("/send-notification/")
async def send_notification(background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, "Notification sent")
    return {"message": "Notification sent"}
```
---
12. Testing FastAPI Applications
- FastAPI is designed to be easy to test.
- Testing using `pytest`:
```python
from fastapi.testclient import TestClient
client = TestClient(app)
def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"Hello": "World"}
```
---
13. FastAPI and Databases
SQLAlchemy with FastAPI
- SQLAlchemy is a SQL toolkit for Python.
- Basic setup:
```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```
---
14. Summary
- FastAPI is a high-performance, easy-to-use framework for building Python APIs.
- Key components include path and query parameters, request bodies, dependency injection, error handling, and security features.
- The framework leverages Python's type hints for auto-generation of interactive documentation and validation.
- FastAPI integrates seamlessly with SQL databases using SQLAlchemy, making it suitable for production applications.
- Testing is straightforward with built-in testing capabilities, and performance benchmarks highlight FastAPI's efficiency.

End of Study Notes