Implementing REST using FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with 3.6+ based on standard type hints. It’s designed to be easy to use and to help developers quickly create robust, production-ready APIs. Below are the core concepts and the framework components that make FastAPI powerful:

**Core Concepts**

1. **ASGI (Asynchronous Server Gateway Interface)**:
   * **Purpose**: The next generation standard for web servers and applications. It allows for asynchronous programming, which can handle many requests concurrently, improving performance for I/O-bound applications.
   * **Usage**: FastAPI is built on ASGI and can run with any ASGI server like Uvicorn or Daphne.
2. **Pydantic**:
   * **Purpose**: Data validation and settings management using type annotations.
   * **Usage**: FastAPI uses Pydantic for data validation, serialization, and deserialization. It ensures that data coming into your API is valid and allows you to easily work with complex data types.
3. **Type Hints**:
   * **Purpose**: Provide type hints for function arguments and return values to ensure type safety.
   * **Usage**: FastAPI leverages ’s type hints to automatically generate request validation, documentation, and serialization.
4. **Swagger and ReDoc**:
   * **Purpose**: Automatic generation of interactive API documentation.
   * **Usage**: FastAPI automatically generates and serves documentation with Swagger UI and ReDoc, making it easy to test and understand your APIs.
5. **Dependency Injection**:
   * **Purpose**: A design pattern to implement IoC (Inversion of Control) for resolving dependencies.
   * **Usage**: FastAPI’s dependency injection system allows you to define and manage dependencies in a flexible and reusable way.

**Framework Components**

1. **Application Instance**:
   * The main entry point of a FastAPI application. You create an instance of the FastAPI class, which serves as the central object for your API.

from fastapi import FastAPI

app = FastAPI()

1. **Path Operations**:
   * **Purpose**: Define endpoints (routes) to handle different HTTP methods (GET, POST, PUT, DELETE, etc.).
   * **Usage**: Path operations are defined using decorators like @app.get(), @app.post(), etc.

@app.get("/items/{item\_id}")

def read\_item(item\_id: int):

return {"item\_id": item\_id}

1. **Request Handling**:
   * **Purpose**: Handle request data and validate it using Pydantic models.
   * **Usage**: Define request body, query parameters, and path parameters using Pydantic models and type hints.

from pydantic import BaseModel

class Item(BaseModel):

name: str

description: str = None

price: float

tax: float = None

@app.post("/items/")

def create\_item(item: Item):

return item

1. **Response Handling**:
   * **Purpose**: Define and return responses, including status codes and custom response models.
   * **Usage**: Use Pydantic models for response validation and customization.

from pydantic import BaseModel

class ItemResponse(BaseModel):

name: str

price\_with\_tax: float

@app.post("/items/", response\_model=ItemResponse)

def create\_item(item: Item):

item.price\_with\_tax = item.price + (item.tax if item.tax else 0)

return item

1. **Dependency Injection**:
   * **Purpose**: Inject dependencies such as database connections, configurations, or other services into your path operations.
   * **Usage**: Define dependencies using Depends and reusable functions.

from fastapi import Depends

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/items/")

def read\_items(skip: int = 0, limit: int = 10, db: Session = Depends(get\_db)):

items = db.query(Item).offset(skip).limit(limit).all()

return items

1. **Middleware**:
   * **Purpose**: Execute code globally before and after each request.
   * **Usage**: Add custom middleware using @app.middleware.

from fastapi import Request

@app.middleware("http")

async def add\_process\_time\_header(request: Request, call\_next):

start\_time = time.time()

response = await call\_next(request)

process\_time = time.time() - start\_time

response.headers["X-Process-Time"] = str(process\_time)

return response

1. **Background Tasks**:
   * **Purpose**: Run tasks in the background while returning an immediate response to the client.
   * **Usage**: Use BackgroundTasks to define and run background operations.

from fastapi import BackgroundTasks

def write\_log(message: str):

with open("log.txt", "a") as log:

log.write(message)

@app.post("/send-notification/")

def send\_notification(background\_tasks: BackgroundTasks, email: str):

background\_tasks.add\_task(write\_log, f"Notification sent to {email}")

return {"message": "Notification sent"}

1. **Asynchronous Support**:
   * **Purpose**: Leverage ’s async and await syntax to handle asynchronous operations.
   * **Usage**: Define path operations as async functions to handle asynchronous I/O operations efficiently.

import httpx

@app.get("/async-items/")

async def get\_async\_items():

async with httpx.AsyncClient() as client:

response = await client.get("https://api.example.com/items")

return response.json()

**Example Project Structure**

css

myproject/

├── main.py

├── models.py

├── schemas.py

├── database.py

├── crud.py

├── dependencies.py

├── routers/

│ ├── \_\_init\_\_.py

│ ├── items.py

│ └── users.py

**Summary**

* **FastAPI** is a modern web framework for building APIs with 3.6+.
* **ASGI**: FastAPI is built on ASGI for high performance.
* **Pydantic**: Used for data validation and serialization.
* **Type Hints**: FastAPI leverages type hints for request validation and documentation.
* **Swagger and ReDoc**: Automatically generated interactive API documentation.
* **Dependency Injection**: FastAPI's built-in system for managing dependencies.

FastAPI provides a simple yet powerful way to create APIs that are fast, secure, and well-documented, making it an excellent choice for modern web API development.

Implementing REST APIs using FastAPI is straightforward and efficient, thanks to its modern, fast, and highly performant capabilities. FastAPI is designed for building APIs with 3.6+ based on standard type hints. Here's a detailed guide on how to get started:

**1. Set Up FastAPI**

First, you need to install FastAPI and a web server like Uvicorn to serve your FastAPI application.

pip install fastapi uvicorn

**2. Create a Basic FastAPI Application**

Start by creating a basic FastAPI application. Create a file named main.py.

# main.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

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

You can run this application using Uvicorn:

uvicorn main:app --reload

**3. Define Data Models with Pydantic**

FastAPI uses Pydantic for data validation. You can define your data models using Pydantic.

# models.py

from pydantic import BaseModel

from typing import Optional

class Book(BaseModel):

title: str

author: str

published\_date: Optional[str] = None

isbn: Optional[str] = None

pages: Optional[int] = None

cover: Optional[str] = None

language: Optional[str] = None

**4. Create API Endpoints**

Define your API endpoints for CRUD operations. Update your main.py to include these endpoints.

# main.py

from fastapi import FastAPI, HTTPException

from typing import List

from models import Book

app = FastAPI()

# Temporary in-memory storage

books = []

@app.get("/books", response\_model=List[Book])

def get\_books():

return books

@app.post("/books", response\_model=Book)

def create\_book(book: Book):

books.append(book)

return book

@app.get("/books/{book\_id}", response\_model=Book)

def get\_book(book\_id: int):

if book\_id >= len(books) or book\_id < 0:

raise HTTPException(status\_code=404, detail="Book not found")

return books[book\_id]

@app.put("/books/{book\_id}", response\_model=Book)

def update\_book(book\_id: int, book: Book):

if book\_id >= len(books) or book\_id < 0:

raise HTTPException(status\_code=404, detail="Book not found")

books[book\_id] = book

return book

@app.delete("/books/{book\_id}", status\_code=204)

def delete\_book(book\_id: int):

if book\_id >= len(books) or book\_id < 0:

raise HTTPException(status\_code=404, detail="Book not found")

books.pop(book\_id)

return None

**5. Running the Server**

Run your FastAPI application using Uvicorn:

uvicorn main:app --reload

**6. Adding Database Support with SQLAlchemy**

FastAPI doesn't include ORM support by default, but you can easily integrate SQLAlchemy.

First, install the required packages:

pip install sqlalchemy databases

Update your project structure:

css

myproject/

├── main.py

├── models.py

├── database.py

├── crud.py

**Define Database Configuration**:

# database.py

from sqlalchemy import create\_engine, MetaData

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

DATABASE\_URL = "sqlite:///./test.db"

engine = create\_engine(DATABASE\_URL)

metadata = MetaData()

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

**Define SQLAlchemy Models**:

# models.py

from sqlalchemy import Column, Integer, String

from database import Base

class Book(Base):

\_\_tablename\_\_ = "books"

id = Column(Integer, primary\_key=True, index=True)

title = Column(String, index=True)

author = Column(String)

published\_date = Column(String, nullable=True)

isbn = Column(String, nullable=True)

pages = Column(Integer, nullable=True)

cover = Column(String, nullable=True)

language = Column(String, nullable=True)

**Create the Database**:

# Create the database tables

from database import engine

from models import Base

Base.metadata.create\_all(bind=engine)

**CRUD Operations**:

# crud.py

from sqlalchemy.orm import Session

from models import Book as BookModel

from models import Book as BookSchema

def get\_books(db: Session, skip: int = 0, limit: int = 10):

return db.query(BookModel).offset(skip).limit(limit).all()

def create\_book(db: Session, book: BookSchema):

db\_book = BookModel(\*\*book.dict())

db.add(db\_book)

db.commit()

db.refresh(db\_book)

return db\_book

def get\_book(db: Session, book\_id: int):

return db.query(BookModel).filter(BookModel.id == book\_id).first()

def update\_book(db: Session, book\_id: int, book: BookSchema):

db\_book = db.query(BookModel).filter(BookModel.id == book\_id).first()

if db\_book:

for key, value in book.dict().items():

setattr(db\_book, key, value)

db.commit()

db.refresh(db\_book)

return db\_book

def delete\_book(db: Session, book\_id: int):

db\_book = db.query(BookModel).filter(BookModel.id == book\_id).first()

if db\_book:

db.delete(db\_book)

db.commit()

return db\_book

**Update API Endpoints**:

# main.py

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from database import SessionLocal, engine

from models import Base, Book as BookModel

from schemas import Book as BookSchema

import crud

app = FastAPI()

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

@app.get("/books", response\_model=List[BookSchema])

def read\_books(skip: int = 0, limit: int = 10, db: Session = Depends(get\_db)):

books = crud.get\_books(db, skip=skip, limit=limit)

return books

@app.post("/books", response\_model=BookSchema)

def create\_book(book: BookSchema, db: Session = Depends(get\_db)):

return crud.create\_book(db=db, book=book)

@app.get("/books/{book\_id}", response\_model=BookSchema)

def read\_book(book\_id: int, db: Session = Depends(get\_db)):

db\_book = crud.get\_book(db, book\_id=book\_id)

if db\_book is None:

raise HTTPException(status\_code=404, detail="Book not found")

return db\_book

@app.put("/books/{book\_id}", response\_model=BookSchema)

def update\_book(book\_id: int, book: BookSchema, db: Session = Depends(get\_db)):

db\_book = crud.update\_book(db, book\_id=book\_id, book=book)

if db\_book is None:

raise HTTPException(status\_code=404, detail="Book not found")

return db\_book

@app.delete("/books/{book\_id}", status\_code=204)

def delete\_book(book\_id: int, db: Session = Depends(get\_db)):

db\_book = crud.delete\_book(db, book\_id=book\_id)

if db\_book is None:

raise HTTPException(status\_code=404, detail="Book not found")

return None

**Running the Server with Database Support**

Run your FastAPI application using Uvicorn:

uvicorn main:app --reload

**Summary**

1. **Install FastAPI and Uvicorn**: Set up the basic FastAPI application.
2. **Define Data Models**: Use Pydantic for request validation and response serialization.
3. **Create API Endpoints**: Define CRUD endpoints for your resources.
4. **Database Integration**: Use SQLAlchemy for database interactions, encapsulate CRUD operations, and ensure the API endpoints interact with the database.
5. **Run and Test**: Start the FastAPI server and test your endpoints.

By following these steps, you can effectively implement a REST API using FastAPI, providing a fast, modern, and type-safe way to build web APIs.