FastAPI Fundamentals

Building a Personal Banking Management System

- 🚀 Routers, Services, and the Repository Pattern
- 🗄️ SQLAlchemy ORM Basics
- ⚙️ Pydantic for Configuration

Agenda

1. **Introduction to FastAPI:** What and Why?
2. **Project Structure:** The Clean Architecture Approach
3. **Core Components:**
   - **Routers: Handling HTTP Requests**
   - **Services: Implementing Business Logic**
   - **Repositories: Abstracting Data Access**
4. **Database Integration:** SQLAlchemy ORM
5. **Configuration:** Pydantic Settings Management
6. **Conclusion & Key Takeaways**

banking_api.zip

What is FastAPI?

- A modern, high-performance web framework for building APIs with Python.
- Built on top of **Starlette** (for web parts) and **Pydantic** (for input/data validation).

Key Features:

⚡ **Fast:** Very high performance, on par with NodeJS and Go.

🚀 **Fast to code:** Increases development speed by 200% to 300%.

🐛 **Fewer bugs:** Reduces about 40% of human-induced errors.

🧠 **Intuitive:** Great editor support, completion everywhere.

📚 **Easy:** Designed to be easy to use and learn.

📖 **Automatic Docs:** Interactive API documentation (Swagger UI and ReDoc).

The Big Picture: Our Goal is to Build APIs

What is an API?

An Application Programming Interface. In simple terms, it's how our different software applications talk to each other over the web.

At FIL, we build APIs to:

- Provide data to our front-end websites and mobile apps
- Connect different internal services together
- Create powerful, reusable logic

Our tool for this job? FastAPI.

The Foundation: What is Starlette? 🚀

Before we learn FastAPI, we need to meet its "secret weapon": Starlette.

Starlette is a lightweight, high-performance Python web **toolkit**.

Its two superpowers are:

1. **Extreme Speed:** It's built on a modern standard called ASGI
2. **Asynchronous Support:** It's designed from the ground up for async/await

> **Analogy:** Think of Starlette as a powerful, high-performance car engine. It's minimalist but incredibly fast and efficient.

The Star of the Show: What is FastAPI? ✨

FastAPI is a modern, fast web framework for building APIs.

The Key Relationship:

FastAPI is **built directly on top of Starlette**. It *uses* Starlette for all its core power.

Analogy: If Starlette is the engine, **FastAPI is the entire high-tech car built around it**. It takes the engine's power and adds a sleek body, an easy-to-use dashboard, GPS, and amazing safety features.

Why This Matters: FastAPI Inherits Starlette's Superpowers

Because FastAPI is built on Starlette, it gets all of its benefits for free!

- **Incredible Performance:** FastAPI is one of the fastest Python web frameworks *because* Starlette is one of the fastest toolkits
- **Full Async Support:** async/await is a native feature, allowing us to build highly concurrent applications that don't get stuck waiting for databases or other APIs
- **Rock-Solid Foundation:** We are building on top of reliable, well-tested code

So... What Does FastAPI Add to the Party? 🤔

If Starlette is the engine, what are the "car parts" that FastAPI adds on top?

1. **Automatic Interactive API Docs** 📖
   This is a game-changer. FastAPI automatically generates a web page where you can see and test your API endpoints live. (Powered by OpenAPI & Swagger UI)

2. **Powerful Data Validation** ✅
   Using a library called Pydantic, FastAPI automatically checks, validates, and documents all your data. This massively reduces bugs.

3. **Dependency Injection System** 💉
   An elegant system for managing resources like database connections, security, and configurations.

Our Journey: From Engine to Race Car

> Now you know the secret: **FastAPI's magic comes from the power of Starlette**.

We won't be writing pure Starlette code. Instead, we'll be using FastAPI, which gives us the best of both worlds.

So, let's stop looking at the engine and **start driving the car!**

Next up: Let's build our first FastAPI endpoint.

The Project: Personal Banking API

We will build a simple API to manage bank accounts.

Core Features:

- Create a new bank account
- View account details by ID
- Get a list of all accounts
- Deposit money into an account
- Withdraw money from an account

> 💡 **Pro Tip:** This project will serve as our practical example for learning FastAPI concepts.

Think first of the Architecture?

Architecture must help to **separate concerns**, making the **application clean**, **testable**, and **maintainable**.

- Organized
- Easy to Test
- Flexible & Maintainable
- Adaptable

The Restaurant Analogy

How to Organize Our Code for Success

Why Do We Need Architecture? 🤔

A Simple, Clean Architecture

Imagine a chaotic kitchen where every cook does everything: takes orders, cooks, and runs to the storeroom. It would be a mess!

Problem:

Code without a clear structure becomes slow, buggy, and impossible to update.

Solution:

We use an architectural pattern called **Separation of Concerns** .

This means every part of our code has exactly one clear job, just like a well-organised restaurant team.

Meet Our Restaurant Team!

A Simple, Clean Architecture

Our API project will have three main layers, just like a restaurant has three main roles for handling an order.

The Waiter

(Handles Customer Interaction)

The Head Chef

(Manages the Cooking Logic)

The Storeroom Manager

(Handles Ingredients)

Layer 1: The Router (Our Waiter) 🤵

A Simple, Clean Architecture

The Router is the public face of our API. It's the only part that talks directly to the user (the "customer").

Their Job:

- **Takes the order:** Receives the incoming HTTP request.
- **Checks the menu:** Validates that the data is correct (e.g., the customer isn't ordering something impossible).
- **Passes order to the kitchen:** Calls the right method in the Service layer.
- **Delivers the food:** Takes the result from the Service and sends back an HTTP response.

The Waiter's Mantra: "I don't cook. I just talk to customers and the kitchen."

Layer 2: The Service (Our Head Chef) 👨‍🍳

A Simple, Clean Architecture

The Service layer is the "brains" of the operation. This is where all the rules and logic happen.

Their Job:

- **Follows the recipe:** Contains all the business logic (e.g., how to calculate a total price, apply a discount, or combine ingredients).
- **Manages the kitchen:** Orchestrates the process. If it needs data, it asks the Storeroom Manager.
- **Prepares the final dish:** Assembles the final result to be given back to the Waiter.

The Chef's Mantra: "I don't talk to customers or get ingredients myself. I am the expert on the recipe."

Layer 3: The Repository (Our Storeroom Manager) 📦

The Repository's only job is to manage data. It is the only layer that touches the database (the "storeroom").

Their Job:

- **Gets ingredients:** Fetches data from the database when the Chef asks for it.
- **Puts away deliveries:** Saves new or updated data to the database.
- **Knows where everything is:** Abstracts the database. The Chef doesn't need to know if ingredients are in the fridge or the pantry; they just ask the Storeroom Manager.

The Storeroom Manager's Mantra: "I don't cook or talk to customers. I am the only one allowed in the storeroom."

The Full Workflow: A Customer Orders a Pizza

Let's see how they all work together!

1. User sends a request (POST /orders).

2. Router (Waiter) receives the request, validates the data (e.g., quantity > 0), and calls order_service.create_pizza().

3. Service (Chef) gets the call. The "recipe" says it needs ingredients. It calls pizza_repository.get_ingredients().

4. Repository (Storeroom Manager) gets the call, writes a query (SELECT * FROM ingredients...), and gets the data from the Database (Storeroom).

5. Repository returns the ingredients to the Service.

6. Service applies its logic (assembles the pizza conceptually) and returns a "success" message to the Router.

7. Router creates a 201 Created HTTP response and sends it back to the User.

The Benefits: Why Our Restaurant is a Success! ✅

This architecture makes our project clean, testable, and maintainable.

🎯 Organized

Everyone knows their job. No more messy code!

🧪 Easy to Test

We can test the Chef's recipe (Service logic) without needing a real database or a live web request.

🔄 Flexible & Maintainable

Change database? Only the Repository (Storeroom Manager) needs to change. The Chef's recipe stays the same!

🧮 Adaptable

New interface? Only the Router (Waiter) needs to change. The Chef's cooking process stays the same!

Architecture: Routers, Services, Repositories

This pattern separates concerns, making the application clean, testable, and maintainable.

Router (Controller)

- Receives HTTP requests
- Validates incoming data
- Calls service methods
- Returns HTTP response

🏦 *A bank teller who receives requests from customers*

Service

- Contains business logic
- Orchestrates operations
- No direct DB interaction

🏦 *The bank manager who approves transactions based on bank rules*

Repository

- Handles DB communication
- Abstracts data source
- Provides CRUD methods

🏦 *The bank's secure vault and filing system*

Setting Up the Project Structure

A clean structure is key to a scalable application.

```
/banking_api
|-- /app
|   |-- __init__.py
|   |-- main.py             # FastAPI app instance
|   |-- database.py         # Database connection setup
|   |-- config.py           # Configuration settings
|   |-- /routers
|   |   |-- __init__.py
|   |   |-- account_router.py
|   |-- /services
|   |   |-- __init__.py
|   |   |-- account_service.py
|   |-- /repositories
|   |   |-- __init__.py
|   |   |-- account_repository.py
|   |-- /models
|   |   |-- __init__.py
|   |   |-- account.py        # SQLAlchemy model
|   |-- /schemas
|   |   |-- __init__.py
|   |   |-- account_schema.py # Pydantic schemas
|-- requirements.txt
```

The Router Layer

app/routers/account_router.py

- Uses APIRouter to group related endpoints
- Handles request validation using Pydantic schemas
- Depends on the Service layer to perform actions

```python
from fastapi import APIRouter, Depends, HTTPException
from ..services.account_service import AccountService
from ..schemas.account_schema import AccountCreate, AccountOut

router = APIRouter(prefix="/accounts", tags=["Accounts"])

@router.post("/", response_model=AccountOut)
def create_account(
    account_data: AccountCreate,
    service: AccountService = Depends()
):
    return service.create_account(account_data)

@router.get("/{account_id}", response_model=AccountOut)
def get_account(account_id: int, service: AccountService = Depends()):
    account = service.get_account(account_id)
    if not account:
        raise HTTPException(status_code=404, detail="Account not found")
    return account
```

💡 **Pro Tip:** FastAPI's Depends() system automatically creates instances, handles nested dependencies, and makes testing easy by allowing dependency overrides.

Short Exercise 1: Create "Get All Accounts" Endpoint

Task:

Add a new endpoint to account_router.py that retrieves a list of all bank accounts.

Requirements:

- **Path:** / (relative to router prefix, so /accounts/)
- **HTTP Method:** GET
- **Response:** A list of accounts, response_model should be list[AccountOut]
- **Logic:** Call a new method in AccountService called get_all_accounts()

The Service Layer

app/services/account_service.py

- Contains the application's business logic
- Doesn't know about HTTP requests or databases
- Depends on the Repository layer for data

```
from fastapi import Depends
from ..repositories.account_repository import AccountRepository
from ..schemas.account_schema import AccountCreate
from ..models.account import Account

class AccountService:
    def __init__(self, repo: AccountRepository = Depends()):
        self.repo = repo

    def get_account(self, account_id: int):
        return self.repo.get_by_id(account_id)

    def get_all_accounts(self):
        return self.repo.get_all()

    def create_account(self, account_data: AccountCreate) -> Account:
        # Business logic: new accounts start with a balance of 0
        new_account = Account(
            owner_name=account_data.owner_name,
            balance=0.0
        )
        return self.repo.create(new_account)
```

The Repository Layer & SQLAlchemy

app/repositories/account_repository.py

The only layer that interacts directly with the database. Uses SQLAlchemy Session to execute queries.

app/models/account.py (The SQLAlchemy ORM Model)

```
from sqlalchemy import Column, Integer, String, Float
from ..database import Base

class Account(Base):
    __tablename__ = "accounts"

    id = Column(Integer, primary_key=True, index=True)
    owner_name = Column(String, index=True)
    balance = Column(Float, default=0.0)
```

Repository Implementation

app/repositories/account_repository.py

```python
from sqlalchemy.orm import Session
from fastapi import Depends
from ..database import get_db
from ..models.account import Account

class AccountRepository:
    def __init__(self, db: Session = Depends(get_db)):
        self.db = db

    def get_by_id(self, account_id: int) -> Account | None:
        return self.db.query(Account).filter(Account.id == account_id).first()

    def get_all(self) -> list[Account]:
        return self.db.query(Account).all()

    def create(self, account: Account) -> Account:
        self.db.add(account)
        self.db.commit()
        self.db.refresh(account)
        return account
```

💡 **Pro Tip:** The get_db dependency is a generator that yields a database session. FastAPI ensures this session is always closed after the request, preventing connection leaks.

Short Exercise 2: Implement Deposit Logic

Task:

Add the logic to deposit money into an account.

Requirements:

- **Repository:** Create update(account: Account) method in AccountRepository
- **Service:** Create deposit(account_id: int, amount: float) method:
  - Fetch the account using repository
  - Return None if account doesn't exist
  - Raise ValueError if amount is negative
  - Add amount to balance and save
- **Router:** Create POST endpoint at /accounts/{account_id}/deposit

Solution 2: Implement Deposit Logic

1. Repository (account_repository.py)

```
# ...
    def update(self, account: Account) -> Account:
        self.db.commit()
        self.db.refresh(account)
        return account
```

2. Service (account_service.py)

```
# ...
    def deposit(self, account_id: int, amount: float) -> Account | None:
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")

        account = self.repo.get_by_id(account_id)
        if not account:
            return None

        account.balance += amount
        return self.repo.update(account)
```

3. Router (account_router.py)

```
from pydantic import BaseModel

class DepositRequest(BaseModel):
    amount: float

@router.post("/{account_id}/deposit", response_model=AccountOut)
def deposit_money(
    account_id: int,
    request: DepositRequest,
    service: AccountService = Depends()
):
    try:
        updated_account = service.deposit(account_id, request.amount)
        if not updated_account:
            raise HTTPException(status_code=404, detail="Account not found")
        return updated_account
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
```

Configuration Management with Pydantic

Hardcoding database URLs or secret keys is a bad practice. We use Pydantic's BaseSettings to manage configuration.

How it works:

- Reads variables from environment variables or a .env file
- Provides type validation and default values

app/config.py

```
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    DATABASE_URL: str = "sqlite:///./test.db"
    APP_NAME: str = "Personal Banking API"

    class Config:
        env_file = ".env"

settings = Settings()
```

.env file (in the root directory)

```
DATABASE_URL="postgresql://user:password@host:port/dbname"
```

💡 **Pro Tip:** By creating a single settings object, you have a single source of truth for all configuration. This makes it easy to manage different environments by simply changing environment variables.

Conclusion

Today we learned how to structure a robust FastAPI application using a clean, layered architecture.

- 🌐 **Routers** handle the web layer
- 🧠 **Services** contain our core business logic
- 🗄️ **Repositories** abstract away the database
- 🔗 **SQLAlchemy** provides a powerful ORM for database interaction
- ✅ **Pydantic** is essential for both data validation and configuration

This separation of concerns leads to code that is **maintainable**, **scalable**, and **easy to test**.

Key Takeaways

🏗️ **Structure Matters:** A good project structure is your best friend. The Router/Service/Repository pattern is a powerful choice.

🪶 **Leverage Dependency Injection:** Use Depends() for clean, testable code. It's one of FastAPI's best features.

📊 **Separate Data Shapes:** Use SQLAlchemy models for database tables and Pydantic schemas for API contracts. Don't mix them.

🧠 **Keep Business Logic in Services:** Your services should be pure Python, unaware of HTTP or SQL. This makes them easy to reuse and test.

⚙️ **Externalize Configuration:** Never hardcode settings. Use Pydantic's BaseSettings to load config from the environment.