SQLAlchemy Understanding

Understanding Your First Database Model with SQLAlchemy

This code uses a powerful Python library called **SQLAlchemy** to define the structure of a database table. Think of it as creating a blueprint for how to store bank account information.

📝 What the Code Does

This code defines a Python class called `Account` that maps directly to a database table named `"accounts"`. An Object-Relational Mapper (ORM) like SQLAlchemy lets you work with your database using familiar Python objects instead of writing raw SQL commands.

In simple terms: You're telling your program, "I want a table called 'accounts', and every row in it should look like this."

🧐 Line-by-Line Breakdown

```
from sqlalchemy import Column, Integer, String, Float
```

This line **imports** the necessary building blocks from the SQLAlchemy library.

- `Column`: Used to define a column in a database table
- `Integer`, `String`, `Float`: These specify the **data type** for each column

🧐 Line-by-Line Breakdown

```
from ..database import Base
```

This imports a special `Base` class from another file in your project. This `Base` acts as a central registry for all your database models.

```
class Account(Base):
```

This creates a Python class named `Account`. It inherits from `Base`, which means it's officially registered as a SQLAlchemy database model.

```
__tablename__ = "accounts"
```

This special attribute explicitly sets the **name of the database table** to `"accounts"`.

The Columns (The actual data fields)

```
id = Column(Integer, primary_key=True, index=True)
```

- `Integer`: Stores whole numbers
- `primary_key=True`: Makes this the **unique identifier**
- `index=True`: Makes searching by ID much faster

```
owner_name = Column(String, index=True, nullable=False)
```

- `String`: Stores text (account holder's name)
- `nullable=False`: This field **cannot be empty**

```
balance = Column(Float, default=0.0, nullable=False)
```

- `Float`: Stores decimal numbers
- `default=0.0`: New accounts start with $0.00
- `nullable=False`: Balance field cannot be empty

🎨 Visualizing the Result

Table: accounts

id (Primary Key)	owner_name (String)	balance (Float)
1	Abinash	1500.75
2	Rahul	250.00
3	Arun	9850.25
...	...	...

Instead of writing complex SQL:

```
INSERT INTO accounts ...
```

You can simply create a Python object:

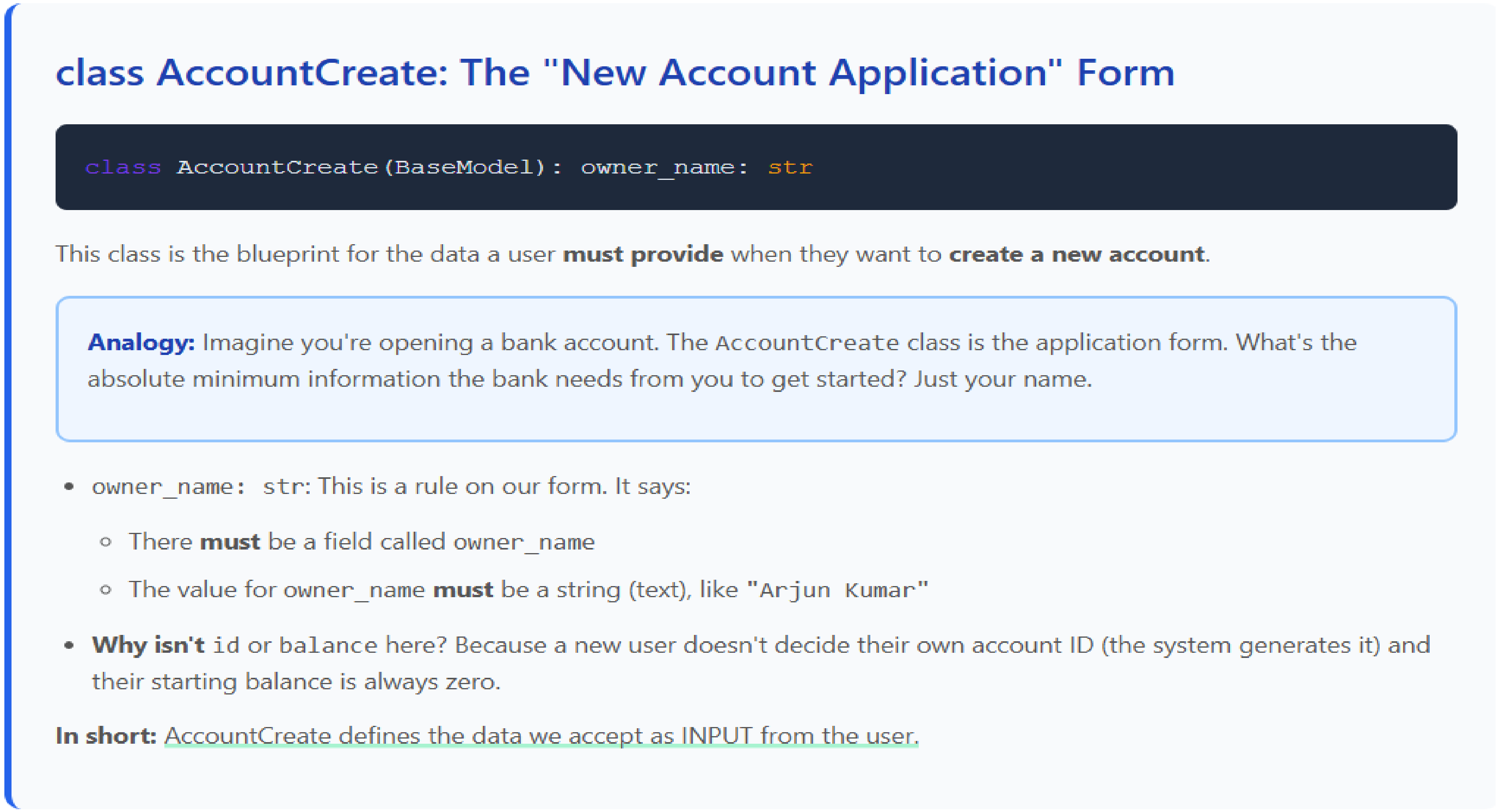
```
new_account = Account(owner_name="David", balance=50.0)
```

SQLAlchemy handles all the database queries for you!

Pydantic Understanding

Understanding Pydantic Models: The "Data Blueprints" 📝

Think of these classes not as regular code, but as **blueprints** or **forms**. They define the exact structure and type of data our application should expect or send. Pydantic uses these blueprints to automatically validate data for us.



class AccountCreate: The "New Account Application" Form

```
class AccountCreate(BaseModel): owner_name: str
```

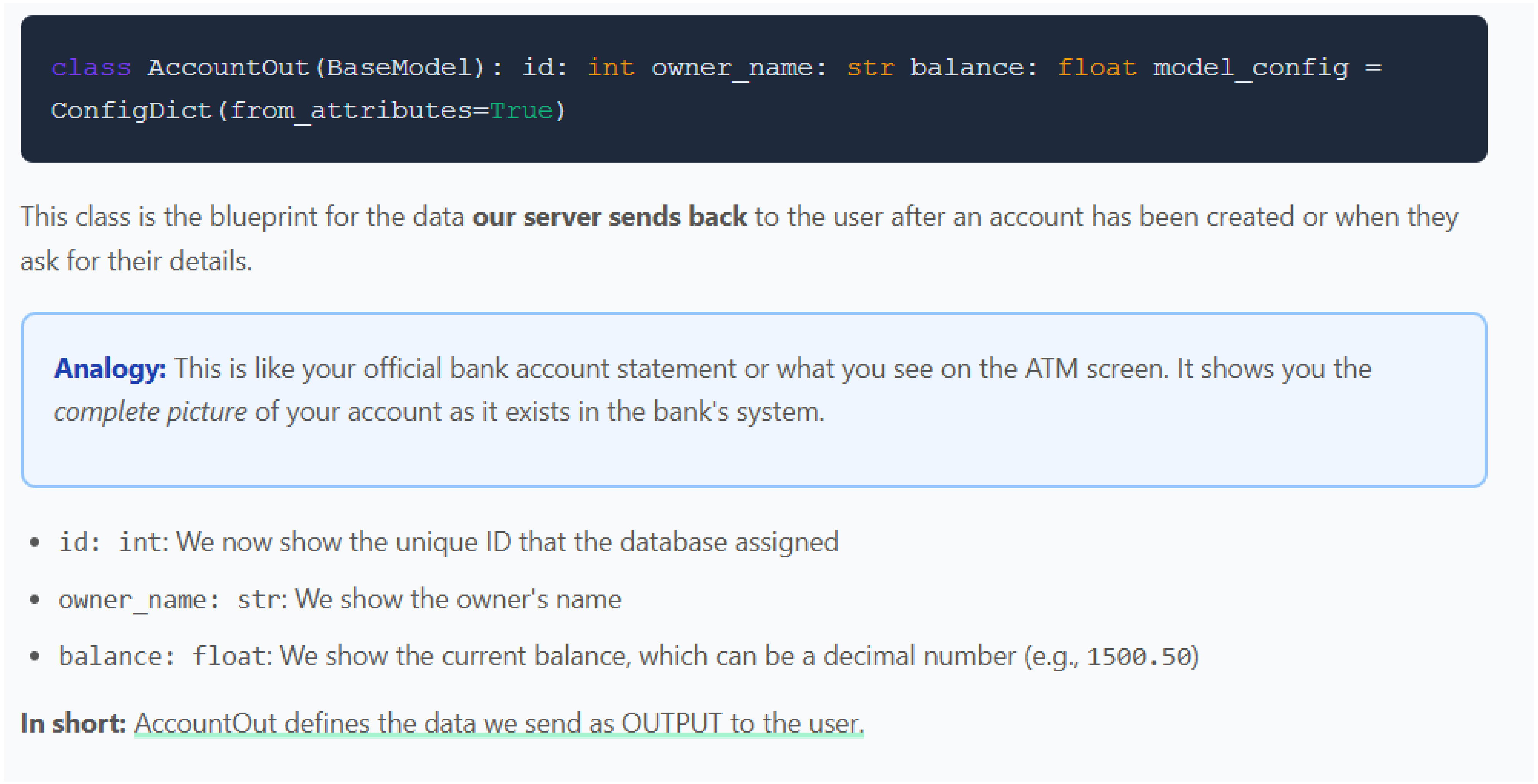
This class is the blueprint for the data a user **must provide** when they want to **create a new account**.

Analogy: Imagine you're opening a bank account. The `AccountCreate` class is the application form. What's the absolute minimum information the bank needs from you to get started? Just your name.

- `owner_name: str`: This is a rule on our form. It says:
  - There **must** be a field called owner_name
  - The value for owner_name **must** be a string (text), like `"Arjun Kumar"`
- **Why isn't** `id` or `balance` here? Because a new user doesn't decide their own account ID (the system generates it) and their starting balance is always zero.

In short: AccountCreate defines the data we accept as INPUT from the user.

class AccountOut: The "Account Statement" Template



```
class AccountOut(BaseModel): id: int owner_name: str balance: float model_config =
ConfigDict(from_attributes=True)
```

This class is the blueprint for the data **our server sends back** to the user after an account has been created or when they ask for their details.

Analogy: This is like your official bank account statement or what you see on the ATM screen. It shows you the *complete picture* of your account as it exists in the bank's system.

- `id: int`: We now show the unique ID that the database assigned
- `owner_name: str`: We show the owner's name
- `balance: float`: We show the current balance, which can be a decimal number (e.g., `1500.50`)

In short: AccountOut defines the data we send as OUTPUT to the user.

What's That Magic model_config Line? ⚙️

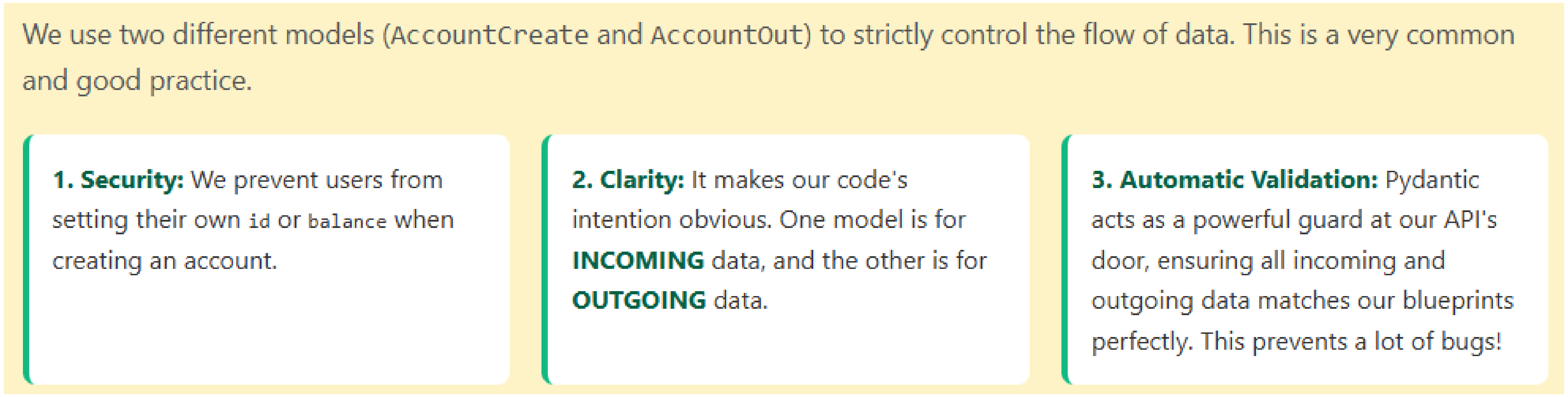
```
model_config = ConfigDict(from_attributes=True)
```

This is a special instruction for Pydantic.

- **The Problem:** Our database gives us data as a Python **object** (e.g., an `Account` object from SQLAlchemy). This object has attributes you access with a dot, like `my_account.id` and `my_account.owner_name`.
- **The Solution:** This line tells `AccountOut`, "Hey, you might get a database object instead of a plain dictionary. That's okay! Just **read the data from the object's attributes** to fill in your fields."

Without this line, Pydantic would get confused and wouldn't know how to read from a database object.

Key Takeaway: Why Use Two Separate Classes?



We use two different models (`AccountCreate` and `AccountOut`) to strictly control the flow of data. This is a very common and good practice.

1. Security: We prevent users from setting their own `id` or `balance` when creating an account.

2. Clarity: It makes our code's intention obvious. One model is for **INCOMING** data, and the other is for **OUTGOING** data.

3. Automatic Validation: Pydantic acts as a powerful guard at our API's door, ensuring all incoming and outgoing data matches our blueprints perfectly. This prevents a lot of bugs!

FastAPI Depends()

The Magic Glue for Dependency Injection

Depends() is FastAPI's Dependency Injection system that automatically provides necessary resources to your endpoints without manual creation.

What Depends() Does

1 Automatic Creation

Creates and provides service instances automatically

2 Layer Connection

Connects Router, Service, and Repository layers cleanly

3 Easy Testing

Enables simple mocking and testing of components

DI in FastAPI Understanding

FastAPI Depends()

The Magic Glue for Dependency Injection

Code Comparison

❌ Without Depends() (Bad Practice)

```python
def get_account(account_id: int):
    # Manually creating every time
    service = AccountService()
    account = service.get_account(account_id)
    # ...
```

✅ With Depends() (Good Practice)

```python
def get_account(...,
    service: AccountService = Depends()):
    # Service magically provided!
    account = service.get_account(account_id)
    # ...
```

FastAPI Depends()

Architecture Connection

Router (get_account function) = The "Waiter" taking orders

Service (AccountService) = The "Head Chef" processing requests

Depends() = The system ensuring Waiters get direct access to the Chef automatically

🎯 Key Takeaway

Depends() is not just a shortcut—it's the core of FastAPI's design for writing **clean**, **decoupled**, and **highly testable** code. It handles the "how" of getting necessary components so your endpoint function can focus on the "what."

Repository Layer Understanding

Code Deep Dive: The AccountRepository 🧐

Understanding the Repository Pattern & Database Operations

💡 The Big Idea

This code creates a class that acts as a dedicated **"data manager"** for user Account information. It uses a common and powerful design called the **Repository Pattern**.

Core Concept: Instead of writing database logic all over our application, we centralize it in one place. This class is the only thing that should talk directly to the Account table in the database.

⚙️ Breaking Down the Code

1. The __init__ Method (The Setup)

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

- **__init__**: Constructor that runs every time we create a new AccountRepository object
- **db: Session**: SQLAlchemy Session - your active connection to the database
- **Depends(get_db)**: FastAPI's Dependency Injection magic ✨ - automatically provides database session

Code Deep Dive: The AccountRepository 🧐

Understanding the Repository Pattern & Database Operations

2. Reading Data (get_by_id and get_all) 🚀

```
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()
```

- `get_by_id` : Finds a single account by ID or returns None
- `self.db.query(Account)` : Start a query on the Account table
- `.filter()` : Apply WHERE condition to match specific ID
- `.first() / .all()` : Execute query and return first result or all results

Code Deep Dive: The AccountRepository 🧐

Understanding the Repository Pattern & Database Operations

3. Creating Data (create) 💾

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

Three-Step Transaction Process:

- **Stage** : self.db.add(account) - Add to session (pending)
- **Save** : self.db.commit() - Write to database permanently
- **Update** : self.db.refresh(account) - Get updated info (auto-generated ID, defaults)

🎯 Key Takeaways

Repository Pattern: Keeps code clean, organized, and testable by separating database logic from business logic

Dependency Injection: FastAPI's Depends automatically provides necessary components like database sessions

Session is King: The db session object is your gateway to all database operations with SQLAlchemy

Commit is Final: Changes are not saved to database until you call .commit() - remember this!

Router Layer Understanding

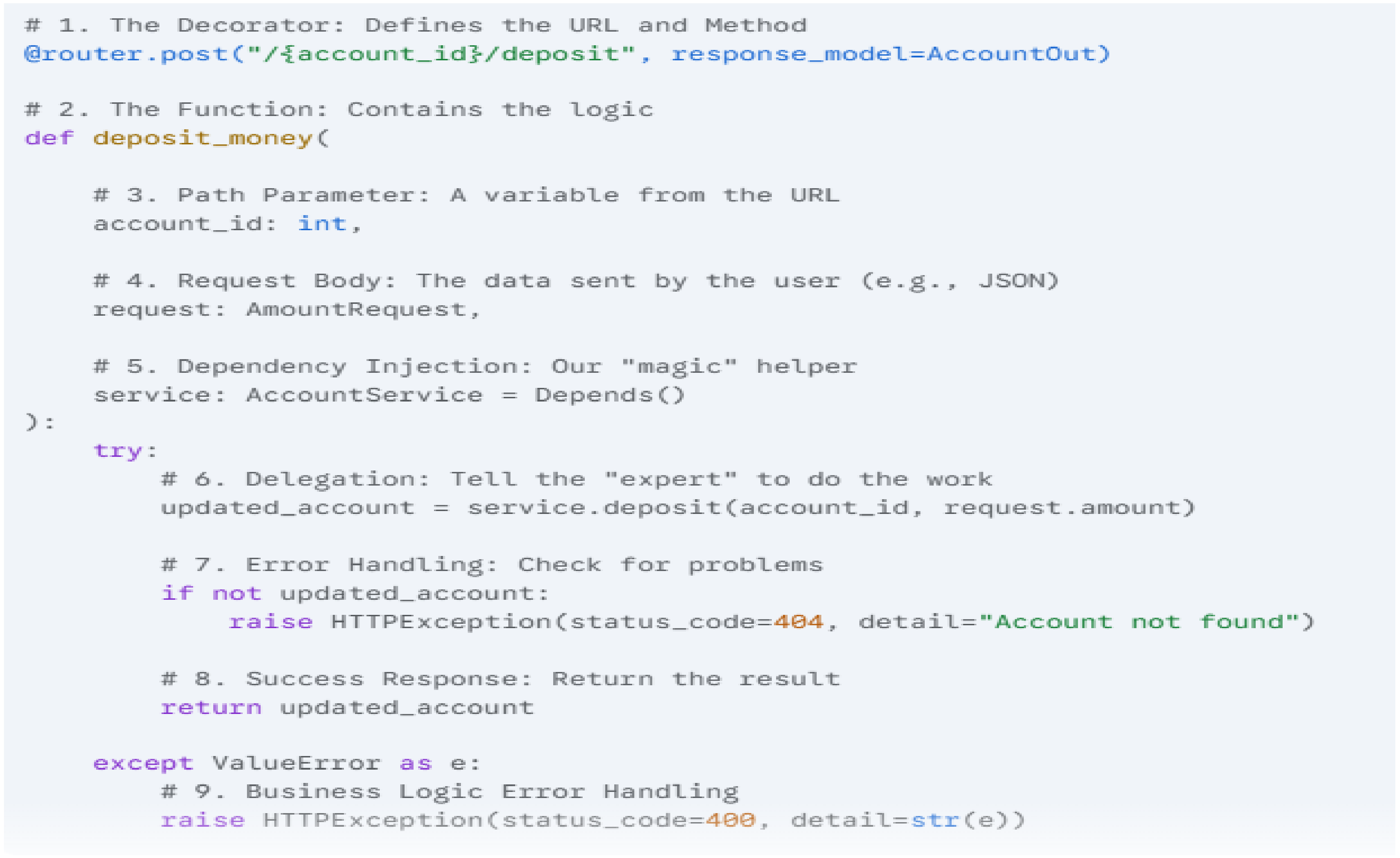
Understanding the API Layer: The "Accounts" Router

🏦 The Big Picture: What This Code Does

Think of this file as the **front desk or receptionist** for our application's bank accounts. Its main job is to handle incoming web requests and direct them to the correct "expert" to do the actual work.

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

Anatomy of an Endpoint: deposit_money Function



```python
# 1. The Decorator: Defines the URL and Method
@router.post("/{account_id}/deposit", response_model=AccountOut)

# 2. The Function: Contains the logic
def deposit_money(

    # 3. Path Parameter: A variable from the URL
    account_id: int,

    # 4. Request Body: The data sent by the user (e.g., JSON)
    request: AmountRequest,

    # 5. Dependency Injection: Our "magic" helper
    service: AccountService = Depends()
):
    try:
        # 6. Delegation: Tell the "expert" to do the work
        updated_account = service.deposit(account_id, request.amount)

        # 7. Error Handling: Check for problems
        if not updated_account:
            raise HTTPException(status_code=404, detail="Account not found")

        # 8. Success Response: Return the result
        return updated_account

    except ValueError as e:
        # 9. Business Logic Error Handling
        raise HTTPException(status_code=400, detail=str(e))
```

🧠 Key Concepts

1. Separation of Concerns

Router (Receptionist): Handles HTTP requests/responses, validates data

Service (Expert): Contains business logic, database operations

Why? Makes code cleaner, easier to test and maintain

2. Pydantic Models

Define the **"contract"** for our data:

- **AccountCreate/AmountRequest:** What user sends
- **AccountOut:** What we return

FastAPI uses these for automatic validation & documentation

3. Dependency Injection

The **"magic"** that provides objects to our functions

- Makes code decoupled
- Router doesn't create dependencies
- Makes testing incredibly easy

Service Layer Understanding

Understanding the Service Layer

The "Account" Expert 🧑‍💼

The Big Picture: What This Code Does

The AccountService contains the **business logic** for managing accounts. Business logic is the rules of your application that make it work correctly.

Rule 1: New accounts must start with a balance of $0

Rule 2: You cannot deposit a negative amount of money

Rule 3: You cannot withdraw more money than you have

Code in Action: The Deposit Method

Business Logic Example

```
def deposit(self, account_id: int, amount: float):
    # Rule Check: Is the amount valid?
    if amount <= 0:
        raise ValueError("Deposit amount must be positive")

    # Get current data from repository
    account = self.repo.get_by_id(account_id)

    # Perform the business operation
    account.balance += amount

    # Save the updated data
    return self.repo.update(account)
```

Understanding the Service Layer

The "Account" Expert 🧑‍💼

Three-Layer Architecture

API Layer / Router

(The Receptionist)

- Handles HTTP requests
- Talks to outside world
- Calls Service Layer

Service Layer

(The Bank Manager)

- Contains business rules
- Coordinates operations
- Calls Repository Layer

Repository Layer

(The File Clerk)

- Handles database communication
- CREATE, READ, UPDATE, DELETE
- Only talks to database

🧠 The Service Layer is the BRAIN of your application - it contains all the rules and logic that make your business work correctly!

Database Layer Understanding

Database Connection Setup

SQLAlchemy Configuration for FastAPI Applications

📚 Library Analogy

- Database = Library Building (full of books/data)
- SQLAlchemy = Multilingual Librarian (ORM)
- Engine = Librarian's main computer & connection
- Session = Your personal library card & temporary desk

Why This Matters:

Just like you get a library card for each visit and return it when done, each API request gets a fresh session that's properly closed afterward.

🔧 Core Setup Process

- Create Engine with database URL from config
- SessionLocal = Factory for new sessions
- Base = Blueprint for all database tables
- get_db() = Safe session management function

Safety Pattern:

try...yield...finally ensures every database connection is properly closed, preventing resource leaks.

Engine

Single powerful connection to database. Created once and reused.

Session

Short-lived conversation with database for single task/request.

get_db()

FastAPI dependency that safely provides & closes sessions.