Transaction Flows in a Software System

A Deep Dive into the transaction system for Python

Professional Development Training

Master the art of building robust, scalable banking transaction systems with industry-standard frameworks and best practices.

Transactions Flow_python.zip

📋 Training Agenda - PYTHON

- 🎯 Introduction: What are transactions and why are they critical in banking?
- ⚡ The ACID Properties: A quick refresher
- 🔄 Transaction Flow in Python (FastAPI & SQLAlchemy)
- 📦 SQLAlchemy Session Management
- ⚡ Async External Service Calls
- 🎨 AOP using Decorators
- 🚨 FastAPI Exception Handlers
- ✨ Best Practices & Pro Tips
- 🎯 Conclusion & Key Takeaways
- ❓ Q&A Session

Introduction: What are Transactions?

What is a Transaction?

A transaction is a single, indivisible unit of work that must either be completed fully or not at all. Think of it as a promise: **"I will do all of these steps, or I will do none of them."**

🏦 Why are they Critical in Banking?

Example: Bank Transfer from Alice to Bob for ₹100

This involves two separate operations:

1. Debit ₹100 from Alice's account
2. Credit ₹100 to Bob's account

Introduction: What are Transactions?

What if the system crashes after Step 1 but before Step 2?

- Alice has lost ₹100
- Bob has received nothing
- The bank's records are now inconsistent and incorrect!

Transactions prevent this!

The debit and credit are wrapped in a single transaction. If any part fails, the entire operation is cancelled (rolled back), as if it never happened.

The ACID Properties: A Quick Refresher

ACID is a set of properties that guarantees database transactions are processed reliably. It's the foundation of trust in our systems.

A - Atomicity

The "all or nothing" rule. The bank transfer either fully completes (both debit and credit) or it fails and no changes are saved.

C - Consistency

Ensures a transaction brings the database from one valid state to another. The total money in the bank remains the same; it just moves between accounts.

I - Isolation

Transactions running concurrently will not interfere with each other. One transaction will wait for the other to complete before reading data.

D - Durability

Once a transaction is successfully completed (committed), it is saved permanently. It will survive system crashes, power outages, etc.

Transaction Flow in FastAPI & SQLAlchemy

Let's see how this looks in our code. There are two main paths:

✅ The Happy Path (Success)

1. HTTP Request: A user requests to create a new practitioner via POST to /practitioners

2. Dependency: FastAPI provides a database Session to the endpoint

3. Business Logic: The service uses the session to create a new Practitioner object

4. Commit: No errors occurred, session changes are committed to the database

5. HTTP Response: A 201 Created response is sent back to the user

Transaction Flow in FastAPI & SQLAlchemy

❌ The Unhappy Path (Failure)

1-3. Same as above

4. Error Occurs: An exception is raised (missing field, constraint violated)

5. Rollback: All session changes are rolled back

6. HTTP Response: A 4xx or 5xx error response is sent back

SQLAlchemy Session Management

The Session object is the heart of our database interaction in SQLAlchemy. It's like a temporary workspace or "scratchpad" where we stage our changes before saving them.

How We Manage It: The Dependency Pattern

In FastAPI, we use Dependency Injection to manage the session's lifecycle for each request.

Standard get_db dependency:

```
# in app/db/session.py or app/deps.py

from app.db.session import SessionLocal

def get_db():
    db = SessionLocal()  # 1. Create a new session for this request
    try:
        yield db         # 2. Provide the session to the endpoint
    finally:
        db.close()       # 3. ALWAYS close the session when done
```

SQLAlchemy Session Management

Using it in an endpoint:

```
# in app/api/routes/practitioners.py

from sqlalchemy.orm import Session
from fastapi import Depends, APIRouter

router = APIRouter()

@router.post("/")
def create_practitioner(
    *,
    db: Session = Depends(get_db),  # FastAPI calls get_db for us
    practitioner_in: PractitionerCreate
):
    # ... use the 'db' session here ...
    return practitioner
```

Critical: The try...finally block guarantees that the database session is closed, even if errors occur, preventing resource leaks.

The Challenge: Async External Service Calls

What if our operation needs to call another service?

Like sending an SMS or email?

The Problem:

Database transactions should be short-lived. Holding a transaction open while waiting for a slow network call is a major performance bottleneck. It locks database rows, preventing other users from working with that data.

⛔ The WRONG Way

```
def create_and_notify(
    db: Session,
    payload: PractitionerCreate
):
    # 1. Start transaction
    new_practitioner = Practitioner(
        **payload.dict()
    )
    db.add(new_practitioner)

    # 2. Slow network call
    # DB is locked here!
    await send_welcome_email(
        payload.email
    )

    # 3. Commit
    db.commit()
```

✅ The RIGHT Way

```
def create_and_notify(
    db: Session,
    payload: PractitionerCreate
):
    # 1. Create and add
    new_practitioner = Practitioner(
        **payload.dict()
    )
    db.add(new_practitioner)

    # 2. Commit FIRST
    db.commit()
    db.refresh(new_practitioner)

    # 3. External call AFTER
    await send_welcome_email(
        new_practitioner.email
    )

    return new_practitioner
```

Key Principle: This releases the database lock as quickly as possible. If the email fails, you can handle it separately (e.g., with a background retry mechanism).

AOP using Decorators for Cleaner Code

Managing try/except/commit/rollback in every service function is repetitive. We can use Aspect-Oriented Programming (AOP) via Python decorators to handle this cleanly.

AOP helps us separate our core business logic from "cross-cutting concerns" like transaction management, logging, or authentication.

AOP stands for Aspect-oriented programming

AOP using Decorators for Cleaner Code

Creating a @transactional Decorator

```
# in app/utils/transaction.py

from functools import wraps
from app.db.session import SessionLocal

def transactional(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        db = SessionLocal()
        try:
            # Pass session to the function
            result = func(db=db, *args, **kwargs)
            db.commit()
            return result
        except Exception as e:
            db.rollback()
            raise e  # Re-raise the exception
        finally:
            db.close()
    return wrapper
```

The Challenge

Manual Database Session Management

Every database operation required following these steps manually:

- Open a database session connection
- Execute business logic operations
- Commit changes if successful
- Rollback changes if errors occur
- Close the session to free resources

Problem: This repetitive pattern was prone to human error and led to code duplication across the entire codebase.

Before: Manual Transaction Handling

Repetitive and Error-Prone Code

```
def create_user_manual(user_data): db = SessionLocal() # 1. Open session try: new_user =
User(**user_data.dict()) db.add(new_user) # 2. Business logic db.commit() # 3. Save changes
except Exception as e: db.rollback() # 4. Undo on error raise e finally: db.close() # 5.
Always cleanup
```

⚠️ **Key Issues:**

- 15-20 lines of boilerplate per function
- Easy to forget critical steps like rollback or close
- Duplicated across hundreds of functions
- Business logic buried in infrastructure code

After: The @transactional Decorator

Clean, Safe, and Reusable

```
from app.utils.transaction import transactional @transactional def create_user(user_data: UserCreate, db: Session): new_user = User(**user_data.dict()) db.add(new_user) # That's it! No manual commit, rollback, or close
```

✨ **Benefits Achieved:**

- Reduced from 15-20 lines to just 3-4 lines
- Automatic transaction safety guaranteed
- Business logic is crystal clear
- Zero code duplication

Direct Comparison

See the Transformation

❌ Before

```
def create_user(data): db =
SessionLocal() try: user = User(**data)
db.add(user) db.commit() except Exception
as e: db.rollback() raise e finally:
db.close()
```

✅ After

```
@transactional def create_user( data:
UserCreate, db: Session ): user =
User(**data) db.add(user)
```

📊 75% reduction in code volume

Impact & Benefits

Why This Improvement Matters

🛡️ Enhanced Safety

Atomicity guaranteed - transactions are always all-or-nothing with automatic rollback

🎯 Code Clarity

Business logic separated from infrastructure concerns, easier to read and understand

♻️ Reusability

Write once, use everywhere - DRY principle applied effectively

🐛 Fewer Bugs

Eliminates common errors like forgotten rollbacks or unclosed sessions

⚡ Faster Development

Developers focus on business logic, not repetitive transaction handling

🔧 Easy Maintenance

Transaction logic updated in one place benefits entire application

Using the Decorator in the Service Layer

Now our service code becomes much cleaner and focuses only on business logic!

❌ Before: Repetitive Boilerplate

```
def create(payload):
    db = SessionLocal()
    try:
        obj = Practitioner(
            name=payload.name,
            specialty=payload.specialty
        )
        db.add(obj)
        db.commit()
        return obj
    except Exception as e:
        db.rollback()
        raise e
    finally:
        db.close()
```

✅ After: Clean Business Logic

```
@transactional
def create(db, payload):
    # NO try/except/commit/rollback!
    # The decorator handles it all.

    obj = Practitioner(
        name=payload.name,
        specialty=payload.specialty
    )
    db.add(obj)
    return obj
```

Using the Decorator in the Service Layer

Benefits of This Approach:

- Reduced code duplication across all service functions
- Focus on business logic instead of infrastructure
- Consistent error handling application-wide
- Easier to test and maintain
- Single place to modify transaction behavior

FastAPI Exception Handlers

What happens when our decorator re-raises a database error, like a UNIQUE constraint violation? By default, FastAPI shows a generic 500 Internal Server Error.

> **We can do better!** Create custom exception handlers to catch specific errors and return meaningful HTTP responses.

Example: Handling Duplicate Entries

Let's say a user tries to create a practitioner with a username that already exists. The database will raise an IntegrityError.

```python
# in main.py

from fastapi import FastAPI, Request, status
from fastapi.responses import JSONResponse
from sqlalchemy.exc import IntegrityError

app = FastAPI()

@app.exception_handler(IntegrityError)
async def integrity_error_exception_handler(
    request: Request,
    exc: IntegrityError
):
    # This runs whenever an IntegrityError is raised
    return JSONResponse(
        status_code=status.HTTP_409_CONFLICT,
        content={
            "detail": "A resource with this identifier already exists."
        },
    )

# ... include your routers ...
```

Now, instead of a 500 error, the user gets a clean 409 Conflict error, which is much more descriptive and professional.

Best Practices & Pro Tips ⭐

1. **Keep Transactions Short:** The longer a transaction is open, the more database resources it locks. Do your work and commit/rollback quickly.
2. **Commit First, Then Act:** Perform non-transactional actions (like sending emails, calling external APIs) after the database commit.
3. **One Unit of Work:** A single API endpoint should ideally perform a single unit of work and map to a single transaction.
4. **Use Decorators:** Abstract away transaction logic from your business logic for cleaner, more maintainable code.
5. **Handle Exceptions Gracefully:** Use FastAPI's exception handlers to turn raw database errors into meaningful API responses.
6. **Leverage Pydantic:** Use Pydantic models for request validation. Catching bad data early prevents unnecessary database calls.

Conclusion & Key Takeaways

What We Learned Today:

- **Transactions are a promise of "all or nothing,"** ensuring data integrity
- **ACID properties** (Atomicity, Consistency, Isolation, Durability) are the theoretical foundation that makes this promise reliable
- **In FastAPI,** we manage the transaction lifecycle per-request using a Session managed by a dependency
- **Clean code patterns** like decorators and custom exception handlers separate concerns and make our application robust
- **Performance matters:** keep transactions short and avoid network calls within them

Remember: Good transaction management is the backbone of reliable, scalable applications. Master these patterns and you'll build systems that users can trust.

Q&A Session

Ready for your questions!

Q: What's the difference between db.commit() and db.flush()?

A:

db.flush() sends your SQL commands to the database but does not end the transaction. The changes are pending and visible within your current session but not to others. db.commit() makes the changes permanent and ends the transaction. You usually just need db.commit().

Q: Can I have one endpoint that uses multiple transactions?

A:

You can, but you should question why. It's usually a sign that your endpoint is doing too much. Try to break it down into smaller, single-purpose endpoints.

Q&A Session

Ready for your questions!

Q: How do I know if I'm in a transaction?

A:

As soon as you start interacting with the Session object provided by our dependency, you are inside a transaction block that is waiting to be committed or rolled back.

The ACID VS BASE

A reliable transaction system must guarantee four fundamental properties:

A
Atomicity

All operations within a transaction are completed successfully, or none are. The transaction is an "atomic" unit.

C
Consistency

The database remains in a valid state before and after the transaction. Fund transfers shouldn't violate balance rules.

I
Isolation

Concurrent transactions do not interfere with each other. Operations are hidden until transaction completion.

D
Durability

Once committed, changes are permanent and will survive system failures (power outage, crash, etc.).

Compare - BASE

BA
Basically Available

The system strives to remain operational and respond to requests, even if it cannot guarantee immediate consistency across all data nodes.

S
Soft State

The state of the system can change over time even without explicit input, due to eventual consistency mechanisms and the nature of distributed systems..

E
Eventual Consistency

Data across all nodes in the system will eventually converge to a consistent state, though temporary inconsistencies may exist immediately after updates.

What is SQLAlchemy?

The Python SQL Toolkit and Object Relational Mapper (ORM) 🐍 ⚙️ 💾

A powerful library for interacting with databases using Pythonic code.

Transform Your Database Interactions

SQLAlchemy bridges the gap between Python objects and database tables, making database operations intuitive and maintainable.

The Core Idea

Bridging Two Worlds: Python & SQL

SQLAlchemy allows you to work with your database in a more natural, Python-centric way.

Object Relational Mapper (ORM): It maps Python objects (your classes) to database tables. You manipulate objects, and SQLAlchemy translates that into SQL commands.

Database Agnostic: Write your Python code once and run it on different database systems (like PostgreSQL, MySQL, SQLite) with minimal changes.

Full SQL Power: Still gives you the ability to write raw SQL when you need optimal performance or complex queries.

How It Works (The ORM)

From Python Class to Database Table

You define a **model** in Python, and SQLAlchemy handles the database representation.

PYTHON CODE (YOUR MODEL):

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import
declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String)
```

RESULTING SQL (SIMPLIFIED):

```
CREATE TABLE users (
    id INTEGER NOT-NULL PRIMARY KEY,
    name VARCHAR,
    email VARCHAR
);
```

Key Benefits

Why Use SQLAlchemy?

🚀 Productivity

Write less boilerplate code. Focus on your application logic instead of repetitive SQL statements for common operations (Create, Read, Update, Delete).

🛠️ Maintainability

Your database logic lives with your Python code, making it easier to manage, version control, and understand.

🔐 Security

Helps prevent SQL injection attacks by using parameter binding and secure query construction.

⚡ Flexibility

Provides a powerful expression language for building queries programmatically, offering a great balance between high-level ORM and low-level SQL.

✨ Best Practices & Pro Tips

- ⚡ **Keep Transactions Short:** Long-running transactions lock database resources and hurt performance.
- 🔄 **Idempotency:** Design API endpoints to be idempotent. Use unique transaction IDs to prevent duplicates.
- 🔗 **Connection Pooling:** Always use database connection pools (HikariCP in Spring, SQLAlchemy's built-in pooling).
- ⚡ **Async All the Way:** In FastAPI, ensure database drivers and I/O libraries are also async.
- 🎯 **Propagation Levels:** Understand transaction propagation (REQUIRED vs REQUIRES_NEW) in Spring.

🔑 Key Takeaways

✅ Always wrap database operations that must succeed or fail together in a transaction.

🎨 Use AOP to keep your business logic clean. Log, monitor, and secure without cluttering.

🛡️ Handle exceptions gracefully. Failed transactions should rollback with clear error messages.

🌐 Be mindful of external calls. Handle them with proper timeouts and robust error handling.

AOP: Aspect-Oriented Programming

❓ Q&A Session

Questions? Discussion? Let's dive deeper!

Thank You! 🙏

Personal Finance Tracker

Building a Mini-Service for Budget Management & Financial Reporting

Audience

Python Developers

Duration

30-45 minutes

Format

Hands-on Exercise

pbms_transactionflow_python.zip

Mini-Project Overview

What We're Building

A simple Personal Finance Tracker with two core functions that will help you understand data modeling and basic API logic.

📊 Create Monthly Budget

Define spending limits for different categories (e.g., Groceries, Transport)

📈 Generate Spending Report

Compare actual spending against the set budget to see where your money went

This hands-on exercise will help you understand data modelling and basic API logic practically.

The Budget Data Model

First, we need to define what a "Budget" looks like. It's a simple structure that holds the spending limit for a specific category in a given month.

Language-Agnostic Representation (JSON)

```
{ "category": "Groceries", "allocatedAmount": 500.00, "month": "August", "year": 2025 }
```

JSON

```json
{
  "category": "Groceries",
  "allocatedAmount": 500.00,
  "month": "August",
  "year": 2025
}
```

Java Implementation

```java
public class Budget {
    private String category;
    private double allocatedAmount;
    private String month;
    private int year;

    // Constructors, Getters, and Setters
}
```

Python Implementation

```python
from dataclasses import dataclass

@dataclass
class Budget:
    category: str
    allocated_amount: float
    month: str
    year: int
```

Task 1 - Implement Create Budget API (Mock)

1 Create a function that acts like an API endpoint. It will accept budget data and save it in memory.

A mock API simulates real API behavior without needing a network or database. We'll use a simple in-memory list to store budgets.

Java Approach

```java
import java.util.ArrayList;
import java.util.List;

public class BudgetService {
    private static final List<Budget> budgetDatabase = new ArrayList<>();

    public String createBudget(Budget budget) {
        budgetDatabase.add(budget);
        System.out.println("Budget added: " + budget);
        return "Budget for " + budget.getCategory() + " created successfully!";
    }
}
```

Python Approach

```python
BUDGET_DATABASE = []

def create_budget(budget: Budget):
    """Simulates a POST API endpoint to create a budget."""
    BUDGET_DATABASE.append(budget)
    print(f"Budget added: {budget}")
    return f"Budget for {budget.category} created successfully!"
```

Task 2 - Generate a Financial Report

2 Create a report function that analyzes transactions and compares total spending against stored budgets.

First, let's define some sample transaction data:

Java Sample Data

```java
// Assuming a simple Transaction class exists
List<Transaction> transactions = List.of(
    new Transaction("Groceries", 75.50),
    new Transaction("Transport", 40.00),
    new Transaction("Groceries", 120.00)
);
```

Python Sample Data

```python
transactions = [
    {"category": "Groceries", "amount": 75.50},
    {"category": "Transport", "amount": 40.00},
    {"category": "Groceries", "amount": 120.00},
]
```

Report Generation Logic

The Logic (3 Steps):

- Calculate total spending for each category from transactions
- Find corresponding budget for each category
- Format and print a comparison

Python Implementation

```python
def generate_report(budgets: list[Budget], transactions: list[dict]):
    report_lines = ["--- Monthly Financial Report ---"]
    actual_spending = defaultdict(float)

    for trans in transactions:
        actual_spending[trans["category"]] += trans["amount"]

    budget_map = {b.category: b.allocated_amount for b in budgets}

    for category, spent in actual_spending.items():
        budgeted = budget_map.get(category, 0)
        status = "Under Budget" if spent <= budgeted else "Over Budget"
        report_lines.append(
            f"Category: {category} | Budget: ${budgeted:.2f} | "
            f"Spent: ${spent:.2f} | Status: {status}"
        )
    return "\n".join(report_lines)
```

Java Implementation

```java
public String generateReport(List<Budget> budgets, List<Transaction> transactic
    StringBuilder report = new StringBuilder("--- Monthly Financial Report ---\
    // Logic to:
    // 1. Group transactions by category and sum amounts.
    // 2. For each category, find the budget from the 'budgets' list.
    // 3. Append a formatted string like:
    //    "Category: Groceries | Budget: $500.00 | Spent: $195.50 | Status: Unc
    //    to the 'report' StringBuilder.
    return report.toString();
}
```

Putting It All Together

Let's simulate the entire process: create budgets and generate the report.

Java Main Method

```java
public static void main(String[] args) {
    BudgetService budgetService = new BudgetService();

    // 1. Create budgets using the mock API
    budgetService.createBudget(new Budget("Groceries", 500.00, "August", 2025)
    budgetService.createBudget(new Budget("Transport", 150.00, "August", 2025)

    // 2. Define sample transactions
    List<Transaction> transactions = ... ; // (from previous slide)
    List<Budget> budgets = budgetService.getBudgets(); // Assume a getter exis

    // 3. Generate and print the report
    String report = new ReportGenerator().generateReport(budgets, transactions
    System.out.println(report);
}
```

Python Script Execution

```python
if __name__ == "__main__":
    # 1. Create budgets using the mock API function
    create_budget(Budget("Groceries", 500.00, "August", 2025))
    create_budget(Budget("Transport", 150.00, "August", 2025))

    # 2. Define sample transactions (from previous slide)
    transactions = [...]

    # 3. Generate and print the report
    report = generate_report(BUDGET_DATABASE, transactions)
    print(report)
```

Conclusion & Key Learnings ✅

🎉 Congratulations!

In under 45 minutes, you've built the core logic for a personal finance tracker.

📊 Data Modeling

How to represent real-world concepts as structured data using Java Classes and Python Dataclasses/Dictionaries.

🔧 API Mocking

Creating mock APIs to simulate backend functionality, allowing for rapid development and testing.

💼 Business Logic

Implemented practical logic to process data—aggregating expenses and comparing them against limits.

🌐 Cross-Language Concepts

Fundamental principles of data structures and functions are universal across programming languages.

🏗 Modular Building

We created two distinct, reusable components: one for managing budgets and another for reporting.

Project Time: Building the "Fast-Funds Transfer API"

Putting Theory into Practice

The Big Picture: What Are We Building?

The Scenario: Abinash wants to send ₹1,000 to Rahul.

Our mission is to build the backend API that makes this happen safely and reliably.

Simple Flow:

1. A mobile app sends a request to our API: "Transfer ₹1,000 from account 123 to account 456."
2. Our FastAPI application receives this request.
3. It securely interacts with the database to update the account balances.
4. It sends back a "Success" message.

Your Mission, Should You Choose to Accept It...

You will build a single API endpoint: POST /transfer

✅ Be Atomic: The transfer must be all-or-nothing (This is the ACID challenge!).

✅ Handle Errors: What if Abinash’s account doesn't have ₹1,000? What if Rahul’s account doesn't exist? Our API shouldn't crash!

✅ Be Asynchronous: After a successful transfer, it must call an external service without blocking.

✅ Be Clean & Reusable: We will use modern Python patterns like Decorators to keep our code tidy.

The Core Endpoint: POST /transfer

This is the only endpoint you need to create.

HTTP Method: POST

URL: /transfer

Request Body (What the user sends us):

```
{
  "from_account_id": 1,
  "to_account_id": 2,
  "amount": 1000.00
}
```

Success Response (What we send back):

```
{
  "status": "success",
  "message": "Transfer of 1000.00 from account 1 to acc
}
```

The "What Ifs?" - Handling Failures

A great developer thinks about what can go wrong! Your API must gracefully handle cases like:

Insufficient Funds:

The from_account doesn't have enough money.

Action: Return an HTTP 400 Bad Request error with a clear message.

Account Not Found:

The from_account_id or to_account_id does not exist.

Action: Return an HTTP 404 Not Found error.

We will use FastAPI Exception Handlers to manage this elegantly.

The "And Also..." - Extra Features

To make our API truly production-grade, we'll add two more things:

Logging with Decorators (AOP):

Why? We need a record of every transaction attempt.

How? We'll create a @log_operation decorator to automatically log the function call.

Async Notification Call:

Why? After a successful transfer, Alice should get an SMS.

How? We'll use httpx to make a non-blocking POST request to a fake "Notification Service."

Let's Plan Our Attack! (The Approach)

Don't worry, we'll build this step by step. Here is our roadmap:

1. Foundation: Project setup and database models.
2. The Core Logic: Write the fund transfer function (the "Service Layer").
3. The Magic: Create the decorators for transactions and logging (AOP).
4. The Entrypoint: Build the FastAPI endpoint (the "API Layer").
5. The Safety Net: Implement the custom exception handlers.
6. The Final Touch: Add the asynchronous notification call.

🚀 **Let's get started!**

Step 1 - Foundation (Models & Schemas)

First, let's define our data structures.

1. SQLAlchemy Model (models.py): This is our database table.

```
# models.py
class Account(Base):
    id = Column(Integer, primary_key=True)
    owner_name = Column(String)
    balance = Column(Numeric(10, 2))
```

2. Pydantic Schemas (schemas.py): This defines our API request body.

```
# schemas.py
class TransferRequest(BaseModel):
    from_account_id: int
    to_account_id: int
    amount: float
```

Step 2 - The Heart of the Logic (Service Layer)

Create a function that contains the pure business logic for the transfer. Don't mix API code here!

```python
# services.py

# Define custom exceptions first!
class InsufficientFundsError(Exception): pass
class AccountNotFoundError(Exception): pass

def transfer_funds(db: Session, from_id: int, to_id: int, amount: float):
    # 1. Get accounts from DB
    from_account = db.query(Account).filter(Account.id == from_id).first()
    to_account = db.query(Account).filter(Account.id == to_id).first()

    # 2. Validate!
    if not from_account or not to_account:
        raise AccountNotFoundError("One or both accounts not found.")
    if from_account.balance < amount:
        raise InsufficientFundsError("Insufficient funds.")

    # 3. Perform the debit and credit
    from_account.balance -= amount
    to_account.balance += amount

    print("Transfer logic executed.")
```

Step 3 - The Magic (AOP with Decorators)

Now, let's create decorators to handle cross-cutting concerns, such as transactions and logging. This keeps our service logic clean!

Transaction Decorator:

```
# decorators.py
def transactional(db_session_arg_name="db"):
    def wrapper(func):
        @functools.wraps(func)
        def inner(*args, **kwargs):
            db = kwargs[db_session_arg_name]
            try:
                result = func(*args, **kwargs)
                db.commit() # COMMIT!
                return result
            except Exception as e:
                db.rollback() # ROLLBACK!
                raise e
        return inner
    return wrapper
```

This decorator automatically wraps a function in a commit/rollback block.

Step 4 - The Entrypoint (FastAPI Endpoint)

Now, let's expose our logic via a FastAPI route.

```
# main.py
from services import transfer_funds
from decorators import transactional

@app.post("/transfer")
@transactional(db_session_arg_name="db") # Our magic decorator!
async def perform_transfer(req: TransferRequest, db: Session = Depends(get_db)):
    # The endpoint is now SUPER clean!
    transfer_funds(
        db=db,
        from_id=req.from_account_id,
        to_id=req.to_account_id,
        amount=req.amount
    )
    # TODO: Add async notification call here later

    return {"status": "success", "message": "Transfer completed."}
```

> Notice how the decorator handles the session management for us!

Step 5 - The Safety Net (Exception Handlers)

If our service layer raises InsufficientFundsError, we don't want a 500 server error. Let's catch it and return a nice message.

```python
# main.py
from fastapi import Request, status
from fastapi.responses import JSONResponse
from services import InsufficientFundsError, AccountNotFoundError

@app.exception_handler(InsufficientFundsError)
async def insufficient_funds_handler(req: Request, exc: InsufficientFundsError):
    return JSONResponse(
        status_code=status.HTTP_400_BAD_REQUEST,
        content={"message": str(exc)},
    )

@app.exception_handler(AccountNotFoundError)
async def account_not_found_handler(req: Request, exc: AccountNotFoundError):
    return JSONResponse(
        status_code=status.HTTP_404_NOT_FOUND,
        content={"message": str(exc)},
    )
```

Step 6 - The Final Touch (Async Call)

Finally, after the transaction is committed, call the external service.

You're Ready to Code!

Your Task:

- ✓ Set up the project structure.
- ✓ Implement the models and schemas.
- ✓ Write the service, decorators, and endpoint.
- ✓ Add the exception handlers.
- ✓ Test your successful transfer and your error cases!

Good luck, and ask questions any time! 🎉