Banking Service
Architecture Specification

High-level and deep-dive architecture with component, data, class, and sequence diagrams

Author: Karan Mehta

Version: 1.0
Date: 2026-02-25

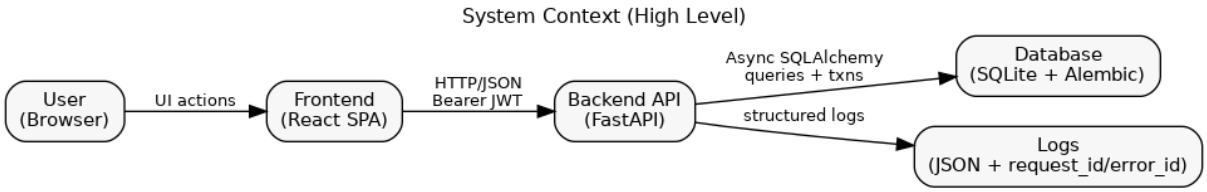
1. System Overview

This document describes the architecture of a small banking platform built as a take-home assessment. It includes a React frontend, a FastAPI backend, an SQLite database managed via Alembic migrations, and operational features such as structured logging, health probes, and environment-specific configuration.

2. High-Level Architecture

At runtime, the browser-based frontend calls the backend over HTTP/JSON using JWT for authentication. The backend enforces authorization and business invariants, persists state in SQLite, and emits structured logs.

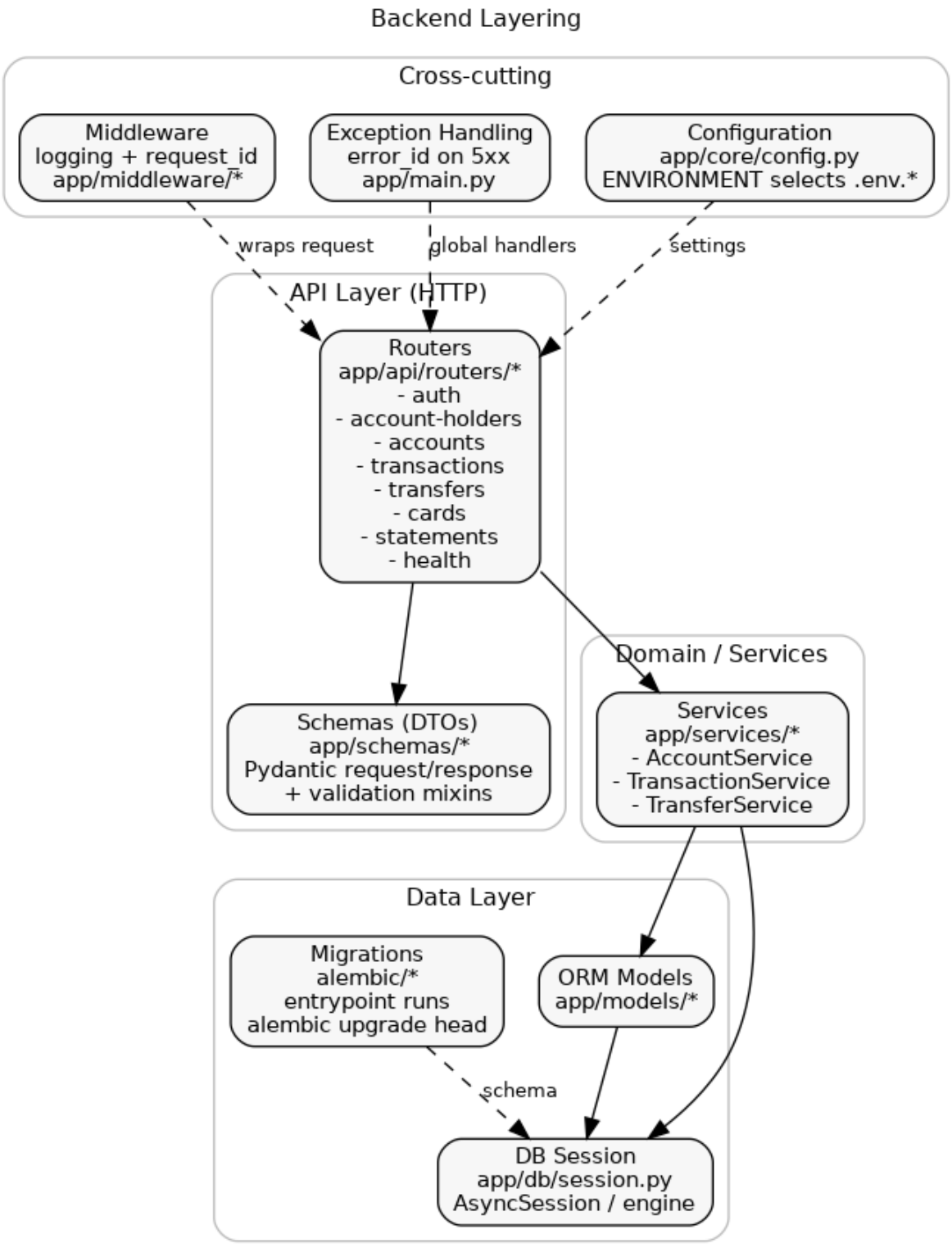
Figure: System Context (High Level)



3. Backend Deep Dive

The backend follows a layered approach: routers handle HTTP and authentication dependencies; schemas perform request/response validation; services encapsulate business logic and transaction boundaries; models and sessions provide persistence. Cross-cutting middleware adds request correlation, and global exception handlers generate error correlation IDs for 5xx responses.

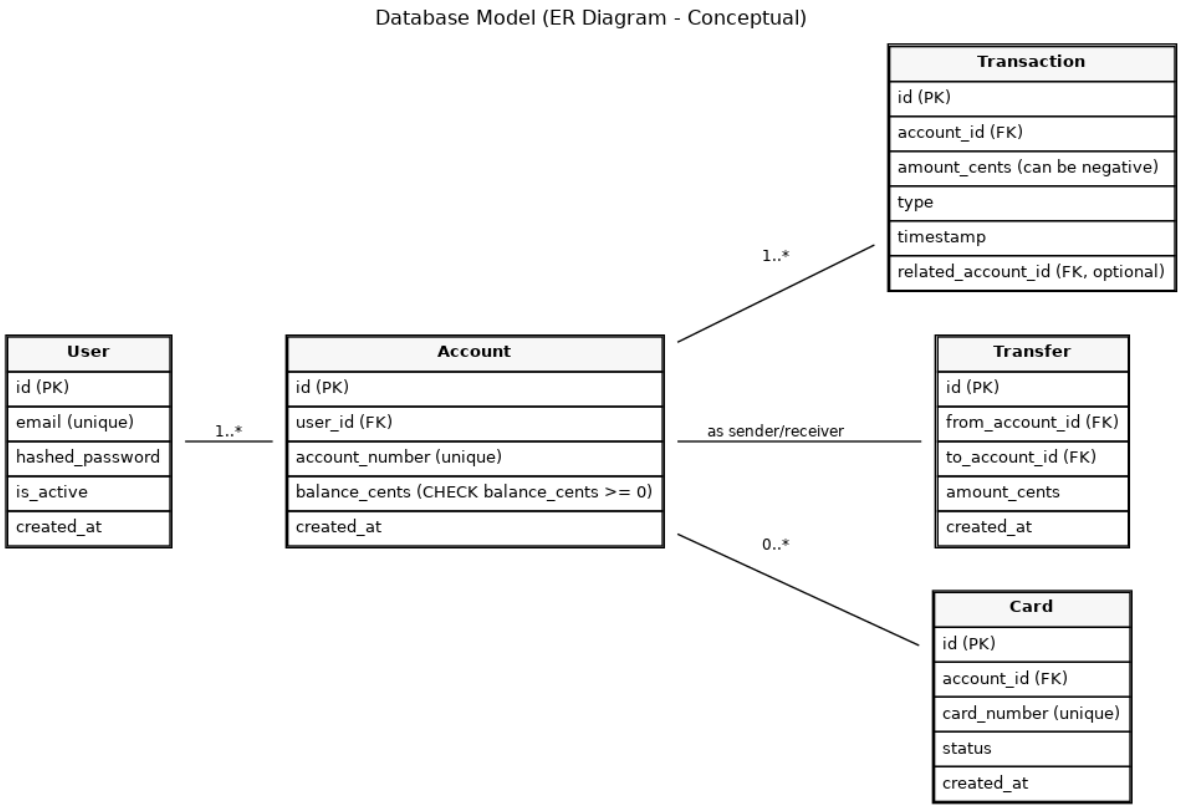
Figure: Backend Layering


Backend Layering
Cross-cutting
Middleware
logging + request_id
app/middleware/*
Exception Handling
error_id on 5xx
app/main.py
Configuration
app/core/config.py
ENVIRONMENT selects .env.*
wraps request
global handlers
settings
API Layer (HTTP)
Routers
app/api/routers/*
- auth
- account-holders
- accounts
- transactions
- transfers
- cards
- statements
- health
Schemas (DTOs)
app/schemas/*
Pydantic request/response
+ validation mixins
Domain / Services
Services
app/services/*
- AccountService
- TransactionService
- TransferService
Data Layer
Migrations
alembic/*
entrypoint runs
alembic upgrade head
ORM Models
app/models/*
schema
DB Session
app/db/session.py
AsyncSession / engine

4. Data Architecture

Core entities include users, accounts, transactions, transfers, and cards. Account balances are stored in cents, and debits can be represented as negative transaction amounts. Database integrity constraints enforce key invariants such as non-negative account balances.

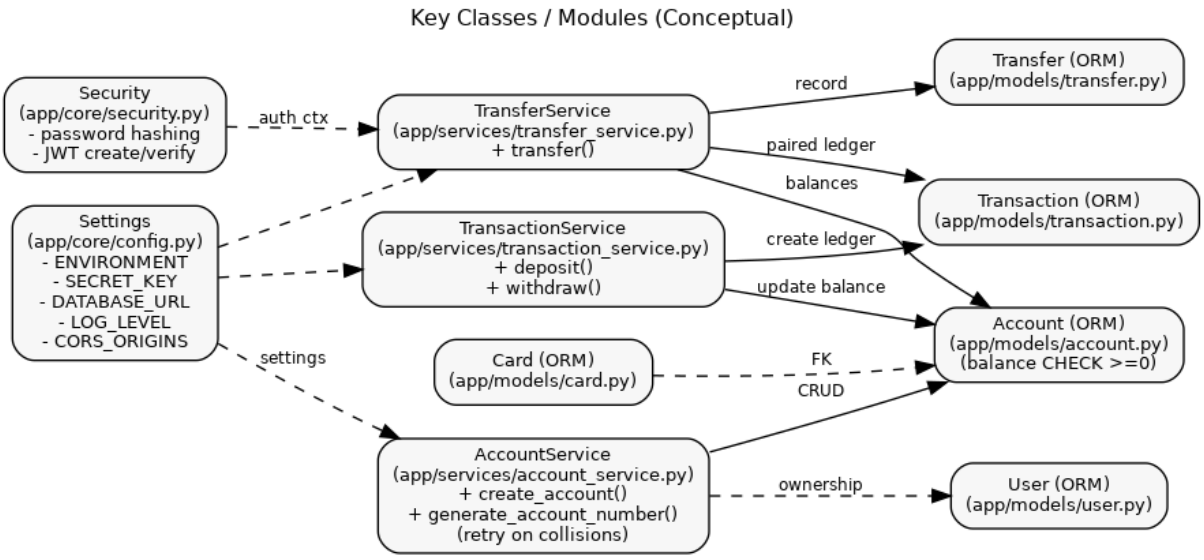
Figure: Database Model (ER Diagram - Conceptual)



5. Core Modules (Conceptual Class View)

The key modules include configuration (environment selection and secrets), security (password hashing and JWT), domain services (account creation, transaction posting, transfers), and ORM models.

Figure: Key Classes / Modules (Conceptual)

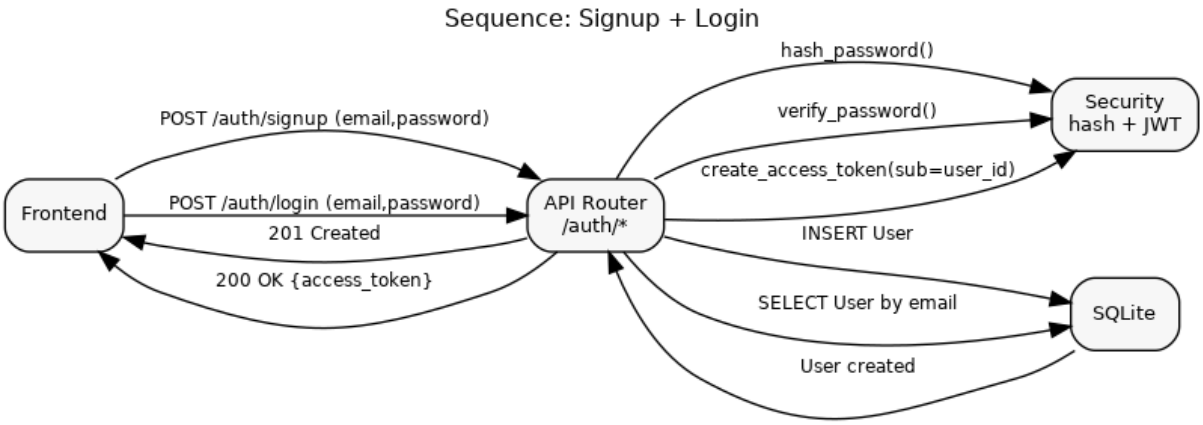

Key Classes / Modules (Conceptual)
Security
(app/core/security.py)
- password hashing
- JWT create/verify
auth_ctx
TransferService
(app/services/transfer_service.py)
+ transfer()
record
Transfer (ORM)
(app/models/transfer.py)
paired ledger
balances
Transaction (ORM)
(app/models/transaction.py)
Settings
(app/core/config.py)
- ENVIRONMENT
- SECRET_KEY
- DATABASE_URL
- LOG_LEVEL
- CORS_ORIGINS
TransactionService
(app/services/transaction_service.py)
+ deposit()
+ withdraw()
create ledger
update balance
Account (ORM)
(app/models/account.py)
(balance CHECK >=0)
settings
Card (ORM)
(app/models/card.py)
FK
CRUD
AccountService
(app/services/account_service.py)
+ create_account()
+ generate_account_number()
(retry on collisions)
ownership
User (ORM)
(app/models/user.py)

6. Key Sequences

6.1 Signup and Login

Signup creates a user with a hashed password. Login verifies credentials and returns a JWT access token.

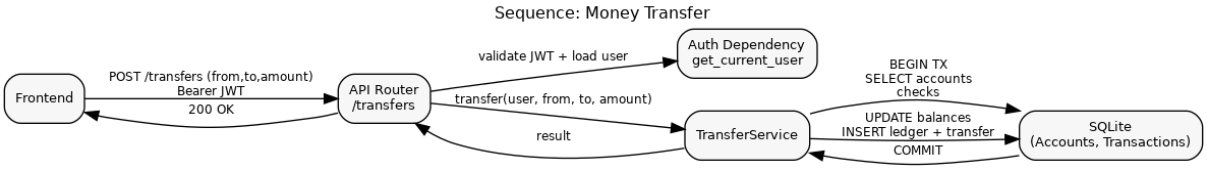
Figure: Sequence: Signup + Login



6.2 Money Transfer

Transfers validate sender ownership and balance, perform an atomic update of both account balances, and create paired ledger entries.

Figure: Sequence: Money Transfer

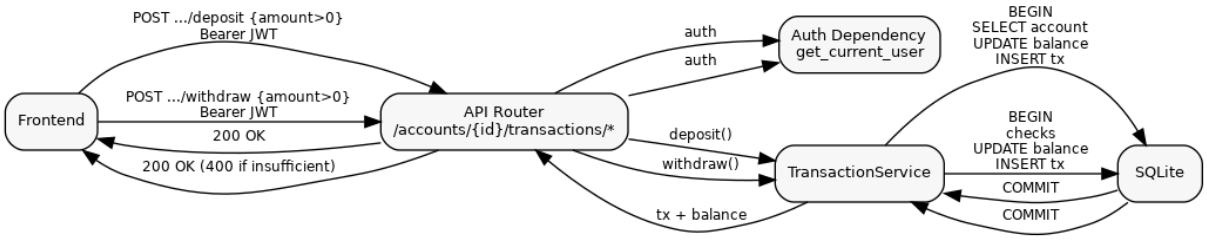


6.3 Deposit and Withdraw

Deposit and withdraw endpoints validate amount > 0 at the API boundary. Withdraw enforces 'Insufficient Funds' as a business rule.

Figure: Sequence: Deposit / Withdraw

Sequence: Deposit / Withdraw

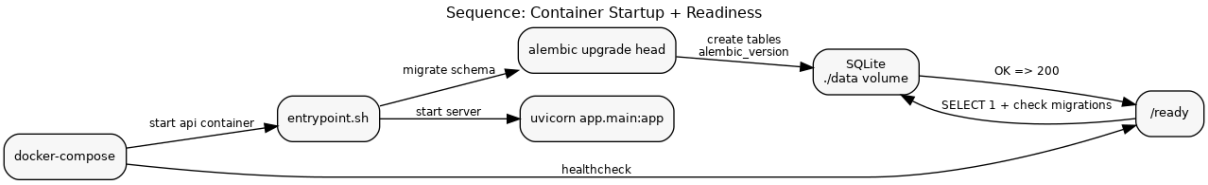


7. Operations and Reliability

7.1 Startup and Readiness

On container startup, Alembic migrations are applied before Uvicorn starts. Readiness checks validate database connectivity and schema readiness.

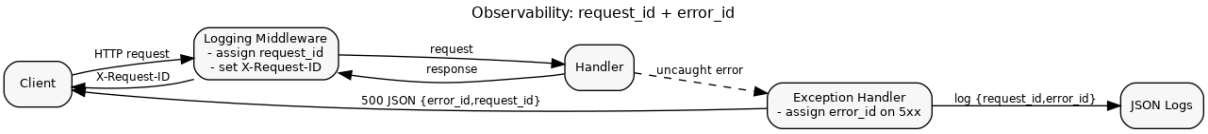
Figure: Sequence: Container Startup + Readiness



7.2 Observability: Request and Error Correlation

Every request receives a request_id (from X-Request-ID if present, otherwise generated). Server errors (5xx) include an error_id. Both IDs are logged, enabling fast correlation between client reports and server logs.

Figure: Observability: request_id + error_id



8. Environment Configuration

The system supports multiple environments via environment-specific .env templates. An ENVIRONMENT selector chooses the appropriate configuration file, while start scripts can create .env from templates and generate a secure SECRET_KEY.

9. Testing Strategy

The test suite includes API integration tests for endpoint behavior and authorization, plus unit tests for business logic. Tests run deterministically by injecting safe settings such as SECRET_KEY within the test harness.