🏦 Mini-Project: Building a Resilient Banking API

Goal: Implement a production-ready "Account Balance" service

This hands-on exercise will guide you through applying modern backend principles to a simple FastAPI application for a Personal Banking Management System.

Topics Covered:

Environment Profiles
dev, stage, prod configurations

Configuration & Secrets
Secure credential management

Database & Caching
Connection pooling and Redis caching

Logging & Deployment
Structured logging and Docker containers

🎯 Project Requirements

You are tasked with building a single, crucial API endpoint: GET /balance/{user_id}

Configurable

It should connect to different databases and services depending on the environment (dev/stage/prod).

Performant

It must respond quickly by caching frequently accessed account balances.

Secure

Database credentials must be handled as secrets, not hardcoded.

Observable

All requests should be logged in a structured format.

Deployable

It must be packaged in a Docker container for consistent deployment.

Implementation Time: 30-45 minutes

Step 1 - Project Setup & Dependencies

Let's lay the foundation for our project

Create the project directory:

```
BASH

mkdir banking_api
cd banking_api
python -m venv venv
source venv/bin/activate
```

Create the main application file main.py and a settings file config.py.

Step 1 - Project Setup & Dependencies

Define dependencies in requirements.txt:

```
REQUIREMENTS

fastapi
uvicorn[standard]
pydantic-settings
redis
fastapi-cache2[redis]
structlog
```

Install the dependencies:

```
BASH

pip install -r requirements.txt
```

Step 2 - Profiles & Configuration

Using Pydantic-Settings to manage configuration from environment variables

Environment Files:

```
.ENV.DEV

APP_ENV=development
DATABASE_URL="sqlite:///./dev_balance.db"
REDIS_URL="redis://localhost:6379/0"
LOG_LEVEL="INFO"
```

```
.ENV.PROD

APP_ENV=production
DATABASE_URL="postgresql://user:${DB_PASSWORD}@prod-db:5432/banking"
REDIS_URL="redis://prod-redis:6379/1"
LOG_LEVEL="WARNING"
```

Key Idea:
The `APP_ENV` environment variable acts as a switch to load the correct .env file.

Configuration Implementation (config.py)

```python
import os
from functools import lru_cache
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    APP_ENV: str = "development"
    DATABASE_URL: str
    REDIS_URL: str
    LOG_LEVEL: str

    # Load the .env file corresponding to APP_ENV
    model_config = SettingsConfigDict(
        env_file=f".env.{os.getenv('APP_ENV', 'development')}"
    )

@lru_cache() # Cache the settings object
def get_settings() -> Settings:
    return Settings()

settings = get_settings()
```

This configuration allows us to switch contexts easily between development, staging, and production environments.

Step 3 - Main Application & Caching

Building the FastAPI endpoint with Redis caching

PYTHON (MAIN.PY)

```python
from fastapi import FastAPI, Depends
from fastapi_cache import FastAPICache
from fastapi_cache.backends.redis import RedisBackend
from fastapi_cache.decorator import cache
import structlog

app = FastAPI()
log = structlog.get_logger()

@app.get("/balance/{user_id}")
@cache(expire=30) # Cache for 30 seconds
async def get_balance(user_id: int, settings: Settings = Depends(get_settings)):
    log.info("request_received", endpoint="/balance", user_id=user_id)
    return get_balance_from_db(user_id)

@app.on_event("startup")
async def startup():
    settings = get_settings()
    redis = aioredis.from_url(settings.REDIS_URL)
    FastAPICache.init(RedisBackend(redis), prefix="fastapi-cache")
```

Step 4 - Containerization & Secrets

Creating a portable Docker image

DOCKERFILE

```
FROM python:3.11-slim as builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/python3.11/site-packages
COPY . .

ENTRYPOINT ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Secrets Management:

- Secrets like `DB_PASSWORD` are injected at runtime
- Never hardcode credentials in code or Docker images
- Use secret managers (AWS Secrets Manager, Vault) in production

Running the Project

Docker Compose Setup:

```yaml
YAML (DOCKER-COMPOSE.YML)

services:
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  app:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - redis
    environment:
      - APP_ENV=${APP_ENV}
```

Development Mode:

```bash
BASH

export APP_ENV=development
docker-compose up --build
```

Production Mode:

```bash
BASH

export APP_ENV=prod
export DB_PASSWORD='my-super-secret-prod-password'
docker-compose up
```

🧠 Conclusion & Learnings

What we achieved in under 45 minutes

Profile Management

Single codebase, multiple environments with .env files

Secrets Security

Environment-based secret injection, no hardcoded credentials

Multi-level Caching

In-memory settings cache + Redis response cache

Database Best Practices

Connection strings as attached resources

Structured Observability

JSON logs for monitoring and debugging

Immutable Deployments

Docker containers eliminate environment inconsistencies

🚀 Production-Ready Service Complete!