Deployment Guide

Author: Karan Mehta

Date: 2026-02-25

Table of Contents

Deployment Guide

This document supplies an in-depth operational guide, architectural explanations, and manual deployment instructions for the Banking REST Service.

Prerequisites

To run this application, ensure the following tools and versions are installed on your host machine:

- **Docker:** Version 20.10.0 or higher.
- **Docker Compose:** Version 2.0.0+ (or the `docker-compose` standalone version 1.29.2).

Architecture Overview

This project uses a containerized, decoupled microservices topology:

- **FastAPI Backend (`api`):** The primary server utilizing asynchronous Python 3.9+, integrating SQLAlchemy 2.0.
- **Nginx Frontend (`frontend`):** A multi-stage build creates the production-ready React/Vite asset bundle, delivered blazing-fast using an Alpine-based Nginx container with SPA fallback routing.
- **SQLite Volume:** Data relies on a self-contained asynchronous SQLite database, maintaining persistent state via volume mounts instead of an isolated database connection container.

Manual Deployment

To launch the operational stack cleanly without relying on the provided `start.sh` or `start.ps1` shell scripts:

1. **Build and Background the Services:**

Run the following from the root directory to initiate both containers:

```
docker-compose build
docker-compose up -d
```

Code: bash

2. **Execute Database Seeding/Migrations Manually:**

Since a manual boot does not auto-populate synthetic test data, execute the seeder script independently inside the running API core:

```
docker-compose exec api python seed_data.py
```

Code: bash

Network Topology

Understanding the bridging topology between logical services and your host OS is useful for overriding port configurations and preventing clashes:

The Nginx web server binds internally to port 80. The `docker-compose.yml` natively proxies this traffic to `8080` on the host side for dashboard access.

- **Host 8080 Process -> Container 80:**
- **Host 8000 Process -> Container 8000:**

The Python Uvicorn worker runs internally on `8000`, symmetrically exposed on the host to allow local API interactions.

Data Persistence

Because the banking application utilizes an embedded local database engine (SQLite), we persist the `.db` layer by creating a bounded Docker volume mapping:

- **Persistence Mapping:** Local modifications within the `/data` folder update the host map dynamically.
- **Truncating the Ledger:** To fully reset data to its baseline (destroying all test users, transactions, and session caches), forcefully detach and obliterate the assigned volume:

```
docker-compose down -v
```

Code: bash

Troubleshooting

The following operations can solve generalized container integration flaws:

- **"Port 8080 / Port 8000 is Already in Use"**

If localized background processes claim these ports, override the defaults utilizing a `.env` file dropped exactly at the project root structure:

```
API_PORT=8001
FRONTEND_PORT=8081
```

Code: env

(Remember to refresh the Docker instances utilizing `docker-compose up -d` after updates).

- **Viewing Real-Time Metrics/Logs:**

To monitor raw HTTP connections mapping to either container block:

```
docker-compose logs -f api
# OR
docker-compose logs -f frontend
```

Code: bash

- **Force Rebuilds after Major Updates:**

Should dependency packages shift critically, cache layers must be busted:

```
docker-compose up -d --build
```

Code: bash