Project Protocol: Media Ratings Platform (MRP)

Author: Peyman Aparviz

1. App Design

Design Decisions

- **Architecture**: A **Layered Architecture** (`Handler` -> `Service` -> `Repository` -> `Database`) was chosen to enforce **Separation of Concerns** (SoC) and adhere to SOLID principles. This allows for easier testing and maintenance.
- **No Frameworks**: Adhering to strict project constraints, no heavy frameworks (like Spring Boot or Hibernate) were used.
    - **HTTP Server**: `com.sun.net.httpserver.HttpServer` is used for handling REST requests.
    - **Persistence**: **Pure JDBC** is used for database interactions to demonstrate understanding of low-level SQL handling.
- **Database & Security**:
    - **PostgreSQL**: Used as the relational database.
    - **SQL Injection Prevention**: All SQL queries use `PreparedStatement` to sanitise inputs.
    - **Singleton Pattern**: `DatabaseManager` is a singleton to manage a central database connection (simulating a simple connection pool/manager).
- **Data Transfer Objects (DTOs)**: `UserDTO` and `MediaDTO` are used to decouple the API layer (JSON structure) from the Model/Database layer. This prevents sensitive data (like password hashes) from accidentally leaking and allows the API schema to evolve independently of the DB schema.
- **Authentication**:
    - **Token-Based Auth**: Custom implementation using persistent tokens stored in the database.
    - **Format**: `mrp-token-<UUID>` (excluding PII like username from the token string for better security).

    - **Password Hashing**: SHA-256 is used for hashing passwords.

- **Builder Pattern**: Implemented in Models (`User` , `Media` , `Rating`) to handle complex object construction with many optional fields cleanly.

- **Resource Routers & Granular Handlers**:

    - Initially, monolithic handlers (`UserHandler` , `MediaHandler`) handled all endpoints for a resource. This grew unwieldy.
    - **Refactoring**: Adopted a pattern where **Routers** (`UserRouter`) dispatch requests to **Granular Handlers** (e.g., `LoginHandler` , `CreateMediaHandler`).
    - **Benefit**: Each handler class has a single responsibility (SRP), making the code cleaner and easier to navigate.

Project Structure

The application is structured into the following packages:

- `at.fhtw.swen1.mrp.data` : Infrastructure layer handling database connections and creation.
- `at.fhtw.swen1.mrp.dto` : Plain Old Java Objects (POJOs) for JSON serialization/deserialization.
- `at.fhtw.swen1.mrp.handler` : The Controller layer. Contains **Routers** (dispatchers) and **Granular Handlers** (endpoint logic).
- `at.fhtw.swen1.mrp.model` : The Domain layer. Represents the core business entities.
- `at.fhtw.swen1.mrp.repository` : The Persistence layer. Contains all SQL logic and conversion from `ResultSet` to Models.
- `at.fhtw.swen1.mrp.service` : The Business Logic layer. Orchestrates business rules.

2. Unit Testing Strategy

I implemented a comprehensive testing strategy to ensure the reliability of the application, covering both individual components and the system as a whole.

Unit Tests (JUnit 5 + Mockito)

For the business logic in the `Service` layer, I used **JUnit 5** and **Mockito**. This allowed me to test the logic in isolation without requiring a running database.

- **Mocking Repositories**: By mocking `UserRepository` or `MediaRepository` , I could simulate various database states (e.g., finding a user, returning an empty list) and verify how the

Service handles them.

- **Coverage**: I implemented over **30 Unit Tests** covering:
  - `UserService` : Registration (hashing), Login (token generation), Profile retrieval.
  - `MediaService` : Creation (ownership), Filtering logic.
  - `RatingService` : Validation (1-5 stars), Prevention of duplicate ratings, Average rating calculation.

Integration Tests (Rest-Assured)

For the final verification, I moved away from fragile shell scripts and implemented a robust Java-based Integration Test Suite (`MRPIntegrationTest.java`) using **Rest-Assured**, which includes **16 test cases**.

- **Setup**: The test class starts the `HttpServer` programmatically on a separate port (`8082`) and resets the database before the suite runs.
- **Scope**: It tests the full stack (HTTP -> Handler -> Service -> Repository -> DB) by simulating real HTTP requests (POST, GET, etc.) and asserting on the JSON responses and status codes.
- **Scenarios**: It covers complex flows like "User Registers -> Creates Media -> Another User Rates it -> Leaderboard Updates".

3. SOLID Principles

I consciously applied SOLID principles throughout the development. Here are two concrete examples:

Single Responsibility Principle (SRP)

I strictly separated the concerns into different layers:

- **Granular Handlers** (e.g., `LoginHandler`) are *only* responsible for parsing a specific HTTP request and sending its response.
- **Routers** (e.g., `UserRouter`) are *only* responsible for dispatching requests to the correct handler.
- **Services** (e.g., `UserService`) contain the actual business rules (hashing passwords, verifying duplicates). They don't know about HTTP or SQL.
- **Repositories** (e.g., `UserRepository`) are solely responsible for SQL execution and mapping `ResultSet` rows to Objects.
  This separation made the code much easier to read and allowed me to unit test the Services independently.

Dependency Injection (DIP/IOC)

Instead of classes instantiating their dependencies internally (e.g., `UserService` creating a `new UserRepository()`), I injected them via the constructor.

- **Implementation**: In `Main.java`, I manually wire everything together:

```
UserRepository userRepo = new UserRepository();
UserService userService = new UserService(userRepo); // Injection

// Handlers are injected with services
LoginHandler loginHandler = new LoginHandler(userService, objectMappe

// Routers are injected with specific handlers
UserRouter userRouter = new UserRouter(loginHandler, ...);
```

- **Benefit**: This was critical for testing. In `UserServiceTest`, I could inject a **mock** `UserRepository` into the `UserService` instead of the real one. This adheres to the Dependency Inversion Principle, as the high-level module (`UserService`) depends on an abstraction (the injected instance contract) rather than a concrete internal instantiation.

4. Lessons Learned

During this project, I encountered several challenges that deepened my understanding of backend development:

A. Manual Dependency Injection vs. Frameworks
Coming from a world where frameworks often do the heavy lifting, manually wiring all dependencies in `Main.java` was an eye-opener. It forced me to understand exactly which component depends on which. While it became a bit verbose as the application grew (adding `RecommendationService`, `LeaderboardHandler`, etc.), it gave me full control and understanding of the object graph. Ideally, for larger projects, I would appreciate a DI container, but doing it manually was a valuable educational exercise.

B. DTOs vs. Entities
At first, creating separate `UserDTO` classes alongside the `User` model felt like redundant work. However, closer to the end, I realized their necessity. They acted as a security buffer—ensuring I never accidentally serialized specific fields (like the internal ID or password hash) back to the client. They also allowed me to handle JSON-specific validation (like ensuring a password is provided on register) without polluting the database model entity.

C. Returning IDs from Database Inserts
Initially, my `repository.save()` methods were `void`. This caused major headaches during integration testing because I couldn't get the ID of a just-created user or media item to use in the next

step (like rating that media). I learned to use JDBC's `Statement.RETURN_GENERATED_KEYS`. Refactoring the repositories to return the generated IDs allowed my integration tests to be dynamic and robust, rather than relying on hardcoded IDs or guessing.

D. Inheritance vs. Composition for Shared Utility

During the refactoring phase, I introduced a `BaseHandler` and `BaseRouter` to share common logic like helper methods for sending JSON responses (`sendResponse`, `sendError`). While Composition (injecting a `ResponseHelper`) is often preferred, I found that **Inheritance** was practical here because `BaseHandler` provides a stable contract (`authenticate`, `readBody`) that every single handler naturally needs. It reduced code duplication significantly without creating complex object graphs.

5. Tracked Time

Estimated time spent on the project phases:

Phase	Description	Time (Hours)
Phase 1	Setup, Docker, Architecture, Basic CRUD	8 h
Phase 2	Ratings, Logic, Engagement (Favorites)	8 h
Phase 3	Advanced Features (Filtering, Recommendations, Leaderboard)	6 h
Phase 4	Refactoring, Unit Tests, Integration Tests, Debugging	9 h
Phase 5	Architecture Refactoring (Routers, Granular Handlers)	7 h
Total		**38 h**

6. Link to Repository

https://github.com/Peym4n/swen1-mrp