Monster Trading Card Game

Final submission protocol - Sofia-Hanna Ginalis (if23b070)

App Design

The overall application is structured around a simple custom HTTP server implemented in Java. The server listens on port **10001** and accepts incoming socket connections. For concurrency, it relies on a fixed thread pool (using Java's ExecuterService) so multiple client requests can be processed in parallel.

Requests received by the server are passed to a Router component that inspects the request path (e.g /users, /packages, /tradings, /battles)and dispatches them to the appropriate controller class. Each Controller (UserController, PackageController, TradingController, BattleController) handles the request handling by parsing inputs, calling the corresponding service, and then sending back the correct response.

From there, each Service implements the business logic for its particular domain:

- **UserService** handles user registration, authentication, coin balance changes, and user stats
- **PackageService** manages package creation and acquisition logic
- **TradingService** deals with creating and deleting trading deals, as well as executing trades
- **BattleService** covers the battle logic, from searching for opponents to simulating rounds and assigning ELO changes

Below the service layer are Repository classes that carry out direct SQL operations on the database via JDBC. To store data persistently, the application relies on the PostgreSQL database. The four main tables are: users, cards, packages and trading_deals. All queries use prepared statements to prevent injection attacks.

Lessons Learned

One crucial lesson was the importance of carefully organizing concurrency in order to be mindful of potential race conditions or shared-state issues.

I also learned that managing SQL statements by hand requires consistent use of prepared statements and thorough error-handling logic (SQL exceptions, rollbacks, etc.). Initially, it was a bit tedious, but it gave a lot of control and transparency.

During testing, I struggled to decide when to rely on integration tests (spinning up the actual server, or at least the real database) versus pure unit tests with mocks. Eventually, I separated tests into two categories: integration tests that test the entire flow (controllers + DB) and unit tests that isolate logic via Mockito. This approach helped maintain clarity and test coverage.

Unit Testing Decisions

For my unit tests, I tried to maintain a strict separation from the real database. I used Mocikto to mock repository methods so that the services could be tested in isolation. This allowed me to validate the business logic (e.g., whether a user deck has at least four cards, whether a trade is locked, or if a user has enough coins) without dealing with SQL or setup overhead.

Where appropriate, I tested edge cases, like incomplete decks, insufficient coins, duplicate deals, or trades attempted by the same user who created the deal. I also covered normal success scenarios, ensuring that each service method behaves as expected.

In total, I now have 22 pure unit tests, supplemented by a handful of integration tests that validate the end-to-end workflow (from an HTTP request to final DB changes).

Unique Feature

I decided to introduce a small reward system for players who engage heavily in battles. Specifically, every time a user's total wins reaches a multiple of 5 (5, 10, 15, ...), they earn 10 bonus coins. This logic is handled in BattleService during ELO and stats updates. If a user's updated win count mod 5 == 0, it increments their coin balance by 10. I felt this was a simple yet fun mechanic that incentivizes ongoing battles and gives players a small "victory bonus."

Time Tracking

- Learning about postgreSQL: 2h
- Initial Setup and Database Schema: 13h
- Learning about threading: 3h
- Service Layer: 12h
- Core Feature Implementation: 20h
- Integration and Testing: 10h
- Unique Feature: 30min
- Debugging and Refactoring: 20h
- Documentation: 2h