Project Documentation

Software Engineering 1
Winter Semester 2023/24

Project: **Monster Trading Cards Game**

Peter Tavaszi
if22b227

Design:

Overall Architecture:

- The application follows a modular structure, with separate components responsible for different concerns, such as user management, card management, trading deals, and battles.
- The architecture separates concerns between presentation logic (controllers), data access logic (repositories and DAOs), and data models (DTOs and models).

Components:

a.) App:

- The App class is responsible for handling HTTP requests coming from the client
- A major responsibility of the App is to authenticate incoming user requests based on the usertoken provided in the request header
- It filters the incoming requests based on pathnames and delegates them to the appropriate controller classes corresponding to their routes

a.) Controllers:

- Controllers handle incoming HTTP requests, route them to appropriate endpoints (repositories), and orchestrate the interaction between the client and the server.
- Each controller is responsible for a specific domain or resource, such as users, cards, trading deals, or battles.
- They delegate business logic to the repository classes and handle data transformation between DTOs and domain models.

b.) Repositories:

- Repositories provide an abstraction layer for interacting with the database.
- They encapsulate CRUD operations for domain entities and translate them into database queries.
- Each repository is typically associated with a specific domain entity or aggregate root and is responsible for data persistence and retrieval.

c.) Data Access Objects (DAOs):

- DAOs directly interact with the database to execute SQL queries and retrieve or update data.
- They encapsulate database-specific operations, such as connection management, prepared statement execution, and result set processing.
- DAOs abstract away the details of database interactions from higher-level components like repositories and controllers.

e.) Data Transfer Objects (DTOs):

- DTOs are lightweight objects used for transferring data between different layers of the application.
- They represent structured data exchanged between the client and server or between layers of the application.
- DTOs help decouple the internal representation of data from its external representation.

Interactions:

- Controllers receive incoming HTTP requests and extract the relevant data from request parameters or bodies.
- They invoke corresponding methods on repository classes, passing input parameters and receiving results.
- Repositories and DAOs handle data persistence and retrieval, executing database queries and mapping results to domain objects or DTOs.
- Communication between components follows a layered architecture, with each layer responsible for specific tasks and dependencies flowing in a downward direction.

Lessons Learned:

1.) Initially, I began developing the application solely based on the textual project description, which later proved to be a significant mistake. I did not think about looking at the API specification or the curl script for additional details. This way, I wasted a lot of time with planning and building an application, which did not correspond to the actual criteria.
2.) It was definitely helpful that we learned about the general architecture of the project during the semester. The example code snippets, which was provided to us by our lecturer helped a lot in understanding the expected structure of the project. I learned that DAOs and Repositories are a good way to implement client-server architectures.
3.) I learned about technologies and concepts, that were completely new for me, such as docker, curl scripts, java multithreading and unit testing (with mocks, stubs and spies), as well as how to create a sequence diagram for planning more complex ideas such as a battle logic.
4.) Developing an application based on an API specification and curl script for the first time was an intriguing experience. Once I learned how to utilize and interpret them effectively, I could rely on them to incrementally build the application, implementing various user-related functionalities, card management, trade deals, and battles.
5.) Another important take-away for me also from this project was that it is better to plan a project thoroughly before starting to implement it. This planning phase should include at least a class diagram in case of an object-oriented architecture as well as sequence-diagrams for the most complex methods and wokr-flows in the application.
6.) I found it very interesting to witness how a request and a repsonse object is built up piece by piece at each user interaction with the server. It was definitely worth spending time with understanding how these objects are structured and how they are sent back and forth between the server and the client.

7.) It was also an interesting experience to learn how to build up a PostgreSQL database in a Docker container. I was already familiar with the concepts of SQL but the way we had to implement the new database was completely new and in my opinion very useful to learn. It surprised me how straightforward and easy-to-use Docker actually is, and how it can be used in so many different ways.
8.) Another major takeaway from this project was, how useful it is to write unit tests for the application. I was able to figure out mistakes I made during the implementation that were not very evident for me and would have probably gone unnoticed without checking the functionality of my app with the JUnit tests.
9.) I struggled a bit while creating the in-memory database, which was needed for the JUnit tests, but gained a lot of insight into how the process of building up such databases works.

Unit Testing Decisions:

Class: UserDAOTest

Function: testCreateUser_Success()

Decision: This test verifies that users can be created successfully in the database. By calling the createUser() method with valid username and password parameters, and then checking that the returned status code is 201 (indicating successful creation), this test confirms that the user creation process works as expected.

Function: testCreateUser_Conflict()

Decision: Testing for user creation conflicts ensures that the method handles such cases correctly. By first creating a user and then attempting to create the same user again, this test verifies that the method returns the appropriate status code (409 - Conflict) when attempting to create a user that already exists.

Function: testGetUser_Success()

Decision: This test ensures that user data can be retrieved successfully from the database. By creating a user, retrieving their data using the getUser() method, and verifying that the returned UserDataDTO object is not null and contains the expected default values (e.g., null name, bio, image), this test confirms the correctness of the getUser() method.

Function: testUpdateUser_Success()

Decision: Testing the ability to update user information ensures that the method modifies user data correctly in the database. By creating a user, updating their information using the updateUser() method, and then retrieving their updated data to verify that the changes were applied correctly (e.g., updated name, bio, image), this test confirms the correctness of the updateUser() method. Additionally, by checking that the method returns the expected status code (200 - OK), this test ensures that the update operation was successful.

Function: testLoginUser_Success()

Decision: This test ensures that users can log in successfully with correct credentials. By creating a user, attempting to log in with the correct username and password, and verifying that a valid authentication token is returned, this test confirms that the login process works as expected. Additionally, testing for scenarios such as non-existent users or incorrect passwords helps ensure that the method handles authentication failures appropriately and returns the correct status codes.

Function: testLoginUser_Failure_UserNotFound()

Decision: Testing for the scenario where a user attempts to log in with a non-existent username ensures that the method handles such cases correctly. By attempting to log in with a non-existent username and verifying that the method returns the appropriate status code (404 - Not Found), this test confirms that the method behaves as expected when users are not found in the database.

Function: testLoginUser_Failure_AuthenticationFailed()

Decision: This test verifies that the login process fails when users provide incorrect passwords. By creating a user with a known password, attempting to log in with the correct username but an incorrect password, and verifying that the method returns the appropriate status code (401 - Unauthorized), this test ensures that the method handles authentication failures due to incorrect passwords correctly.

Function: testDeleteUser_Success()

Decision: Testing the deleteUser() method ensures that users can be successfully deleted from the database. By creating a user, deleting the user, and attempting to retrieve their data to verify that it is null, this test confirms that the user deletion process works as expected. Additionally, testing for scenarios such as deleting non-existent users ensures that the method handles such cases gracefully without errors.

Function: testGetStats_Success()

Decision: This test ensures that user statistics can be retrieved successfully from the database. By creating a user, retrieving their statistics, and verifying that the returned data matches the expected values (e.g., default Elo score, wins, losses), this test confirms the correctness of the getStats() method. Additionally, testing for scenarios such as retrieving statistics for non-existent users helps ensure that the method behaves as expected under various conditions.

Function: testGetStats_NonExistentUser()

Decision: Testing the getStats() method with a non-existent user ensures that the method handles such cases correctly. By attempting to retrieve statistics for a user that does not exist and verifying that the returned data is null, this test confirms that the method behaves as expected when users are not found in the database.

Function: testGetScoreboard_Success()

Decision: This test verifies that the scoreboard can be retrieved successfully from the database. By creating multiple users with different Elo scores, retrieving the scoreboard, and verifying that the returned data matches the expected values (e.g., user names, Elo scores, wins, losses), this test confirms the correctness of the getScoreboard() method. Additionally, testing for scenarios such as an empty scoreboard ensures that the method handles such cases gracefully without errors.

Function: testGetScoreboard_EmptyScoreboard()

Decision: Testing the getScoreboard() method when no users are created ensures that the method handles empty scoreboard scenarios correctly. By retrieving the scoreboard and verifying that it is not null and empty, this test confirms that the method behaves as expected when there are no users in the database.

Class: CardDAOTest

Function: createPackage_Success()

Decision: Tests the successful creation of a package of cards. It arranges a list of five valid cards, acts by attempting to create a package with those cards using the cardDAO, and asserts that the status code returned is 201, indicating success.

Function: createPackage_Failure_LessThan5Cards()

Decision: Tests the failure scenario when attempting to create a package with fewer than five cards. It arranges a list of only three valid cards, acts by attempting to create a package with those cards using the cardDAO, and asserts that the status code returned is 400, indicating a bad request.

Function: createPackage_Failure_DuplicateCardIds()

Decision: Tests the failure scenario when attempting to create a package with duplicate card IDs. It arranges a list of five cards with some having duplicate IDs, acts by attempting to create a package with those cards using the cardDAO, and asserts that the status code returned is 400, indicating a bad request.

Function: createPackage_Failure_CardsAlreadyInOtherPackages()

Decision: Tests the failure scenario when attempting to create a package with cards that are already in other packages. It first creates a package with a set of cards, then attempts to create another package with the same set of cards. The test asserts that the status code returned is 409, indicating a conflict.

Function: buyPackage_Success()

Decision: Tests the successful purchase of a package of cards by a user. It creates a user, creates a package of cards, and then simulates the purchase of that package by the user. The test asserts that the purchased cards match the expected cards and that the user now owns the cards from the package.

Function: buyPackage_Failure_NotEnoughCoins()

Decision: Tests the failure scenario when a user attempts to buy a package without having enough coins. It creates a user and simulates the purchase of multiple packages by that user, exceeding the available coins. The test asserts that the purchase fails due to insufficient funds.

Function: buyPackage_Failure_PackageNotFound()

Decision: Tests the failure scenario when a user attempts to buy a non-existent package. It creates a user and simulates the attempted purchase of a non-existent package by that user. The test asserts that the purchase fails due to the package not being found.

Function: updateDeck_Failure_LessThan4UniqueCards()

Decision: Tests the failure scenario when a user attempts to update their deck with fewer than four unique cards. It creates a user, purchases packages, and then attempts to update the user's deck with fewer than four unique cards. The test asserts that the update fails.

Function: updateDeck_Failure_CardsNotInUserStack()

Decision: Tests the failure scenario when a user attempts to update their deck with cards not in their possession. It creates a user, purchases a package, and then attempts to update the user's deck with cards from another package. The test asserts that the update fails.

Function: getUserDeck_Failure_UnconfiguredDeck()

Decision: Tests the scenario where a user's deck has not been configured. It creates a user and purchases packages but does not configure the user's deck. The test asserts that the user's deck is empty.

Function: getUserDeck_Success_ConfiguredDeck()

Decision: Tests the scenario where a user's deck has been configured. It creates a user, purchases packages, configures the user's deck, and then retrieves the user's deck. The test asserts that the user's deck matches the configured cards.

In essence, my focus was on testing the methods of the UserDAO and CardDAO classes, as they play crucial roles in database interaction and the manipulation of users and cards, which are fundamental to the game logic. To facilitate these tests, I constructed an in-memory database, which allowed me to populate it with dummy data without risking any harm to the actual application database. Creating the in-memory database turned out to be a bit more challenging than I initially expected, but I could solve all associated problems and gained valuable insight into the process of building such databases from scratch.

Unique Feature:

Chaos Rounds:
I introduced a unique feature chaos rounds to add unpredictability and excitement to battles. Chaos rounds occur on the 25th, 50th, and 75th rounds of a battle, injecting an element of randomness into the outcome by disregarding the usual card-based damage calculation.

During chaos rounds, instead of relying on the predefined damage values of the user cards, the effective damage inflicted by each user is determined randomly. This randomness adds a twist to the battle dynamics, by introducing moments of uncertainty and surprise.

Logout Functionality:
I have also introduced a logout functionality in the application, which allows users to terminate their current session by removing the authentication token associated with their username. When a user logs out, their authentication token is invalidated, preventing unauthorized access to their account until they log in again.

When we invoke the logoutUser method with the user's username as a parameter, the application checks if the user exists in the system. If the user exists, the method proceeds to remove the authentication token associated with the user from the database, effectively logging them out. If the token removal is successful, the method returns an HTTP status code "200" to indicate successful logout. In case of any failure during the token removal process or if an SQL exception occurs, the method returns an HTTP status code "500" to signify an internal server error.

Tracked Time:

- Planning which classes will be needed, creating the initial PlantUML class diagram ~2h
- Creating the initial classes (models), and setting up the Controllers/DAOs/Repositories architecture ~10h
- Setting up Docker and the Docker container for the PostgreSQL database ~2h
- Planning the tables (create table statements) and creating the PostgreSQL database ~5h
- Creating routes in the App class for user-related requests ~5h
- Creating UserController/UserService (single threaded server)/UserRepository ~5h
- Created DatabaseService class for connecting to the database ~1h
- Creating UserDAO ~15h
- Making the Server multithreaded by creating the Task classes, which implement the Runnable interface ~3h

- Adjusting the initial form of the Server/HttpStatus and Task classes to fit the rest of the application ~10h
- Implemented token-based user login/authorization functionality, adjusted the Request/Response classes ~5h
- Creating routes in the App class for card-related requests ~5h
- Creating CardController/CardRepository ~5h
- Creating CardDAO ~15h
- Creating routes in the App class for user statistics-related requests ~2h
- Updating the UserDAO class for interacting with the database for retrieving and updating statistics ~3h
- Creating routes in the App class for tradedeal-related requests ~3h
- Creating TradeDealController/TradeDealRepository ~3h
- Creating TradeDealDAO ~10h
- Creating the PlantUML sequence diagram for the battle logic ~6h
- Creating routes in the App class for game-related requests, implemented that user requests are put into a queue and wait for each other to start the battle ~4h
- Creating GameController/GameRepository ~2h
- Creating GameDAO ~15h
- Added JavaDoc-style comments to the methods in the DAO classes ~5h
- Creating JUnit tests for the UserDAO and the CardDAO classes ~6h
- Creating the unique features of the application ~2h
- Total: 149h

Link to Git Repository:
https://github.com/gregory567/Monster-Trading-Cards-Game-MTCG.git