# Controllers

* **Purpose**: Handle incoming HTTP requests and map them to appropriate actions.
* **Example**:
  + **UserController**: Handles user-related actions like registration, authentication, and profile updates.
  + **BattleController**: Manages battle-related operations, such as initiating battles, retrieving battle logs, and resolving outcomes.

# Repositories

* **Purpose**: Directly interact with the database, providing CRUD operations and data persistence.
* **Example**:
  + **PostgreSqlBattleRepository**: Manages battle logs, ELO updates, card leveling, and user statistics (e.g., wins, losses, games played).
  + **PostgreSqlUserRepository**: Handles user-related database operations like retrieving user information, updating ELO, and managing decks.

# HTTP Server

* **Purpose**: Acts as the entry point for the application, routing HTTP requests to controllers.
* **Example**:
  + Routes requests such as POST /battle/start or GET /user/stats to their respective controllers.
  + Encodes/decodes JSON for requests and responses using a standardized format.

# Interaction Flow Example

1. **Request**: A user initiates a battle (POST /battle/start).
2. **Controller**: The BattleController processes the request and validates input.
3. **Battle Class** The Battle Class performs the battle logic (damage calculation, special rules, etc.).
4. **Repository**: The PostgreSqlBattleRepository saves the battle log, updates ELO, levels up cards, and increments statistics (wins/losses/games played).
5. **Response**: The result is returned to the HTTP server, which responds to the user.

# Database Schema

Ein Bild, das Text, Screenshot, Diagramm enthält.

Automatisch generierte Beschreibung

# Unique Feature

The Card Leveling system rewards the winning player by increasing the damage of all their cards by 5 after a battle. This progression mechanic motivates players to engage in battles and strengthens their decks over time, with updates saved permanently in the database.

# Unit Tests

**BattleTests**

1. **TestEffectiveness\_FireAgainstWater\_IsNotVeryEffective**: To verify that elemental effectiveness rules are correctly implemented for battles, ensuring balanced gameplay mechanics.
2. **TestEffectiveness\_WaterAgainstFire\_IsSuperEffective**: To ensure the inverse of the previous test is also functioning, confirming that water's advantage over fire is accurately reflected in damage calculation and logs.
3. **TestBattleLog\_SavesLogToDatabase**: To check that the battle log is properly saved and includes user-specific information for debugging and player experience.
4. **TestBattleEndsInDraw\_WhenDecksAreEqual**: To validate the outcome of battles where both players have equal damage, ensuring fairness and correct handling of edge cases.
5. **TestEloChangesAfterBattle**: To confirm the Elo rating system updates accurately after battles, which is critical for matchmaking and progression.

**ControllerTests**

1. **HandleBattle\_ShouldReturn400\_WhenSessionTokenIsMissing**: To ensure that the server properly handles missing authentication tokens, maintaining security and expected behavior.
2. **HandleBattle\_ShouldReturn401\_WhenSessionTokenIsInvalid**: To test the server's response to invalid tokens, ensuring unauthorized access is prevented.

**DatabaseTests**

1. **AddUser\_ShouldReturnFalse\_WhenUserNotAdded**: To validate proper error handling when the database fails to add a user, ensuring robust functionality.
2. **ValidateUser\_ShouldReturnTrue\_WhenUserExists**: To confirm that the system correctly authenticates valid user credentials, which is essential for login functionality.
3. **ValidateUser\_ShouldReturnFalse\_WhenUserDoesNotExist**: To ensure invalid or nonexistent users cannot authenticate, protecting user accounts and system integrity.
4. **IsUsernameAvailable\_ShouldReturnTrue\_WhenUsernameIsAvailable**: To verify that the system can check username availability, improving the user registration experience.
5. **GenerateRandomCard\_ShouldReturnCardOfTypeMonsterOrSpell**: To confirm that random card generation produces valid, game-usable cards tied to specific users, ensuring variety and fairness in gameplay.
6. **GetCardById\_ShouldReturnCard\_WhenCardExists**: To validate the ability to retrieve specific cards, ensuring the database can provide essential gameplay data.
7. **CreateDeck\_ShouldReturnNewDeckId**: To confirm that new decks are properly created with unique IDs, supporting key gameplay features like deck management.

**HTTPServerTests**

1. **TestRequestParsing\_ValidRequest**: To ensure the server correctly interprets valid HTTP requests, a fundamental aspect of the server's operation.
2. **TestRequestParsing\_InvalidRequest**: To verify graceful handling of malformed requests, ensuring the server remains stable under invalid input.
3. **TestServerHandlesLoginRoute**: To confirm that the server processes login requests properly, returning the expected response and authentication token.
4. **TestServerHandlesInvalidRoute**: To ensure the server correctly handles unsupported routes, maintaining expected behavior and providing clear feedback to the client.
5. **TestInvalidMethodHandling**: To verify that unsupported HTTP methods are handled appropriately, reinforcing the API's integrity.
6. **TestServerHandlesScoreboardRoute**: To ensure the scoreboard endpoint returns accurate and meaningful data.

# Lessons learned

I realized just before the deadline that I was sending the token in the body of the request instead of the header, which is the standard and more efficient method. Unfortunately, I didn’t have enough time to change it before the deadline, so I had to proceed with the current implementation.

# Link to Git:

<https://github.com/FireWizard160/MonsterCardTradingGame>

# Tracked Time:

~ 40 hours

# Class Diagram

Ein Bild, das Diagramm, Plan, Schaltung enthält.

Automatisch generierte Beschreibung