Projektprotokoll: Monster Trading Cards Game (MCTG)

1. App-Design und Architektur

1.1 Entscheidungen zum Design

Die Entscheidungen zur Architektur des Projekts basieren auf dem Ziel, eine einfache, aber flexible REST-API für die Verwaltung von Monster Trading Cards zu erstellen. Wichtige Designaspekte sind:

- **Modularität**: Das Projekt wird in klare, voneinander getrennte Module unterteilt: Server, Datenbankzugriff, HTTP-Anfragen und -Antworten, und Nutzer-/Kartendatenmanagement.
- **Testbarkeit**: Alle Geschäftslogiken werden in separaten Klassen gekapselt, was eine einfache Unit-Tests-Abdeckung und zukünftige Erweiterungen ermöglicht.
- **Erweiterbarkeit**: Durch die Wahl eines datengesteuerten Ansatzes können neue Funktionen nachträglich hinzugefügt werden.

1.2 Klassen- und Datenbankstruktur

1.2.1 Server

- Der Server ist für das Handling von HTTP-Anfragen und -Antworten verantwortlich. Dabei werden alle Anfragen, die von Clients gesendet werden, interpretiert und durch passende Antworten wieder an den Client zurückgeschickt. Der Server arbeitet mit selbst

definierten Headern und Statuscodes, um ein vollständiges Protokoll für die Kommunikation zu gewährleisten.

- **Verantwortlichkeiten**:
    - Verarbeiten von HTTP-Anfragen und Rücksendung der Antwort an den Client.
    - Handhaben von verschiedenen HTTP-Statuscodes (z. B. 200 OK, 404 Not Found).
    - Setzen von Headern wie Content-Type und Content-Length, um sicherzustellen, dass die Antwort korrekt formatiert ist.
    - Multithreading multiple Anfragen

1.2.2 Datenmodell und Tabellenstruktur

Die Datenbank ist nach den Entitäten des Spiels strukturiert, wobei alle Tabellen Entitäten basierte Daten speichern. Dies erfolgt in einer relationalen PostgreSQL Datenbank, die sicherstellt, dass alle Beziehungen und Abfragen effizient verwaltet werden.

Tabelle game_user

- Speichert Benutzerinformationen. Jeder Benutzer hat eine eindeutige ID, einen Benutzernamen, ein Passwort und eine Reihe von anderen Attributen wie das Elo-Ranking, die Anzahl der gespielten Spiele, Coins und ein optionaler Token für die Authentifizierung.
- **Felder**:
    - id: Eine eindeutige ID (vom Typ UUID), die für jeden Benutzer erstellt wird.
    - username: Der Benutzername des Spielers, der eindeutig ist.
    - password: Das Passwort des Benutzers.
    - elo: Der Elo-Score des Spielers, ein Wert, der sich mit jedem Spiel ändern kann.
    - games_played: Zählt, wie viele Spiele der Benutzer gespielt hat.
    - coins: Ein Wertefeld, das anzeigt, wie viele Münzen der Benutzer hat.
    - bio: Ein Feld für zusätzliche, optionale Benutzerinformationen (z. B. persönliche Notizen).
    - image: String.
    - name: String und ist Profilname
    - token: Ein optionales Feld, das den Authentifizierungstoken des Benutzers speichern kann.

Tabelle game_package

- Diese Tabelle speichert Informationen über Kartenpakete, die zu bestimmten Zeiten erstellt wurden und für das Erhalten neuer Karten durch die Benutzer verantwortlich sind.
- **Felder**:
    - package_id: Eine eindeutige ID für jedes Paket.
    - created_at: Der Zeitstempel, wann das Paket erstellt wurde.

Tabelle game_card

- Speichert Informationen über jede einzelne Karte im Spiel. Diese Karten können Monster, Zaubersprüche oder andere Elemente des Spiels darstellen.

- **Felder**:
    - card_id: Eindeutige Karten-ID.
    - name: Der Name der Karte.
    - type: Der Kartentyp (z. B. "Monster", "Spell").
    - element_type: Das Element der Karte (z. B. "Fire", "Water", “Normal”).
    - damage: Der Schaden, den die Karte verursacht.
    - created_at: Der Zeitstempel der Kartenerstellung.

Tabelle package_cards

- Verknüpft Karten mit Paketen. Jede Zeile beschreibt, welche Karte zu welchem Paket gehört.
- **Felder**:
    - package_id: Fremdschlüssel zur game_package-Tabelle, die ein Paket beschreibt.
    - card_id: Fremdschlüssel zur game_card-Tabelle, die die Karte beschreibt.

Tabelle user_stack

- Verknüpft Benutzer mit Karten, die zu ihrem virtuellen Kartenstapel gehören. Jeder Benutzer kann einen einzigartigen Kartenstapel haben, der über diese Tabelle verwaltet wird.
- **Felder**:
    - user_id: Der Benutzer, der die Karte besitzt.
    - card_id: Die Karte, die im Benutzerstapel enthalten ist.

Tabelle package_transactions

- Dokumentiert Transaktionen, die zeigen, wann und welches Paket ein Benutzer erworben hat. Dies stellt sicher, dass die Beziehung zwischen Nutzern und ihren erstandenen Paketen nachvollziehbar bleibt.
- **Felder**:
    - transaction_id: Eine eindeutige ID für jede Transaktion.
    - user_id: Der Benutzer, der das Paket erworben hat.
    - package_id: Das Paket, das erworben wurde.
    - transaction_date: Das Datum und der Zeitstempel der Transaktion.

Tabelle user_deck

- Speichert das Deck eines Benutzers, indem es Verweise auf Karten enthält, die der Benutzer tatsächlich in seinen aktiven Stapel aufgenommen hat. Diese Karten sind für das Spielen des Spiels erforderlich.
- **Felder**:
    - user_id: Verweis auf den Benutzer.
    - card_id: Verweis auf die Karte, die Teil des Decks ist.

Klassenstruktur und Einordnung der Komponenten:

Im Projekt wurden die Klassen in die Kategorien `Database`, `Model` und `Server` unterteilt. Dabei wurde darauf geachtet, jede Klasse in eine geeignete Kategorie zuzuordnen, um eine saubere Trennung der Verantwortlichkeiten zu gewährleisten. Ziel war es, die Klassen so zu organisieren, dass sie logisch und funktional sinnvoll strukturiert sind. Alle wesentlichen Komponenten wurden durchdacht und in die entsprechenden Klassen eingeordnet, um die Wartbarkeit und Erweiterbarkeit des Projekts zu optimieren.

Im Server Ordner befinden sich die Klassen, die für den Server (**HttpServer**) und dessen Tätigkeiten zuständig sind. Dazu zählen Aufgaben wie das Verarbeiten von Requests (**HttpRequestParser**). Das Senden einer Response (**HttpResponseSender**) der mit der **HttpHeader** Klasse schöne responses zurückgibt. Der **RequestHandler** ist für das Weiterleiten zu den richtigen (**-Service**) Klassen zuständig. Außerdem befindet sich hier auch noch eine **TokenValidator** Klasse, die für die Authentifizierung zuständig ist.

Im Model Ordner befinden sich die ganzen Modelle der Anwendung (**Card, Deck, HttpRequest, Package, Player, Stack, User, BattleQueue**) in der **BattleQueue** befindet sich auch die Logic wie ein Battle aufgebaut ist, also hier läuft der Kampf ab.

Im Datenbank Folder befinden sich alle Klassen, die für Operationen mit der Datenbank zuständig sind und dadurch auch mit der Datenbank interagieren. Ich habe damit es übersichtlicher wird die Funktionen in verschiedene Klassen unterteilt, wo sich die Funktionen befinden, die einen bestimmten Fokus haben. Außerdem habe ich noch von einigen Klassen Interfaces erstellt da es mir damit, um einiges leichter viel mit den Unittests zu arbeiten. (**AuthDB, CardDB, DBAccess, DeckDB, PackageCreationDB, PackageTransactionDB, UserDB**)

2. Lessons Learned

Extraktion von Kartentyp und Elementtyp aus dem Namen:

- In einigen Fällen waren die Werte für den Kartentyp und den Elementtyp nicht direkt vorhanden, sondern mussten aus dem Namen der Karte extrahiert werden. Diese Entscheidung wurde getroffen, um die Datenstruktur und die Speicherung in der Datenbank effizienter zu gestalten. Das Extrahieren der Werte aus dem Namen erfolgt durch eine angepasste Logik, die die Karte basierend auf einem festgelegten Namensschema analysiert und die entsprechenden Attribute automatisch zuweist. Auf diese Weise wird sichergestellt, dass die Kartenobjekte korrekt erstellt werden, selbst wenn die Typen zunächst nicht vorliegen. Diese Entscheidung trägt zu einer dynamischen und skalierbaren Verarbeitung bei, besonders in einem sich ständig ändernden System wie bei zufällig generierten Kartenpaketen.

Speicherung der Karten in der Datenbank bei Paket-Hinzufügung:

- Eine weitere wichtige Entscheidung betraf die Handhabung der Karten im Zusammenhang mit Paketen. Wenn ein neues Paket einem Benutzer zugeordnet wird, werden die enthaltenen Karten gleichzeitig einzeln in der Datenbank gespeichert. Dies ermöglicht eine saubere Trennung und eine verbesserte Skalierbarkeit des Systems, sodass Karten nicht dupliziert werden müssen, sondern nur einmal gespeichert und später referenziert werden können. Zudem erfolgt die Speicherung der Zuordnung der Karten zu einem Paket in der Tabelle package_cards, was eine effektive Verwaltung der Beziehung zwischen Karten und Paketen sicherstellt. Diese Entscheidung trägt zur Konsistenz der Daten und einer einfacheren Datenabfrage bei.

Optimierung durch Tabellenstruktur:

- Die Datenbankstruktur wurde so optimiert, dass sowohl Karten als auch Pakete klar voneinander getrennt gespeichert werden, aber dennoch effizient miteinander verknüpft sind. Die Verwendung einer separaten Tabelle für die game_card und die game_package, sowie deren Verknüpfung in der package_cards-Tabelle, gewährleistet, dass die Karten nicht redundant gespeichert werden und gleichzeitig flexibel unterschiedlichen Paketen zugeordnet werden können. Diese Entscheidung verbessert die Wartbarkeit der Datenbank und fördert eine effiziente Nutzung von Datenbank-Ressourcen.

BattleLogic

- Hier konnte ich lernen das eine solide Objektorientierung essenziell war. Außerdem lernte ich wie man 2 Spieler mittels einer Queue miteinander verbinden kann hierbei hatte ich vor allem Probleme, bis ich herausgefunden habe, wie ich das am besten mache. Beim Spielablauf konnte ich auch lernen, wie ich bestimmte Abläufe richtig handhabe und einen gut strukturierten BattleLog erstellen kann.

3. Unit-Testing-Entscheidungen

3.1 Testabdeckung

Tests für HTTP-Antworten: Es wurden Unit-Tests für HTTP-Statuscodes und -antworten durchgeführt, um sicherzustellen, dass sowohl Header als auch Body korrekt aufgebaut werden.

Tests für das Request Parsen: Es wurden Unit-Tests erstellt, die die korrekte Verarbeitung von HTTP-Anfragen durch das Request Parsing sicherstellen.

Token-Authentifizierung: Es wurden Tests zur Sicherstellung der korrekten Verarbeitung und Validierung von Token bei der Benutzerautorisierung durchgeführt.

Package-Erstellung: Weitere Tests wurden hinzugefügt, um zu überprüfen, ob die Erstellung und Zuweisung von Packages sowie die zugehörige Datenbankintegration fehlerfrei funktioniert.

User-Registrierung: Es wurden Tests entwickelt, um sicherzustellen, dass die Registrierung von Benutzern korrekt und fehlerfrei erfolgt.

Multithreading: Es wurde ein Test erstellt der die Fähigkeit Multible Anfragen zu verarbeiten überprüft.

3.2 Teststrategie

- **Mocking** wurde eingesetzt, um zu vermeiden, dass echte Datenbankanfragen während der Tests gesendet werden.
- **JUnit 5** wird als Test-Framework verwendet, um sowohl einfache als auch komplexe Anfragen und deren Bearbeitung durch den Server zu prüfen.
- **Interfaces für Klassen** verwendet damit die Tests besser funktionieren

4. Besondere Merkmale des Projekts

Ein besonderes Merkmal des Projekts ist die Implementierung einer eigenen **HTTP-Protokoll- und Header-Verarbeitung** ohne die Nutzung von externen HTTP-Frameworks. Dies wurde mit dem Ziel entwickelt, ein besseres Verständnis der HTTP-Kommunikation und Server-Client-Interaktion zu gewinnen. Was so wie von der Angabe gewünscht war.

Sonst könnte man auch noch erwähnen das ich Interfaces in meiner Datenbank Struktur verwendet habe damit die Unittests besser funktionieren.

Ich habe in der Datenbank noch eine Tabelle für PackageTransaktions eingefügt mit der verständlich abgespeichert wird wann und wer welches bestimmte Paket erlangt hat.

Außerdem habe ich noch bei meiner BattleLogic ein zusätzliches Feature implementiert namens (Wetterbedingungen). Am Anfang jedes Kampfes wird eine bestimmte Wetterbedingung zufällig gesetzt. Diese dient dazu die schwächen bestimmter Karten ein wenig auszugleichen (gibt einem Typen oder ElementTypen einen buff und deren Counterpart einen debuff) und gibt jedem Kampf die Möglichkeit etwas anders zu sein als der Letzte.

5. Zeitaufwand

Die nachfolgende Zeitverfolgung enthält eine detaillierte Auflistung der aufgewendeten Zeit für verschiedene Tätigkeiten:

Aktivität	*Zeitaufwand*
Architekturplanung und Design	10 Stunden
Implementierung der API	40 Stunden
Datenbankintegration	25 Stunden
Implementierung der Authentifizierung	8 Stunden
Unit-Tests und Fehlerbehebung	12 Stunden
Dokumentation und Protokoll	5 Stunden
Gesamtzeit	100 Stunden

6. Link zum GitHub-Repository

Das vollständige Projekt sowie die Dokumentation sind im folgenden Git-Repository verfügbar:

https://github.com/siky11/MonsterCardGame.git