App Design: Entscheidungen und Struktur

Designentscheidungen

Architektur

Die Anwendung folgt einer mehrschichtigen Architektur (Layered Architecture) mit klar getrennten Verantwortlichkeiten:

1. **Controller-Schicht:**
    a. Verantwortlich für die Verarbeitung von HTTP- Requests und - Responses.
    b. Bereitstellung von REST-API-Endpunkten für verschiedene Entitäten.
2. **Service-Schicht:**
    a. Implementierung der business logic, z. B. Kartenmanagement, Handelsregeln, ELO-Berechnungen.
    b. Validierung der Anwendungslogik.
3. **Repository-Schicht:**
    a. Direkter Datenzugriff auf eine PostgreSQL-Datenbank.
    b. Persistierung und Abfrage von Entitäten wie Benutzern, Karten und Handelsgeschäften.
4. **Datenbank:**
    a. Nutzung von PostgreSQL zur sicheren Speicherung aller Daten.
    b. Tabellen für Benutzer, Karten, Decks, Packages, Trading_Deals, ELO und Stats.

Warum wurde diese Architektur gewählt?

- **Trennung der Verantwortlichkeiten:** Jede Schicht hat eine klar definierte Aufgabe, was die Wartbarkeit und Erweiterbarkeit der Anwendung verbesserten.
- **Testbarkeit:** Durch die klare Trennung der Schichten können Unit-Tests und Integrationstests einfacher durchgeführt werden.
- **Wiederverwendbarkeit:** Die business logic in den Services kann leicht in anderen Teilen der Anwendung wiederverwendet werden.

Technologie-Entscheidungen

- **Programmiersprache:** Java, um die serverseitige Logik zu implementieren.
- **HTTP-Kommunikation:** Eigene Implementierung des HTTP-Protokolls ohne Frameworks, um die Kontrolle über die Kommunikation zu maximieren.
- **Sicherheitsmechanismen:** Token-basierte Authentifizierung zur Gewährleistung, dass Aktionen nur von berechtigten Benutzern durchgeführt werden.

- **ELO-System:** Einfache Berechnung (+3 Punkte für einen Sieg, -5 Punkte für eine Niederlage) zur Bewertung der Spielerleistung.

Struktur der Anwendung

Die Anwendung ist in folgende Pakete und Klassen unterteilt:

- **Paketstruktur:**
  - **entities**: Enthält alle Entitätsklassen (User, Card, Deck, Package, TradingDeal, UserStats, UserElo).
  - **controllers:** Enthält die Controller-Klassen (UserController, CardPackageController, TradingController, BattleController, etc.).
  - **services:** Enthält die Service-Klassen (UserService, CardPackageService, TradingService, BattleService, etc.).
  - **repositories:** Enthält die Repository-Klassen (UserRepository, CardPackageRepository, TradingRepository, etc.).
- **Datenbankintegration:**
  - Die PostgreSQL-Datenbank wurde verwendet, um die Daten der Anwendung persistent zu speichern.
  - Die Tabellenstruktur wurde so entworfen, dass sie die Beziehungen zwischen den Entitäten abbildet (z.B. 1: n-Beziehung zwischen User und Card).
  - Die Repository-Klassen sind für die Kommunikation mit der Datenbank verantwortlich und verwenden SQL-Abfragen, um Daten abzurufen, zu speichern und zu aktualisieren.

Design-Patterns

- **Repository Pattern:**
  - Das Repository Pattern wurde verwendet, um die Datenbankzugriffe zu kapseln und die Business Logik von der Datenzugriffsschicht zu trennen.
  - Dies ermöglicht eine einfachere Testbarkeit und Austauschbarkeit der Datenbankzugriffslogik.
- **Service Layer:**
  - Die Service-Schicht wurde implementiert, um die Business Logik von den Controllern zu trennen. Dies ermöglichten eine bessere Wiederverwendbarkeit und Testbarkeit der Logik.

Kommunikation zwischen den Schichten

- **Datenfluss:**
  1. Ein HTTP-Request wird vom Client an den Controller gesendet.
  2. Der Controller validiert die Eingaben und leitet die Anfrage an den entsprechenden Service weiter.

3. Der Service führt die Business Logik aus und ruft bei Bedarf das Repository auf, um Daten aus der Datenbank abzurufen oder zu speichern.
4. Das Repository führt die SQL-Abfragen aus und gibt die Ergebnisse an den Service zurück.
5. Der Service verarbeitet die Daten und gibt das Ergebnis an den Controller zurück.
6. Der Controller sendet die HTTP-Response an den Client.

Describes Lessons Learned

Während der Entwicklung des Monster Trading Cards Game (MTCG) habe ich viele wertvolle Erfahrungen gesammelt, die mir nicht nur bei diesem Projekt, sondern auch für zukünftige Softwareentwicklungsvorhaben helfen werden. Hier sind die wichtigsten Lessons Learned:

1. Wichtigkeit einer klaren Architektur

- **Lektion:** Eine gut durchdachte Architektur (z.B. Layered Architecture mit Entitäten, Controllern, Services und Repositorys) ist entscheidend für die Wartbarkeit und Skalierbarkeit einer Anwendung.
- **Erfahrung:** Durch die klare Trennung der Verantwortlichkeiten konnte ich Fehler leichter lokalisieren und Code einfacher erweitern. Beispielsweise konnte ich die Business Logik in den Services isolieren, was das Testen und Debuggen erheblich vereinfacht hat.
- **Zukünftige Anwendung:** In zukünftigen Projekten werde ich von Anfang an auf eine saubere Architektur achten, um spätere Refactorings zu vermeiden.

2. Testen ist unerlässlich

- **Lektion:** Unit-Tests und Integrationstests sind nicht optional, sondern ein Muss, um die Stabilität und Funktionalität der Anwendung sicherzustellen.
- **Erfahrung:** Während der Entwicklung habe ich festgestellt, dass Fehler oft erst durch Tests aufgedeckt wurden, insbesondere in komplexen Logiken wie der Battle-Logik oder dem Trading-System. Ohne Tests wären viele Fehler erst in der Produktion aufgetreten.
- **Zukünftige Anwendung:** Ich werde in Zukunft von Anfang an Test-Driven Development (TDD) oder zumindest regelmäßige Tests in den Entwicklungsprozess integrieren.

3. Datenbankdesign ist kritisch

- **Lektion:** Ein gut durchdachtes Datenbankdesign ist die Grundlage für eine effiziente Anwendung.

- **Erfahrung:** Bei der Implementierung der Datenbank musste ich feststellen, dass einige Tabellenstrukturen nicht optimal waren, was zu Performance-Problemen führte. Beispielsweise war die Beziehung zwischen User und Card anfangs nicht gut normalisiert, was zu redundanten Daten führte.
- **Zukünftige Anwendung:** In zukünftigen Projekten werde ich mehr Zeit in die Planung des Datenbankdesigns investieren und sicherstellen, dass die Tabellen gut normalisiert sind.

4. Kommunikation zwischen Schichten

- **Lektion:** Die Kommunikation zwischen den Schichten (Controller, Service, Repository) muss klar und effizient sein.
- **Erfahrung:** Anfangs gab es einige Inkonsistenzen in der Kommunikation zwischen den Schichten, z.B. wurden manchmal Daten direkt zwischen dem Controller und dem Repository ausgetauscht, was die Business Logik umging. Dies führte zu Fehlern und schlechter Wartbarkeit.
- **Zukünftige Anwendung**: Ich werde sicherstellen, dass die Kommunikation zwischen den Schichten immer über die richtigen Kanäle erfolgt (z.B. Controller → Service → Repository).

5. Zeitmanagement

- **Lektion:** Gutes Zeitmanagement ist entscheidend, um Deadlines einzuhalten und den Überblick über den Fortschritt zu behalten.
- **Erfahrung:** In einigen Phasen des Projekts habe ich unterschätzt, wie viel Zeit bestimmte Aufgaben (z.B. Implementierung der Battle-Logik oder des Trading-Systems) in Anspruch nehmen würden. Dies führte zu Zeitdruck in der Endphase des Projekts.
- **Zukünftige Anwendung:** Ich werde in Zukunft realistischere Zeitpläne erstellen und regelmäßig den Fortschritt überprüfen, um sicherzustellen, dass ich im Zeitplan bleibe.

6. Nutzung von Version Control (Git)

- **Lektion:** Die konsequente Nutzung von Version Control (z.B. Git) ist unerlässlich, um den Überblick über Änderungen zu behalten und bei Bedarf zu früheren Versionen zurückzukehren.
- **Erfahrung:** Durch die regelmäßige Nutzung von Git konnte ich problemlos zwischen verschiedenen Entwicklungsständen wechseln und Fehler beheben, die in früheren Versionen aufgetreten waren.
- **Zukünftige Anwendung:** Ich werde weiterhin Git konsequent nutzen und sicherstellen, dass alle Änderungen gut dokumentiert sind.

Describes Unit Testing Decisions

UserService

Die Unit-Tests für die UserService-Klasse wurden sorgfältig entworfen, um die Kernfunktionalitäten der Benutzerverwaltung abzudecken. Die Tests konzentrieren sich auf die wichtigsten Methoden, die für die Benutzerregistrierung, Anmeldung, Datenabfrage und Profilaktualisierung verantwortlich sind. Hier ist eine detaillierte Beschreibung der Testentscheidungen:

Getestete Methoden

1. **create(User user):**

   Tests: Erfolgreiche Registrierung (testCreateUser_Success) und Registrierung mit existierendem Benutzernamen (testCreateUser_AlreadyExists).

2. **getUserData(String username):**

   Tests: Erfolgreiches Abrufen von Benutzerdaten (testGetUserData_Success) und Abfrage eines nicht existierenden Benutzers (testGetUserData_NotFound).

3. **login(User user):**

   Tests: Erfolgreiche Anmeldung (testLogin_Success) und fehlgeschlagene Anmeldung (testLogin_Failure).

4. **updateUser(String username, User updatedUser):**

   Tests: Erfolgreiche Aktualisierung des Profils (testUpdateUser_Success) und Aktualisierung eines nicht existierenden Benutzers (testUpdateUser_NotFound).

Mocking und Testabdeckung

Mocking: UserDbRepository und TokenService wurden mit Mockito gemockt, um externe Abhängigkeiten zu isolieren.

Testabdeckung: Die Tests decken die wichtigsten Szenarien ab, einschließlich erfolgreicher und fehlgeschlagener Fälle.

CardPackageService

Die Unit-Tests für die CardPackageService-Klasse decken die kritischen Funktionalitäten rund um das Erstellen, Kaufen und Abrufen von Kartenpaketen ab. Hier ist eine kompakte Zusammenfassung:

Getestete Methoden

1. **createPackage(ArrayList<Card> cards, String token):**

Testfälle: Erfolgreiches Erstellen eines Pakets durch den Admin (testCreatePackage_Success) und fehlgeschlagener Versuch durch einen regulären Benutzer (testCreatePackage_NotAdmin).

Ziel: Sicherstellen, dass nur der Admin Pakete erstellen kann.

2. **buyPackage(String token):**

   Testfälle: Erfolgreicher Kauf eines Pakets (testBuyPackage_Success) und fehlgeschlagener Kauf, wenn keine Pakete verfügbar sind (testBuyPackage_NoPackagesAvailable).

   Ziel: Überprüfen, dass Benutzer Pakete kaufen können und die Münzen korrekt abgezogen werden.

3. **getUserCards(String token):**

   Testfall: Erfolgreiches Abrufen der Karten eines Benutzers (testGetUserCards_Success).

   Ziel: Sicherstellen, dass die Karten eines Benutzers korrekt zurückgegeben werden.

Mocking und Testabdeckung

Mocking: CardPackageRepository und UserDbRepository wurden mit Mockito gemockt, um externe Abhängigkeiten zu isolieren.

Testabdeckung: Die Tests decken die wichtigsten Szenarien ab, einschließlich Erfolgs- und Fehlerfälle.

DeckService

Die Unit-Tests für die DeckService-Klasse decken die kritischen Funktionalitäten rund um das Erstellen und Abrufen von Decks ab. Hier ist eine kompakte Zusammenfassung:

Getestete Methoden

1. createDeck(ArrayList<UUID> cardsID, String token):

   Testfälle: Erfolgreiches Erstellen eines Decks (createDeck_ValidData_CreatesDeckSuccessfully).

   Fehlgeschlagene Versuche durch ungültige Token (createDeck_InvalidToken_ThrowsIllegalArgumentException), nicht existierende Benutzer (createDeck_UserNotFound_ThrowsIllegalArgumentException), falsche Deck-Größe (createDeck_DeckSizeInvalid_ThrowsIllegalArgumentException) und ungültige Karten (createDeck_InvalidCard_ThrowsBadRequestException).

Ziel: Sicherstellen, dass Decks nur mit gültigen Daten erstellt werden.

2. getDeck(String token):

   Testfälle: Erfolgreiches Abrufen eines Decks (getDeck_ValidToken_ReturnsDeck) und fehlgeschlagener Versuch ohne Token (getDeck_TokenNotProvided_ThrowsIllegalArgumentException).

   Ziel: Sicherstellen, dass Decks in einfacher Form korrekt abgerufen werden.

Mocking und Testabdeckung

Mocking: DeckDbRepository, UserDbRepository und CardPackageRepository wurden mit Mockito gemockt, um externe Abhängigkeiten zu isolieren.

Testabdeckung: Die Tests decken die wichtigsten Szenarien ab, einschließlich Erfolgs- und Fehlerfälle.

BattleService

Die Unit-Tests für die BattleService-Klasse decken die Schadensberechnung und spezielle Kampfregeln ab. Hier ist eine kompakte Zusammenfassung:

Getestete Methoden

1. calculateDamage(Card attacker, Card defender):

   Testfälle: Feuer gegen Wasser: Überprüft, ob der Schaden verdoppelt wird, da Wasser gegen Feuer effektiv ist (testDamageCalculationFireAgainstWater).

   Goblin gegen Drache: Überprüft, ob Goblins keine Schäden verursachen, da sie Angst vor Drachen haben (testGoblinVsDragon).

   Zauberer gegen Ork: Überprüft, ob Orks keinen Schaden verursachen, da Zauberer sie kontrollieren (testWizardVsOrk).

   Ziel: Sicherstellen, dass die Schadensberechnung und speziellen Regeln korrekt implementiert sind.

Mocking und Testabdeckung

Mocking: BattleRepository, UserDbRepository und DeckDbRepository wurden mit Mockito gemockt, um externe Abhängigkeiten zu isolieren.

Testabdeckung: Die Tests decken die wichtigsten Szenarien ab, einschließlich Effektivität von Elementen und speziellen Monsterfähigkeiten.

StatsService

Die Unit-Tests für die StatsService-Klasse decken die Abrufung von Benutzerstatistiken und die Token-Verarbeitung ab. Hier ist eine kompakte Zusammenfassung:

Getestete Methoden

1. getUserStats(String token):

   Testfälle: Erfolgreiches Abrufen der Statistiken (getUserStats_ValidToken_ReturnsUserStats).

   Fehlgeschlagene Versuche durch ungültigen Token (getUserStats_InvalidToken_ThrowsIllegalArgumentException), nicht existierende Benutzer (getUserStats_UsernameNotFound_ThrowsException) und fehlende Statistiken (getUserStats_StatsNotFound_ThrowsSQLException).

   Ziel: Sicherstellen, dass Benutzerstatistiken korrekt abgerufen werden und Fehlerfälle behandelt werden.

2. extractUsernameFromToken(String token):

   Testfälle: Erfolgreiche Extraktion des Benutzernamens (extractUsernameFromToken_ValidToken_ReturnsUsername).

   Fehlgeschlagene Versuche durch ungültiges Token (extractUsernameFromToken_InvalidToken_ThrowsIllegalArgumentException) und null-Token (extractUsernameFromToken_NullToken_ThrowsIllegalArgumentException).

   Ziel: Überprüfen, dass der Benutzername korrekt aus dem Token extrahiert wird.

Mocking und Testabdeckung

Mocking: StatsDbRepository und UserDbRepository wurden mit Mockito gemockt, um externe Abhängigkeiten zu isolieren.

Testabdeckung: Die Tests decken die wichtigsten Szenarien ab, einschließlich Erfolgs- und Fehlerfälle.

TradeService

Die Unit-Tests für die TradeService-Klasse decken die Erstellung von Handelsangeboten ab. Hier ist eine kompakte Zusammenfassung:

Getestete Methoden

1. createTradingDeal(TradingDeal tradingDeal, String token):

   Testfälle: Karte im Deck: Fehlgeschlagener Versuch, ein Handelsangebot zu erstellen, wenn die Karte im Deck ist (createTradingDeal_CardInDeck_ThrowsException).

Ungültiges Handelsangebot: Fehlgeschlagener Versuch, ein ungültiges Handelsangebot zu erstellen (createTradingDeal_InvalidTradingDeal_ThrowsException).

Erfolgreiche Erstellung: Erfolgreiche Erstellung eines Handelsangebots (createTradingDeal_ValidTradingDeal_Success).

Ziel: Sicherstellen, dass Handelsangebote nur mit gültigen Daten erstellt werden und Fehlerfälle korrekt behandelt werden.

Mocking und Testabdeckung

Mocking: TradingDealRepository, UserDbRepository und CardPackageRepository wurden mit Mockito gemockt, um externe Abhängigkeiten zu isolieren.

Testabdeckung: Die Tests decken die wichtigsten Szenarien ab, einschließlich Erfolgs- und Fehlerfälle.

Github Repository Link:

https://github.com/Mohamad1102/http-server-and-app