LABORPROTOKOLL

im Studiengang Bachelor Informatik / Computer Science
Lehrveranstaltung Software Engineering 1

Retake Exam SWEN1
Projektabgabe

Ausgeführt von: Johanna Nasr
Personenkennzeichen: if23b284

BegutachterIn: Hr. Michael Strommer

Wien, 06.03.25

FH University of Applied Sciences
TECHNIKUM WIEN

Inhalt

1 App Design (Entscheidungen, Struktur)

Ich habe für dieses Projekt eine Mehrschichten-Architektur verwendet (Controller → Service → Repository → Database). Die Controller implementieren die REST-Endpunkte (/users, /sessions, /packages, /transactions/packages, /cards, /deck, /stats, /scoreboard, /battles, /tradings) und nehmen die Anfragen entgegen. Das Service-Package enthält die Geschäftslogik. Das Repository behandelt die Operationen für die Datenbank. Das Database-Package stellt die Verbindung zur PostgreSQL-Datenbank bereit. Model definiert die Datenstrukturen, in meinem Fall User und Card. Die Kommunikation erfolgt über JSON. Die Kommunikation über JSON bedeutet, dass Daten (zum Beispiel Anfragen und Antworten) im JSON-Format übertragen werden. In den Controllern wird beim Empfangen von den Requests der Body in einen String umgewandelt und der String enthält dann JSON-Daten. Ein weiteres lesson learned in dem Bezug ist die library Jackson, die die Strings in POJOs umwandelt.

Ich habe bei diesem Antritt mich bewusst vorm Anfangen des Projekts bzw. vor dem Programmieren mit Projektstrukturen befasst, damit ich eine geeignete Struktur habe. Ich habe mich dann eben für diese entschieden und so einfach wie möglich gestaltet. Jedes Package hat seine Aufgabe und jede Class ist auch gleich aufgebaut, damit ich gezielt nach auftretenden Problemen suchen kann. Ich habe dies so aufgebaut, damit ich neue Endpunkte und Funktionen einfacher hinzufügen kann. Ich habe die Datenbank „statisch“ angelegt und nicht per Dockerfile, weil es für mich auf diese Art und Weise einfacher zum Nachvollziehen war. Die Prepared Statements waren ebenfalls bewusste Entscheidungen, um SQL-Injections zu verhindern.

Der ServerMain startet den Server und registriert alle Controller. Jeder Controller ist für seinen „Bereich“ zuständig, Das Service ruft das Repository auf und das wiederum greift auf die Datenbank zu (Prepared Statements).

Ich habe mir als Struktur Folgendes überlegt: Packages werden im Service erstellt und im Repository werden die Karten gesammelt. Wenn man 5 Karten hat, wird das in eine statische Liste gespeichert. Wenn ein User ein Package kauft, wird das erste aus der Liste genommen und den User-Karten zugeordnet. Das Deck wird über den DeckController (REST-Endpunkt /deck) und den DeckService konfiguriert. Das DeckService ruft Repository auf, um das bestehende Deck des Users zu löschen und neue IDs zu speichern, weshalb ich kein eigenes Model für Deck habe, sondern in der DB das abbilde.

Update 06.03.2025: Datenbank (samt Tabellen) wird jetzt per Dockerfile erstellt. Ich habe allerdings in dem yaml file den Port auf 5433 geändert, weil 5422 default Postgres belegt ist.

2 Lessons learned

Mein erstes Problem war beim Einloggen in die DB. Das default-password hatte ich nicht beabsichtigt umgestellt und musste dann durch das config file das Passwort umändern, was mich viele Stunden gekostet hat. → **Datenbank-Anbindung**

Dadurch, dass ich nach dem intermediate Hand-In meine Zeit nicht richtig gemanagt habe, hatte ich kein funktionierendes Projekt. Ich habe daraus gelernt, dass man sich neben den anderen Fächern die Zeit besser aufteilen muss. Außerdem habe ich bei diesem Anlauf die Datenbank- und allgemeine Projektstruktur komplett umgedacht, weshalb ich auch da schon viel mehr Zeit gebraucht habe. → **Zeitmanagement**

Das Code-Kommentieren bzw. das „Debuggen“ durch System.out.println hat mir sehr geholfen → **Ausgabe / Code refactoring**

Jedes Mal, wenn etwas funktioniert hat, hätte ich direkt den Code pushen bzw. committen sollen. Ich hätte damit nicht nur wenige Commits übers ganze Projekt, sondern tagtäglich mehrere Eintragungen gehabt → **Git**

3 Unit testing decisions

Bei den Unit Tests habe ich darauf geachtet, so viel wie möglich abzudecken in den verschiedensten Bereichen. Die Tests für Registrierung, Login, Package-Kauf, Deck-Configuration und Battle-Logik stellen sicher, dass mal die Anfragen an sich funktionieren ohne die Detailfunktionalitäten. Viele Tests fokussieren sich auf die Service-Klasen.

Update 06.03.2025: Ich habe 4 Tests bezogen auf den User in eine eigene Klasse gegeben und den Rest in der GameServiceTest-Klasse gelassen.

4 Unique feature

➔ Die Sphinx-Karte!

Sobald im Battle eine Karte namens „Sphinx“ gespielt wird (und der Gegner keine Sphinx hat), gewinnt die Sphinx automatisch die Runde. Dieser Effekt ist in der Methode `startBattle()` in der Klasse `GameService` implementiert. Im Battle-Code wird bei jeder Runde geprüft, ob eine der beiden Karten „Sphinx“ ist. Wenn genau eine Karte „Sphinx“ ist, gewinnt diese Runde automatisch. Ich habe mich für dieses Feature entschieden, weil ich einen ägyptischen Hintergrund habe und meiner Meinung nach die Sphinx auch als Game-Karte gut geeignet ist.

5 Zeitaufwand

Insgesamt habe ich dieses Projekt wie folgt aufgeteilt bzw. gemacht:

Tag	Zeit (in Stunden)	Was?
1	12	Recherche (!), Projektstruktur neu angelegt, DB-Struktur neu durchdacht, Must-Haves & Features wirklich verstanden, dependencies
2	9	Controller instanziert, User, Session, Card
3	1	DB statisch
4	9	Deck, Package
5	10	Battle
6	6	Transaction
7	3	Unique feature überlegt, eingebaut
8	8	Stats, Trading
9	3	Unit Tests überlegen und codieren
10	1	Scoreboard begonnen, nicht fertig
11	2	Ausgaben zum debuggen
12	3	DB mit docker umgebaut, Protokoll, code & project refactoring

= gesamt 67 Stunden für den Zweitantritt

Ich habe immer einen ganzen Code-„Haufen“ auf einmal committed und gepusht, was ich im Nachhinein gerne anders gemacht hätte (→ lessons learned). Die Tage sind auch nicht „Kalendertage““, also Tag 1 ist nicht vor 12 Tagen passiert, sondern das ist der erste Tag, an dem ich mich mit dem Projekt für den Retake auseinandergesetzt habe.

6 GIT

https://github.com/NaJoKa/MTCG_V2