Beschreibungen und Anmerkungen zu MCTG

Inhalt

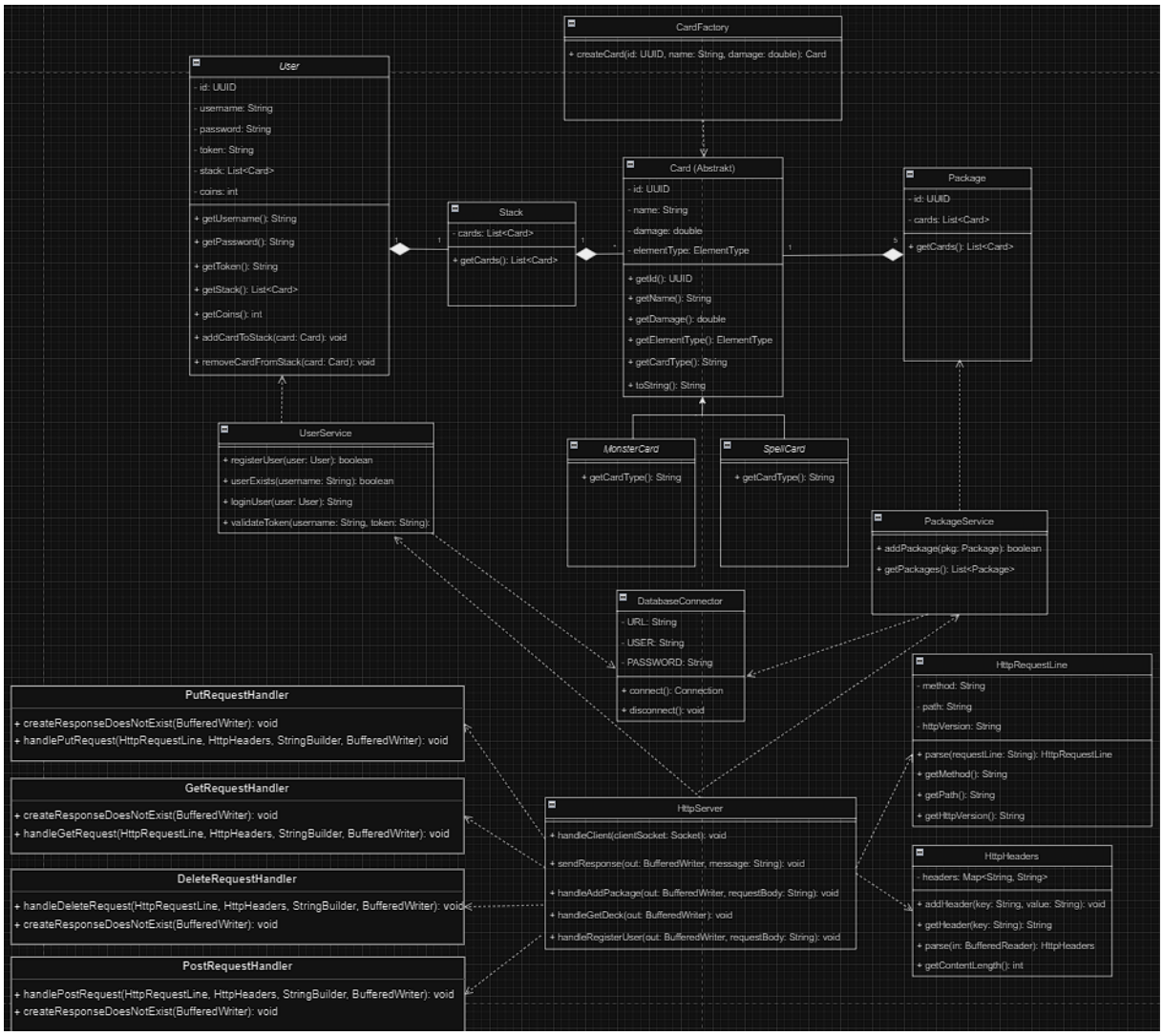
GitHub Link....1
Klassendiagramm....1
Layer....3
Erklärung der Klassen....4
DatabaseConnector....4
Main....4
HttpServer....4
HttpRequestLine und HttpHeaders....4
DeleteRequestHandler, GetRequestHandler und PutRequestHandler....4
PostRequestHandler....4
Card....5
CardFactory....5
Package....5
PackageService....5
Stack....5
User....5
UserService....5
Unit Tests....6
Anmerkungen:....6
Zeitinvestment....6

GitHub Link

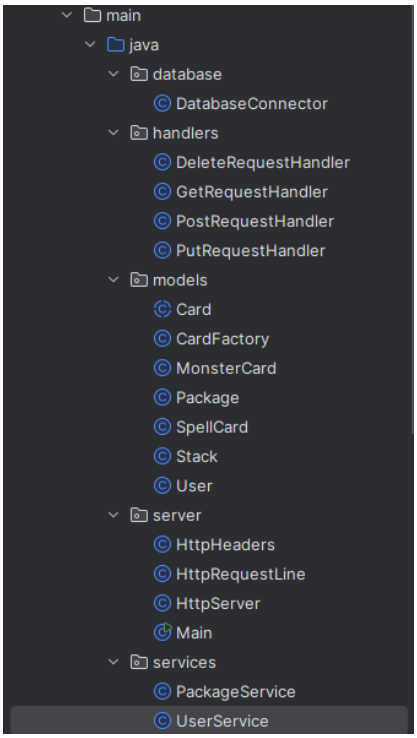
https://github.com/Marcel19985/MCTG_Goessl

Klassendiagramm

Eine detaillierte Struktur kann dem Klassendiagramm entnommen werden. Dieses wurde mit draw.io erstellt und kann daher mit dieser Anwendung geöffnet werden. Ich habe mich für draw.io entschieden, weil wir dafür eine kurze Einführung im letzten Semester erhalten haben und ich seitdem damit arbeite. Eine .io Datei befindet sich in der Abgabe. Überblick:


CardFactory
+ createCard(id: UUID, name: String, damage: double): Card
User
- id: UUID
- username: String
- password: String
- token: String
- stack: List<Card>
- coins: int
+ getUsername(): String
+ getPassword(): String
+ getToken(): String
+ getStack(): List<Card>
+ getCoins(): int
+ addCardToStack(card: Card): void
+ removeCardFromStack(card: Card): void
Stack
- cards: List<Card>
+ getCards(): List<Card>
Card (Abstrakt)
- id: UUID
- name: String
- damage: double
- elementType: ElementType
+ getId(): UUID
+ getName(): String
+ getDamage(): double
+ getElementType(): ElementType
+ getCardType(): String
+ toString(): String
Package
- id: UUID
- cards: List<Card>
+ getCards(): List<Card>
UserService
+ registerUser(user: User): boolean
+ userExists(username: String): boolean
+ loginUser(user: User): String
+ validateToken(username: String, token: String):
MonsterCard
+ getCardType(): String
SpellCard
+ getCardType(): String
PackageService
+ addPackage(pkg: Package): boolean
+ getPackages(): List<Package>
DatabaseConnector
- URL: String
- USER: String
- PASSWORD: String
+ connect(): Connection
+ disconnect(): void
HttpRequestLine
- method: String
- path: String
- httpVersion: String
+ parse(requestLine: String): HttpRequestLine
+ getMethod(): String
+ getPath(): String
+ getHttpVersion(): String
PutRequestHandler
+ createResponseDoesNotExist(BufferedWriter): void
+ handlePutRequest(HttpRequestLine, HttpHeaders, StringBuilder, BufferedWriter): void
GetRequestHandler
+ createResponseDoesNotExist(BufferedWriter): void
+ handleGetRequest(HttpRequestLine, HttpHeaders, StringBuilder, BufferedWriter): void
DeleteRequestHandler
+ handleDeleteRequest(HttpRequestLine, HttpHeaders, StringBuilder, BufferedWriter): void
+ createResponseDoesNotExist(BufferedWriter): void
PostRequestHandler
+ handlePostRequest(HttpRequestLine, HttpHeaders, StringBuilder, BufferedWriter): void
+ createResponseDoesNotExist(BufferedWriter): void
HttpServer
+ handleClient(clientSocket: Socket): void
+ sendResponse(out: BufferedWriter, message: String): void
+ handleAddPackage(out: BufferedWriter, requestBody: String): void
+ handleGetDeck(out: BufferedWriter): void
+ handleRegisterUser(out: BufferedWriter, requestBody: String): void
HttpHeaders
- headers: Map<String, String>
+ addHeader(key: String, value: String): void
+ getHeader(key: String): String
+ parse(in: BufferedReader): HttpHeaders
+ getContentLength(): int

Layer



services: kümmern sich um Methoden, bei denen mit der Datenbank interagiert wird.

server: kümmert sich um Client Requests und Responses

models: Modellklassen

handlers: teilen die http Methoden auf, um unübersichtliche Dateien zu verhindern

database: stellt Verbindung zur Datenbank her

Erklärung der Klassen

DatabaseConnector

Stellt eine Verbindung zur Datenbank mit den jeweiligen Eigenschaften (URL, Username und Passwort) her.

Main

void main(…): Startet einen ServerSocket auf Port 10001 (Portnummer ist im CURL Script vorgegeben. Sobald sich ein Client verbindet wird ein neues Thread erstellt und der clientSocket wird in der Klasse HttpServer weiterverarbeitet.

HttpServer

void handleClient(…): Liest aus dem clientSocket die HTTP Request Line und die headers aus und speichert diese jeweils als Objekt. Der Request Body wird als String gespeichert und weiterverarbeitet. Abhängig von der HTTP Methode wird eine der folgenden Methoden aufgerufen, die die jeweilige CRUD Operation ausführt:

- handlePostRequest
- handleGetRequest
- handlePutRequest
- handleDeleteRequest

Für jede dieser Methode wurde eine eigene Klasse erstellt, die sich im Package „handlers“ befinden. Diese Struktur habe ich erstellt, nachdem der Code sehr unübersichtlich in der handleClient Methode wurde (zunächst wurden alle Methoden in if-else Blöcken abgearbeitet, wodurch seit dem Hinzufügen aller API Endpoints sehr vielen Zeilen entstanden sind).

HttpRequestLine und HttpHeaders

Durch diese Klassen lassen sich die Request Line und Headers als Objekte abspeichern. Die Klassen enthalten nützliche Methoden, um ein Request übersichtlicher bearbeiten zu können.

DeleteRequestHandler, GetRequestHandler und PutRequestHandler

All diese Klassen enthalten Methoden, um die Logik basierend auf dem Pfad (z.B. „/tradings“) auszuführen. Bis jetzt werden hier lediglich die API Endpoints definiert. Es wird bei einem jeweiligen Aufruf nur eine http 501 Response („Not Implementd“) zurückgegeben.

PostRequestHandler

handlePostRequest(…): Basierend auf dem Pfad („/users“ für die Registrierung, „/sessions“ für die Anmeldung oder „/packages“ für das Erstellen neuer Packages) wird die jeweilige Anfrage verarbeitet. Der Ablauf ist bei den ersten beiden ähnlich: Aus den JSON-Daten wird ein Objekt erstellt und die Registrierung/Anmeldung durchgeführt. Abhängig vom Erfolg der Durchführung, wird die jeweilige Response erstellt und an den Client zurück gesendet.

Bei der Erstellung neuer Packages wird zunächst geprüft, ob ein Admin-Token übermittelt wurde. Danach wird das übermittelte JSON in einer Liste gespeichert, in der jedes Element eine Map (Key-Value Pair) ist. Es ist noch nicht möglich, Card Objekte zu erstellen, da noch nicht klar ist um welches Objekt es sich handelt (MonsterCard vs. SpellCard). Hier habe ich mich nämlich für Vererbung entschieden, weil die Karten für die Battles unterschiedliche Implementierungen bekommen werden. Mithilfe einer CardFactory- Klasse werden aus jedem Elemente der Liste (jede Karte) zu den jeweiligen Card- Objekten. Diese werden in der Datenbank in den Tabelle Cards und Packages abgebildet.

Card

Diese Klasse enthält alle Attribute für eine Karte laut Angabe. Der ElementType (Fire, Water oder Normal) wurde als ENUM definiert. Konstruktor und Getter sind implementiert. Es handelt sich um eine abstrakte Klasse, da die Methode getCardType() erst in den Kindklassen SpellCard und MonsterCard implementiert werden.

In der Datenbank besteht eine 1:n Beziehung zwischen cards und packages, wodurch cards einen Fremschlüssel auf packages enthalten.

CardFactory

Card createCard(…): Erstellt eine Karte der Kindklassen (SpellCard oder MonsterCard) basierend auf dem Namen, der im CURL command übergeben wird.

Package

Enthält eine Liste für 5 Karten und dazugehörige getter und einen Konstruktor. Ich habe mich für eine Liste als Container entschieden, um bei Bedarf die Größe der Packages verändern zu können. Um Leistungs- und Speichereffizienz zu erhöhen könnte stattdessen auch ein Array verwendet werden.

PackageService

Kümmert sich um die Datenbank inserts rund um ein Package.

boolean addPackage(…): Fügt ein Package und die zugehörigen Karten in die Datenbank ein. Dabei wird darauf geachtet, dass entweder beide Datenbank query's erfolgreich durchgeführt werden oder keines davon. Damit wird verhindert, dass ein leeres Package (also ohne Karten) hinzugefügt wird.

Stack

Enthält eine Liste für alle Karten, die ein Benutzer besitzt und zugehörige Methoden: hinzufügen und entfernen von Karten um Tradings in der Zukunft implementieren zu können.

User

Enthält alle Attribute für den User. Derzeit werden noch nicht alle verwendet, da diese erst bei späteren CURL Befehlen eine Rolle spielen. Da die Attribut- Namen mit kleinem Anfangsbuchstaben beginnen sollen, werden diese mithilfe von @JsonProperty zugewiesen. In den CURL Befehlen werden nämlich großgeschriebene Keys übergeben. Ich habe einige Methoden hinzugefügt (z.B. für den Kauf von Packages, um Karten vom Stack zu entfernen oder hinzuzufügen, getter und setter. Viele davon werden derzeit noch nicht verwendet.

UserService

Kümmert sich um die Datenbank inserts der user

boolean registerUser(User user): Stellt eine Datenbankverbindung her und fügt das übergebene Objekt in die Tabelle users ein. Falls der Benutzer bereits existiert, wird eine SQLException geworfen, da der username unique sein muss. Das braucht weniger Rechenaufwand als separat (extra SQL Abfrage) zu überprüfen, ob ein Benutzer bereits existiert.

generateToken(User user): Generiert einen Token basierend auf dem übergebenen user- Objekt. An den Benutzernamen wird „-mtcgToken“ angehängt.

loginUser(User user): Meldet einen Benutzer an wenn ein zusammengehöriges Paar an Benutzernamen und Passwort übergeben wurde. Der Token wird nach der Anmeldung an den Client zurückgegeben.

Unit Tests

Es befinden sich grundlegende Tests für verschieden Klassen im Package „test“. Diese sind hilfreich, sollten jedoch noch erweitert werden, um mehr Fälle abzudecken.

Anmerkungen:

Bei allen Tabellen der Datenbank wird eine UUID als Primärschlüssel verwendet. Diese ist sicherer gegenüber einem fortlaufenden integer und die ID’s sind global eindeutig.

Bei allen Methoden, die Datenbank query’s ausführen, werden SQL exceptions geworden und gecatched, um etwaige Fehlermeldungen zu sehen.

Für die Umwandlung von JSON in Java Objekte habe ich mich für Jackson entschieden. Dazu habe ich die meisten Informationen im Internet gefunden.

Die PostgreSQL Datenbank verwende ich über einen Docker, um Speicherplatz am Rechner zu schonen. Zu Docker gibt es einige Materialien im Internet, wodurch das Aufsetzen rasch möglich war.

Das Hinzufügen von Packages wurde bereits implementiert. Dadurch funktionieren folgende Punkte im CURL Script:

1) Create Users
2) Login Users
3) create packages
6) add new packages

Zeitinvestment

Bis zur Zwischenabgabe habe ich gut 30 Stunden investiert (ungefähre Einschätzung, habe die Zeit nicht getimed). Das setzt sich aus Programmieren und Internet Recherchen zusammen. Die größte Herausforderung waren:

- Die Projektanforderungen zu verstehen und gedanklich zu strukturieren. Zu Beginn ist es mir schwer gefallen, das Spiel zu verstehen und hatte auch keine Idee, wie eine Implementierung aussehen könnte, da ich noch kein Wissen über HTTP+REST und Java generell hatte.
- Implementierung von Teilen, die noch nicht Teil des Unterrichts waren (z.B. ist der Grundstein für dieses Projekt HTTP+REST, was erst Thema von Class 8 ist -> daher habe ich beim Beginn des Projektes einiges durch Internetrecherchen erarbeitet)
- Wissen rund um Datenbanken auffrischen, da das erste Semester schon ein bisserl her ist
- Hinzufügen von Packages inklusive Cards war zeitaufwendig, da die Zuteilung der Package-Kindklassen basierend auf dem Karten- Namen erfolgt. Herausforderungen waren das Parsing des Requests und das handling der Klassen.