Softwareentwurf

Der Softwareentwurf beinhaltet eine detaillierte Planung und Spezifikation der Softwarearchitektur, einschließlich der Aufteilung in Module oder Komponenten. Er legt fest, wie diese Module zusammenarbeiten und wie sie miteinander kommunizieren.

Projektstruktur

Unsere Projektstruktur ist in zwei Abschnitte unterteilt. Einmal das Backend, welches die Funktionalitäten bereitstellt, und das Frontend, welches sich um das Design sowie die Interaktionsmöglichkeiten kümmert. Es ist die Schnittstelle, über die unsere User mit der Anwendung interagieren.

Backend:

- microService1
    - src
        - main
            - java
                - com.service.{microService1}
                    - (configuration) if needed
                    - controller
                    - (model) if needed
                    - payload
                    - (security) if needed
                    - (repository) if needed
                    - services
                    - …
                - microService1Application.java
            - resources
                - application.properties
                - (static files)
        - test
    - target
    - mvnw
    - mvnw.cmd
    - pom.xml
    - …
- microService2
    - …
- …

Unsere Architektur basiert auf einer Maven-Architektur, die anschließend layer-basiert arbeitet. Das heißt, dass im Ordner “Controller” jegliche Controller des Micro-Services zu finden sind. Der Ordner “Payload” beinhaltet die DTOs (Data Transfer Objects), die für Anfragen an andere Services und für Abfragen von anderen Services verwendet werden. Die “pom.xml”, also die Konfigurationsdatei von Maven, beinhaltet die ganzen Dependencies, die von unseren Services benötigt werden.

Techstack:

Unser Backend-Techstack besteht aus bewährten Technologien, die sich für die Entwicklung einer stabilen und skalierbaren Anwendungsarchitektur gut eignen.

Für unser Backend wird die Programmiersprache *Java* eingesetzt, da dieses ein großes Ökosystem an Bibliotheken und Frameworks besitzt.
In Kombination dazu werden wir das Framework '*Spring Boot*' verwenden, welches sich für die Entwicklung von Java-basierten Webanwendungen eignet und viele Funktionen für die Entwicklung von RESTful Services bereitstellt.
Für die Übertragung der Objekte wird *JSON* in Zusammenarbeit mit der Objectmapper Bibliothek verwendet.

Für eine reibungslose Kommunikation zwischen Backend und Frontend, insbesondere bei Benachrichtigungen wie "Achievement Unlocked", ist STOMP (Simple Text Oriented Messaging Protocol) die optimale Wahl. Unabhängig von der verwendeten Programmiersprache ermöglicht STOMP eine effiziente Verbindung zwischen den Komponenten einer Anwendung.

Als Datenbank haben wir uns für *PostgreSQL* entschieden. In Kombination dazu werden wir als ORM (Object-Relational Mapping) *Hibernate* einsetzen.
Hibernate ist ein leistungsstarkes Framework für Java, welches die Arbeit mit relationalen Datenbanken vereinfacht, indem es die Kommunikation zwischen Java-Objekten und der Datenbank abstrahiert.
Für das Validieren von Benutzereingaben innerhalb unseres Backends werden wir *Jakarta* verwenden. Dadurch stellen wir sicher, dass die Daten auch den erwarteten Anforderungen entsprechen.

Zum Testen der Software wird *JUnit* verwendet, welches ein weit verbreitetes Framework zur Testautomatisierung von Java-Applikationen ist. Es ermöglicht die einfache Erstellung von Unit-Tests und automatisierten Tests, um die Zuverlässigkeit des Codes sicherzustellen.

Für die Dokumentation unserer API wird *Swagger* eingesetzt. Swagger erleichtert die Erstellung von detaillierten API-Dokumentationen. Die benutzerfreundliche Oberfläche spiegelt die API-Endpunkte wider und erleichtert die Interaktion zwischen den Entwicklern.

Für den Build unserer Applikation wird *Maven* verwendet. Maven ist ein Build- und Dependencies-Management-Tool, welches Abhängigkeiten verwaltet und den Build-Prozess erleichtert.

Für die konsistente Bereitstellung unseres Backends und ihrer Abhängigkeiten in Containern verwenden wir Docker. Docker erleichtert sowohl die Entwicklung als auch die Skalierung signifikant. Zudem ermöglicht es uns, unsere Anwendung leichter zu deployen.

Architektur:

Unser Backend besteht aus mehreren Microservices. Die folgenden Microservices sind Teil unseres Backends: Authentication-Service, Database-Service, Achievement-Service, Session-Service, Cards-Service, Mail-Service und einem User-Service
Zudem existiert ein Traffic-Router, der Traffic-Router hat die Aufgabe die Socket-Connections zu verwalten und entsprechend als Relay-Station für Daten zu dienen. Über den Socket können Daten empfangen und entsprechend gesendet werden.

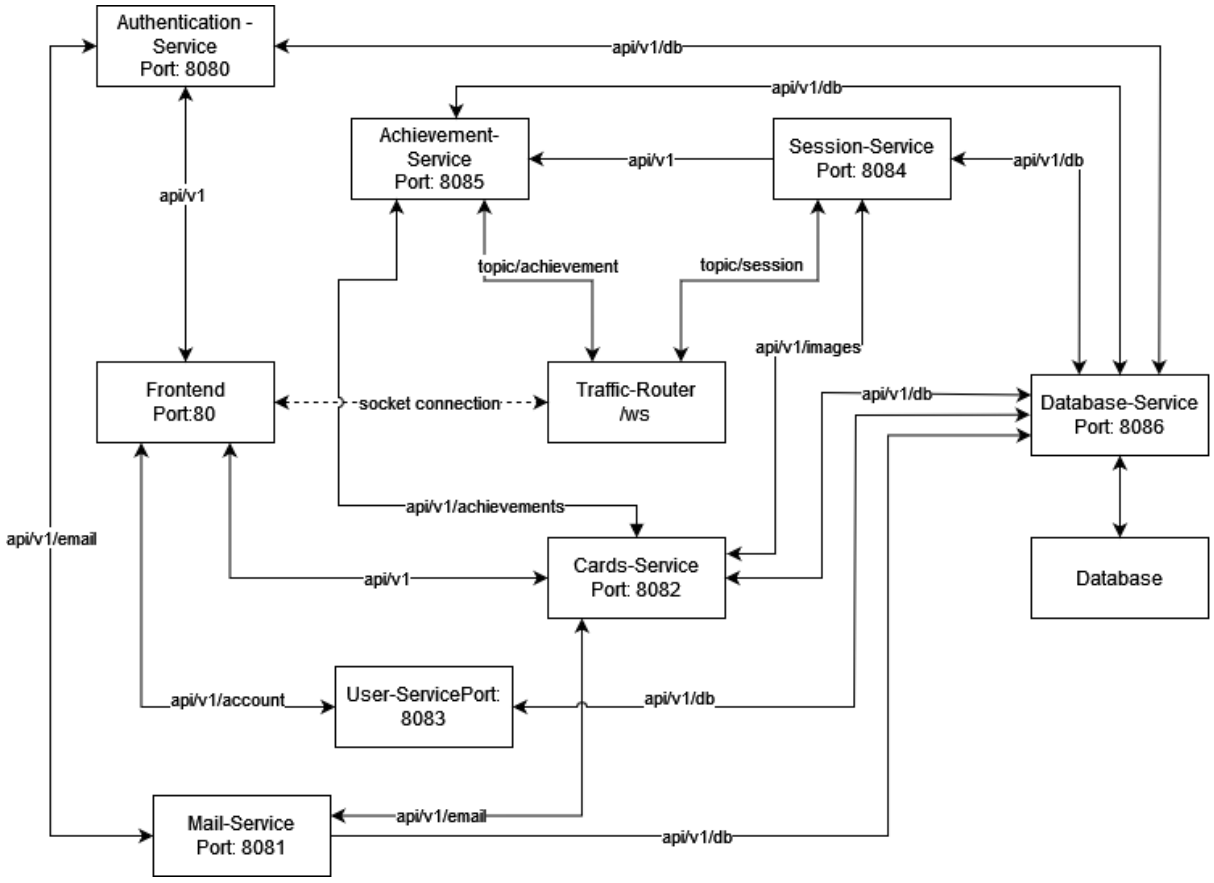

Abbildung 1: Backend-Architektur

Authentication-Service

Der Authentication-Service besitzt die Aufgabe, die Registrierung sowie Anmeldung eines Users zu ermöglichen. Ebenfalls hat dieser die Aufgabe, eine E-Mail zu verifizieren, als auch einen Passwort-Reset zu ermöglichen.

Passwort-Regularien

Unsere Passwort-Regularien richten sich nach der Empfehlung des BSI (Bundesamt für Sicherheit in der Informationstechnik)[1].
Ein Passwort muss mindestens 8 Zeichen und darf maximal 72 Zeichen lang sein, muss mindestens eine Zahl und eines der folgenden Sonderzeichen besitzen.
~ ` ! @ # $ % ^ & * ( ) _ - + = { [}] | : ; < , > . ? /
Des Weiteren muss mindestens ein Groß- und ein Kleinbuchstabe verwendet werden.
Zudem darf das Passwort nicht eine Teilmenge der '*10.000 most used Passwords*'[2] sein.

Beispielhaft invalide Passwörter:

- h3was?3spow –> Großbuchstabe fehlt
- AO$JAWOLSJ –> Kleinbuchstabe fehlt
- A3suasuwas –> Sonderzeichen fehlt
- Ts1ws –> Passwort zu kurz

Beispielhaft valide Passwörter:

- H3was?3s
- A3suas!pw?
- aS1wa3sWa

Password-Hashing

Um die Integrität und Sicherheit der Benutzerdaten zu gewährleisten, müssen Passwörter entsprechend gehasht werden. Hierbei reicht es leider nicht aus, einen einfachen Hashing-Algorithmus wie MD5 zu verwenden, da dieser nicht mehr als sicher gilt.
Um diesem Problem entgegenzuwirken, wird Bcrypt verwendet. Bcrypt verwendet intern einen Salter. Dies hat zur Folge, dass die Kodierung eines Passworts nicht reproduzierbar ist, d.h. für dasselbe Passwort wird jedesmal ein anderer Hash berechnet. Bcrypt speichert den Salt immer im Hash selbst und jeder Hash ist nach folgendem Schema aufgebaut.
Beispiel Hash: $2a$10$8rqvZdxBFgVIEqEJrGlcjuwt6lmIugKRYes1eLAac3719D2AmCzvG
Hierbei ist:
$2a die Version des Bcrypt Algorithmus
$10 die Stärke des Algorithmus
$8rqvZdxBFgVIEqEJrGlcj der random Salt
uwt6lmIugKRYes1eLAac3719D2AmCzvG das gehashte Passwort.

Hiermit ergibt sich, dass jeder Passwort-Eintrag in der Datenbank 60 Zeichen lang ist.

---

[1] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Checklisten/sichere_passwoerter_faktenblatt.pdf?__blob=publicationFile&v=4 - abgerufen am 25.10.2023
[2] https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt - abgerufen am 25.10.2023

Spring Boot ermöglicht es uns, den Algorithmus und das Salting nicht selbst zu implementieren. Hierbei stellt die Klasse *‘BCryptPasswordEncoder’ einen* Encoder bereit.
Beispiel Objekt Erstellung: ‘*PasswordEncoder pwEncoder = new BcryptPasswordEncoder()*’

Anschließend kann ein Passwort mit der Funktion *‘encode’* gehasht werden. Diese nimmt eine CharSequence des ungehashten Passwortes entgegen und gibt einen String mit dem gehashten Passwort zurück.

Wie bereits erwähnt, gestaltet sich der einfache Vergleich der beiden gehashten Passwörter durch den Random-Salt als nicht möglich. Aus diesem Grund steht die Funktion 'matches' zur Verfügung, um die Gleichheit zu überprüfen.
Diese Funktion nimmt eine Zeichenkette (CharSequence) als erstes Argument, welches das ungehashte Passwort repräsentiert. Als zweites Argument erwartet sie ein zuvor mit BCrypt gehashtes Passwort. Das Ergebnis ist ein Boolean-Wert, der angibt, ob die beiden Passwörter identisch sind oder nicht.

Login und Berechtigungen

Die Authentifizierung und Autorisierung werden mithilfe der ‘*Spring Security*’ Library realisiert. Ein bereitgestellter *'AuthenticationManager'* ermöglicht die Verwendung der '*authenticate*'-Methode.
Diese Methode erwartet ein *'UsernamePasswordAuthenticationToken'-Objekt*, das aus Benutzername und Passwort besteht. Da der Benutzername in unserem Fall nicht eindeutig ist, verwenden wir stattdessen die E-Mail-Adresse.

Die erfolgreich authentifizierte Benutzerinformation wird im *'SecurityContextHolder'* gespeichert, um die Berechtigungen innerhalb der Anwendung zu unterscheiden. Anschließend implementieren wir die Klasse *'UserDetailsImpl'*, die das Interface *'UserDetails'* implementiert. Hier werden die Benutzerdaten sowie die entsprechenden Berechtigungen festgehalten. Da unsere Applikation sich auf die Benutzerrolle 'User' beschränkt, setzen wir einen entsprechenden Default-Wert für die Berechtigung.

Danach wird aus den Benutzerdetails ein JWT (JSON Web Token) generiert und dem Benutzer in einem Cookie bereitgestellt. Dies ermöglicht es dem Benutzer, sich nicht bei jedem Besuch der Seite erneut anmelden zu müssen. Gleichzeitig fungiert der Token als Datenprovider für Benutzerinformationen wie die Benutzer-ID und die E-Mail-Adresse.

Der JWT wird bei jeder Request an einen Microservice mitgesendet und auf Gültigkeit überprüft. Dadurch wird sichergestellt, dass nur autorisierte Benutzer Zugriff auf unsere Services haben. Falls der JWT nicht gültig sein sollte, wird der HTTP-Statuscode 401 (Unauthenticated) zurückgegeben.

API-Endpoints

Im Folgenden werden die Endpunkte für den Authentication-Service definiert.
Wichtig: Die Routen 'api/v1/register', 'api/v1/login', 'api/v1/email/verify' und 'api/v1/password/reset' sind ohne Authentifizierung erreichbar.

Es wird eine POST-Route unter *'api/v1/register'* bereitgestellt. Bei erfolgreicher Registrierung wird der HTTP-Statuscode 201 (Created) zurückgegeben. Sollte bereits ein Konto mit der angegebenen E-Mail existieren, wird der HTTP-Statuscode 409 (Conflict) zurückgegeben. Wenn das Passwort in einer Rainbow-Tabelle gefunden wird oder nicht den festgelegten Standards entspricht, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Tritt ein anderer Fehler auf, beispielsweise wenn der Authentication-Service den Database-Service für die entsprechenden Einträge nicht erreichen kann, wird der HTTP-Statuscode 500 (Internal Server Error) zurückgegeben.

Es wird eine POST-Route unter *'api/v1/login'* bereitgestellt. Bei erfolgreicher Anmeldung wird der HTTP-Statuscode 200 (OK) und ein JWT-Token als Cookie zurückgegeben. Falls die angegebenen Anmeldeinformationen (E-Mail und Passwort) nicht in der Datenbank gefunden oder falsch eingegeben wurden, wird der HTTP-Statuscode 401 (Unauthorized) zurückgegeben. Dies hat den Vorteil, dass die Information maskiert wird, dass ein Konto mit diesen Anmeldeinformationen (speziell der E-Mail-Adresse) existiert. Andernfalls könnte der User Informationen erhalten, zu denen er keine Berechtigung hat.

Des Weiteren wird eine PUT-Route unter *'api/v1/email/verify'* bereitgestellt. Diese Route wird für die E-Mail Verifikation genutzt. Bei erfolgreicher Verifikation wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Falls der Token invalide oder abgelaufen sein sollte, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Der HTTP-Statuscode 409 (Conflict) wird zurückgegeben, falls die E-Mail bereits verifiziert worden ist.

Passwort vergessen:
Für das Versenden der E-Mail zum Zurücksetzen des Passworts wird eine POST-Route verwendet, welche unter 'api/v1/password/reset' erreichbar ist. Die Route ist eine POST-Route, da ein neuer PASSWORD_RESET Token in der Datenbank hinzugefügt werden muss. Es wird immer der HTTP-Statuscode 202 (Accepted) zurückgegeben, um zu maskieren, dass diese E-Mail existieren könnte.

In der E-Mail zum Passwort-Reset, welche der User, beim erfolgreichen Angeben seiner E-Mail, bekommt, ist ein Button zu der statischen GET Route: 'password/reset' führt. In den Query-Parametern sind die E-Mail-Adresse sowie der Reset-Token enthalten. Da dies in der fertigen Anwendung (PROD) eine HTTPS Route ist, sind diese ebenfalls verschlüsselt.

Auf der Seite, auf die der Benutzer durch den Button weitergeleitet wird, findet dieser zwei Passwortfelder und einen "Absenden"-Button. Hier kann er zweimal das neue Passwort eingeben. Sollte er nicht das gleiche Passwort eingegeben haben, so ist der "Absenden"-Button ausgegraut. Beim Drücken des "Absenden"-Buttons wird an die PUT-Route 'api/v1/password/reset' das neue Passwort gesendet. Beim Abschicken des neuen Passworts werden die E-Mail und der Reset-Token aus den Query-Parametern ebenfalls mitgesendet.

Ebenso wird eine PUT-Route unter ‘api/v1/password/reset’ bereitgestellt. Diese Route wird für das Zurücksetzen des Passworts verwendet. Hierbei werden im Header ein Token und die E-Mail erwartet und im Body ein neues Passwort. Für den Fall, dass der Token nicht existiert, wird der Statuscode 400 (Bad Request) zurückgegeben. Selbiges gilt, falls das Passwort nicht den festgelegten Standards entspricht oder sich in der Rainbow-Table befindet. Im Falle eines Erfolges wird der HTTP-Statuscode 204 (No Content) zurückgegeben.

Passwort Aktualisieren (Profil):
Für das Aktualisieren eines Passworts wird eine PUT-Route unter ‘api/v1/password’ bereitgestellt. Diese Route wird für das Aktualisieren des Passworts verwendet. Im Header wird ein JWT Token für die Verifikation des Users erwartet. Im Body lediglich das neue Passwort. Für den Fall, dass der Token nicht valide ist oder nicht existiert, wird der HTTP-Statuscode 401 (Unauthorized) zurückgegeben. Falls das Passwort den Standards nicht entspricht oder sich in der Rainbow-Table befindet, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben, andernfalls der HTTP-Statuscode 204 (No Content).

User-Service

Der User-Service ist verantwortlich für die Verwaltung der Profilansicht. Dabei umfassen seine Funktionalitäten das Aktualisieren der Benutzerinformationen sowie das allgemeine Abrufen der Benutzerdaten, wenn die Profilansicht geöffnet wird.

API-Endpoints

Wichtig: Die Route ‘api/v1/account’ benötigt eine entsprechende Authentifizierung!

Unter der GET-Route 'api/v1/account' können die Benutzerinformationen abgefragt werden.
Die ID, die abgefragt werden soll, wird dem JWT Token entnommen.
Die angefragten Informationen beinhalten den aktuellen Benutzernamen, die aktuelle E-Mail-Adresse, die Anzahl der Karten, die pro Session gelernt werden, einen Boolean-Wert, der angibt, ob Lernbenachrichtigungen empfangen werden sollen oder nicht, sowie die aktuell ausgewählte Sprache für die UI und eine Liste an IDs für die Achievements. Für den Fall, dass die Benutzerinformationen erfolgreich abgefragt werden können, wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls kein User unter der ID gefunden wurde, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Unter der PUT-Route ‘api/v1/account’ können Benutzerinformationen entsprechend geupdatet werden. Die ID, die abgefragt werden soll, wird dem JWT Token entnommen.
Die Route wird getriggert, falls der Benutzer im Frontend auf den ‘save’-Button drückt. Hierbei wird bei Erfolg der HTTP-Statuscode 200 (OK) zurückgegeben sowie die entsprechenden Daten, um diese mit dem Frontend zu synchronisieren. Falls die Daten ein falsches Format haben sollten oder invalide Informationen enthalten, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Cards-Service

Der Cards-Service ist verantwortlich für das Verwalten von Decks. Dazu gehören das Erstellen, Löschen, Bearbeiten, Importieren, Exportieren und Teilen von Decks. Des Weiteren kümmert er sich um die Anzeige der History. Zusätzlich ermöglicht er das Anzeigen, Hinzufügen, Aktualisieren und Löschen von Cards im Bearbeitungsbereich.

API-Endpoints

Wichtig: Alle Routen dieses Services benötigen eine entsprechende Authentifizierung!

Die GET-Route ‘api/v1/decks’ gibt ein Array an Namen sowie die Anzahl an zu lernenden Karten für die Decks zurück. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls keine Deckeinträge existieren sollten, wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route ‘api/v1/decks/{id}’ gibt die detaillierten Informationen eines Decks zurück. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Zudem wird als Path-Variable die ID des Decks übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die POST-Route ‘api/v1/decks’ ist für das Erstellen eines neuen Decks da. Diese nimmt im Body ein JSON-Objekt mit einem Decknamen entgegen. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben. Falls der Name nicht den Standards (Länge) erfüllt, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die PUT-Route ‘api/v1/decks/{id}’ ist für das Aktualisieren der Informationen eines Decks da. Die Route nimmt im Body ein JSON-Objekt mit einem neuen Decknamen entgegen. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben und als Path-Variable wird die ID des Decks übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls der Name nicht den Standards (Länge) erfüllt, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die DELETE-Route ‘api/v1/decks/{deckId}’ ist für das Löschen des Decks und der dazugehörigen Karten da. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben und als Path-Variable wird die ID des Decks übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die POST-Route ‘api/v1/decks/import’ ist für das Importieren eines Decks da. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Die .zip File wird als ‘Multipartfile’ als Request-Parameter übergeben. Diese wird dann anschließend vom Server verarbeitet.
Falls Bilder existieren, werden diese auf dem Server abgelegt und die JSON File wird entsprechend in die Datenbank migriert. Für den Fall, dass die Route erfolgreich ist, wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Für den Fall, dass die .zip Datei nicht lesbar oder malformed ist, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route ‘api/v1/decks/{deckId}/export’ ist für das Exportieren eines Decks da. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Die .zip-File wird als Ressource zurückgegeben. Für den Fall, dass die Route erfolgreich ist, wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die POST-Route 'api/v1/decks/{deckid}/share' ist für das Teilen eines Decks da. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Man übergibt die Deck-ID, sowie die E-Mail des zu teilenden Users im Body. Nach dem Abschicken der Request erstellt das Backend einen Token vom Typ “Share”. Dieser hat als Präfix die ID des Decks, gefolgt von einem Minus. Dieser Token wird in der Datenbank entsprechend abgespeichert. Ein Beispiel Token sieht wie folgt aus: “4-UUID”. Das Backend sendet daraufhin eine E-Mail an die angegebene E-Mail, um den Benutzer darüber zu informieren, dass ein User sein Deck teilen will. In der Mail kann der User dann auf den bestätigen Button drücken. Dieser leitet ihn zuerst auf die statische Backend-Route ‘api/v1/decks/share/{share-token}’ weiter. Das Backend überprüft daraufhin, ob der Token gültig ist. Falls der Token nicht existiert, wird ein 404 zurückgegeben. Ansonsten wird ein 301 (Moved Permanently) zurückgegeben und der Benutzer wird dann auf eine statische Seite weitergeleitet, wo er darüber informiert wird, dass das Deck nun in deiner Deckliste zu finden ist. Durch das Aufrufen der Backend-Route weiß der Service, dass die ‘Share’-Anfrage akzeptiert wurde. Das Backend erstellt daraufhin eine exakte Kopie des Decks in der Datenbank mit Besitzerrechten für den User.

Die GET-Route ‘api/v1/historys’ gibt eine Liste von Historys zurück, mit Startzeit, Endzeit, ob diese Session abgeschlossen wurde und wie viele Karten gelernt wurden, sowie eine ID, die diesen Eintrag kennzeichnet. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls keine History-Einträge existieren sollten, wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die POST-Route ‘api/v1/historys’ ist für das Erstellen eines neuen History-Eintrages da. Diese Route wird ausschließlich vom Session-Service aufgerufen. Die Route nimmt im Body ein JSON-Objekt mit den entsprechenden Informationen entgegen. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben. Falls der Eintrag invalide Daten enthalten sollte, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route ‘api/v1/historys/{id}’ gibt die detaillierten Informationen einer Session zurück. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Zudem wird als Path-Variable die ID der Session übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die ID der Session nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route ‘api/v1/decks/{id}/cards’ gibt alle Fragen eines Kartendecks zurück. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Zudem wird als Path-Variable die ID des Decks übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ gibt die detaillierten Informationen einer Karte zurück. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Zudem wird als Path-Variable die ID des Decks übergeben sowie die ID einer Karte. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die ID des Decks oder die ID für die Karte nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die POST-Route ‘api/v1/decks/{deckId}/cards’ ist für das Erstellen einer neuen Karte innerhalb eines Decks da. Die Route nimmt im Body ein JSON-Objekt mit den entsprechenden Informationen entgegen. Zudem wird als Path-Variable die ID des Decks übergeben. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben und die ID des Eintrags, diese wird für das Speichern der Images verwendet, sofern welche existieren sollten. Falls der Eintrag invalide Daten enthalten sollte, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die PUT-Route ‘api/v1/decks/{deckId}/cards/{cardId}’ ist für das Aktualisieren der Informationen einer Karte da. Zudem wird als Path-Variable die ID des Decks übergeben sowie die ID einer Karte. Die Route nimmt im Body ein JSON-Objekt mit den neuen Informationen einer Karte. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben und als Path-Variable wird die ID des Decks übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die Daten fehlerhaft sein sollten, wird der HTTP-Statuscode 400 (Bad Request) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die DELETE-Route ‘api/v1/decks/deckId}/cards/{cardsId}’ ist für das Löschen einer Karte innerhalb eines Decks da. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben und als Path-Variable wird die ID des Decks und die ID einer Karte übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized).

Die GET-Route ‘api/v1/images/{imageId}’ ist für das Ziehen eines Bildes. Der JWT-Token, der für die Authentifizierung benötigt wird, wird im Header übergeben. Die ImageId wird als Path Variable übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized).

Die POST-Route ‘api/v1/images’ ist für das Speichern eines Bildes. Der JWT-Token, der für die Authentifizierung benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized).

Die DELETE-Route ‘api/v1/images/{imageId}’ ist für das Löschen eines Bildes. Der JWT-Token, der für die Authentifizierung benötigt wird, wird im Header übergeben. Die ImageId wird als Path Variable übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Falls die ID des Decks nicht existieren sollte, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized).

Traffic-Router

Die Aufgabe des Traffic-Routers ist es zum einen eine Socket-Connection zum Frontend aufzubauen und Anfragen an- und von dem Session-Service und Achievement-Service zu delegieren.
Der Socket wird unter ‘www.example.de:XXXX/ws’ erreichbar sein.

Der Traffic-Router wird eine HashMap mit einer UserSession-Klasse besitzen. Diese beinhaltet zum einen die User-ID und die Session-ID, dies ist notwendig, um anschließend zu wissen, an welchen Socket welche Nachricht gesendet werden muss.

Socket-Endpoints

Wenn die Socket-Route ‘/connect’ aufgerufen wird, wird die UserID als Payload übergeben, damit anschließend ein entsprechender Eintrag in die HashMap hinzugefügt werden kann.

Wenn die Socket-Route ‘/disconnect’ aufgerufen wird, dann wird der Eintrag mit der entsprechenden UserID in der HashMap gelöscht.

Die GET-Route ‘api/v1/users/{userId}/achievements/{achievementId}’ wird ausschließlich vom Achievement-Service getriggert. Dieser sendet dem Frontend unter ‘topic/achievement/notification’ eine Notification mit der entsprechenden AchievementID. Anschließend kann sich das Frontend die benötigten Informationen selbst anfragen über die Routen ‘api/v1/achievement/{achievementId}/image ‘
und ‘api/v1/achievement/{achievementId}/data.

Peek-Function
Wenn das Frontend den Socket-Endpunkt ‘topic/sessions/peek/start’ aufruft, wird eine HTTP-Anfrage an den Session-Service mit der POST-Route ‘api/v1/sessions/peek’ geschickt, zum initialisieren der Peek-Session.

Wenn das Frontend den Socket-Endpunkt 'topic/sessions/peek/get-card' aufruft, wird eine HTTP-Anfrage an den Session-Service mit der GET-Route 'api/v1/sessions/peek/{peekSessionID}/next' gesendet um eine Random Karte aus dem Kartendeck zu bekommen. Anschließend wird die Karte über den Socket an das Frontend zurückgegeben.

Wenn das Frontend den Socket-Endpunkt ’topic/sessions/peek/next’ aufruft, wird eine HTTP-Anfrage an den Session-Service mit der POST-Route ’api/v1/sessions/peek/{peekSessionID}/cards/{cardID}’ gestellt, um abzuspeichern, dass diese Karte erfolgreich gesehen worden ist.

Wenn das Frontend den Socket-Endpunkt ‘topic/sessions/peek/stop’ aufruft, wird eine HTTP-Anfrage an den Session-Service mit der PUT-Route ‘api/v1/sessions/peek/{peekSessionID}’ gestellt, um den Timestamp abzuspeichern, bis wann gelernt worden ist.

Learn-Function
Wenn das Frontend den Socket-Endpunkt ‘topic/sessions/learn/start’ aufruft, wird eine HTTP-Anfrage an den Session-Service mit der POST-Route ‘api/v1/sessions/learn’ geschickt, zum initialisieren der Learn-Session.

Wenn das Frontend den Socket-Endpunkt 'topic/sessions/learn/get-card' aufruft, wird eine HTTP-Anfrage an den Session-Service mit der GET-Route 'api/v1/sessions/learn’/{learnSessionID}/next' gesendet. Anschließend wird die Karte über den Socket an das Frontend zurückgegeben.

Wenn das Frontend den Socket-Endpunkt ‘’topic/sessions/learn/next’ aufruft, wird eine HTTP an den Session-Service mit der PUT-Route ’api/v1/sessions/learn/{learnSessionID}’. Es wird gesendet, welche Bewertung der User der Karte gegeben hat, um das Intervall bis zum nächsten Anzeigen entsprechend anpassen zu können.

Wenn das Frontend den Socket-Endpunkt ‘topic/sessions/learn/stop’ aufruft, wird eine HTTP-Anfrage an den Session-Service mit der PUT-Route ‘api/v1/sessions/learn/{learnSessionID}’ gestellt, um den Timestamp abzuspeichern, bis wann gelernt worden ist.

Achievement-Service

Der Achievement-Service verwaltet die ganzen Bedingungen für Achievements und hält fest, wann ein Achievement erreicht wird. Es hat die Aufgabe zu überprüfen, ob ein neues Achievement erreicht wurde. Falls dies der Fall ist, wird die GET-Route ‘api/v1/users/{userId}/achievements/{achievementId}’ gecallt, um dem Frontend die entsprechende Information zu providen.

API-Endpoints

Wichtig: Die Route ‘api/v1/achievements/categories/{category-name}’ benötigt eine *entsprechende Authentifizierung!*

Die POST-Route ‘api/v1/achievements/categories/{category-name}’ hat die Aufgabe, eine bestimmte Kategorie an Achievements zu überprüfen, ob diese erfüllt sind oder nicht. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Für den Fall, dass ein Achievement in der Kategorie ‘fertiggestellt’ wurde, wird ein entsprechender Eintrag in der Datenbank getätigt und es wird der HTTP-Statuscode 201 (Created) zurückgegeben. Sollte die Kategorie nicht existieren, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Falls der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Beispiel: Die Kategorie Lernsession absolviert würde unter der Route ‘api/v1/achievements/categories/finished-session’ überprüft werden. Hierbei wird überprüft, ob die Anzahl an Lern-Sessions, die abgeschlossen wurden >= der Anzahl für das Achievement ist und sich noch nicht in den Achievements des Users befindet. Falls dies der Fall ist, wird ein entsprechender Eintrag in der Datenbank vorgenommen und die GET-Route ‘api/v1/users/{userId}/achievements/{achievementId}’ aufgerufen.

Unter der GET-Route ‘api/v1/achievement/{id}/image’ wird das Icon für ein entsprechendes Achievement bereitgestellt. Bei Erfolg wird der HTTP-Statuscode 200 (OK) mit einem Byte-Array zurückgegeben. Falls die Ressource unter dieser ID nicht gefunden wurde, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben.

Unter der GET-Route ‘api/v1/achievement/{id}/data werden die Informationen für ein entsprechendes Achievement bereitgestellt. Bei Erfolg wird der HTTP-Statuscode 200 (OK) mit den entsprechenden Informationen zurückgegeben. Falls die Ressource unter dieser ID nicht gefunden wurde, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben.

Session-Service

Der Session-Service hat zwei Aufgaben: die Lern-Session-Funktion und die Peek-Funktion zu verwalten sowie den gelernten Karten, anhand des Spaced-Repetition-Algorithmus, ein neues Intervall zuzuweisen, wann die Karte erneut angezeigt werden soll.

API-Endpoints

Wichtig: Alle Routen dieses Services benötigen eine entsprechende Authentifizierung!

Peek-Function

Die POST-Route ‘api/v1/sessions/peek’ hat die Aufgabe, eine Peek-Session zu initialisieren. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route 'api/v1/sessions/peek/{peekSessionID}/next' hat die Aufgabe, eine Random-Karte, bzw. in Java Pseudo-Random, zurückzugeben. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Diese kann vom Typ Multiple-Choice oder Basic sein. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls keine nächste Karte existiert, da z.B. in dieser Peek-Session schon alle Karten gelernt wurden, oder das Deck leer ist, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die POST-Route ’api/v1/sessions/peek/{peekSessionID}/cards/{cardID}’ hat die Aufgabe die Karte in die m:n Tabelle zwischen Peek-Session und Cards hinzuzufügen. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben. Falls die PeekSessionID oder die CardID nicht gefunden werden konnten, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die PUT-Route ‘api/v1/sessions/peek/{peekSessionID}’ besitzt die Aufgabe, eine Peek-Session als abgeschlossen zu markieren und um den entsprechenden Timestamp festzuhalten, bis wann gelernt wurde. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Falls die PeekSessionID nicht gefunden werden konnten, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Learn-Function
Die POST-Route ‘api/v1/sessions/learn’ hat die Aufgabe, eine Lern-Session zu starten.
Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 201 (Created) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die PUT-Route ‘api/v1/sessions/learn/{learnSessionID}’ besitzt die Aufgabe, eine Lern-Session zu aktualisieren, z.B. die Einschätzung des Users zu verändern oder den Status auf abgeschlossen zu setzen. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Bei Erfolg der Route wird der HTTP-Statuscode 204 (No Content) zurückgegeben. Falls die learnSessionID nicht gefunden werden konnten, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Die GET-Route 'api/v1/sessions/learn’/{learnSessionID}/next' hat die Aufgabe, die am längsten abgelaufene Karte zu präsentieren. Eine Karte gilt als abgelaufen, wenn der Zeitpunkt, an dem die Karte gelernt wurde, plus das Intervall vom Spaced-Repetition-Algorithmus in der Vergangenheit liegt. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Diese kann vom Typ Multiple-Choice oder Basic sein. Bei Erfolg der Route wird der HTTP-Statuscode 200 (OK) zurückgegeben. Falls keine nächste Karte existiert, da z.B. in dieser Lern-Session schon alle Karten gelernt wurden, oder das Deck leer ist, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Für den Fall, dass der User eine Anfrage auf die Route stellt, ohne authentifiziert zu sein, wird der HTTP-Statuscode 401 (Unauthorized) wiedergegeben.

Mail-Service

Der Mail-Service wird verwendet, um Mails zu versenden. Hier drunter fallen die Mails, für die Passwort-vergessen Funktionalität, für die tägliche Lern-Benachrichtigung sowie für die Mail vom Deck-Share. Dieser sendet auch bei der ersten Registrierung die Verifikations-E-Mail.

Versenden von E-Mails

Für das Versenden von E-Mails kommt die *JavaMail-API* zum Einsatz. Diese ermöglicht es uns, Mails via SMTP (Simple Mail Transfer Protocol) zu versenden und über POP3 oder IMAP zu empfangen. Wir benötigen jedoch lediglich die Option zum Versenden von Mails.
Die JavaMail-API stellt einen '*JavaMailSender*' zur Verfügung. Dieser verfügt über die Methode '*send*', zum Versenden einer Mail. Die Methode nimmt eine Message entgegen, bspw. ein Objekt der Klasse '*MimeMessage*'. Da wir jedoch keinen Response-Body bekommen, ob der User die Mail empfangen hat oder nicht, muss die Funktionalität zum erneuten Versenden einer Mail bereitgestellt werden.

Die Konfiguration eines Mail-Senders kann Beispielhaft wie folgt aussehen. Hierbei wird unsere E-Mail für das SWTP Projekt "softwaretechnikprojekt@gmail.com" verwendet.

```java
@Bean
public JavaMailSender mailSender() {
    JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
    mailSender.setHost("smtp.gmail.com");
    mailSender.setPort(587);

    mailSender.setUsername("softwaretechnikprojekt@gmail.com");
    mailSender.setPassword("SOME_PASSWORD");

    Properties props = mailSender.getJavaMailProperties();
    props.put("mail.transport.protocol", "smtp");
    props.put("mail.smtp.auth", "true");
    props.put("mail.smtp.starttls.enable", "true");
    props.put("mail.debug", "true");

    return mailSender;
}
```

Abbildung 2: Konfiguration des JavaMailSenders

Cronjob Tägliche E-Mail Notifikation

Die Ausdruck *cron = "0 0 16 * * ?"* in der Spring *@Scheduled* Annotation plant eine Aufgabe zur Ausführung. In diesem Fall bedeutet es, dass die Aufgabe jeden Tag um 16 Uhr nachmittags startet. Die sechs Felder in der Cron-Ausdruck repräsentieren Sekunden, Minuten, Stunden, Tag im Monat, Monat und Tag in der Woche, und können angepasst werden, um präzise Zeitpläne zu definieren. Hierbei wird beispielhaft die E-Mail täglich um 16 Uhr versendet. Die @Scheduled Annotation funktioniert jedoch nur, wenn die @SpringBootApplication Komponente zusätzlich mit @EnableScheduling annotiert wurde.

API-Endpoints

Wichtig: Alle Routen dieses Services benötigen eine entsprechende Authentifizierung!

Die POST-Route ‘api/v1/email/{mailType}’ hat die Aufgabe, eine entsprechende Mail zu versenden. Der mailType ist bspw. Verifikation, woraufhin an die im Body angegebene E-Mail Adresse eine E-Mail gesendet wird. Der JWT Token, der für die Account-ID benötigt wird, wird im Header übergeben. Im Falle, dass die Mail erfolgreich versendet worden ist, wird der HTTP-Statuscode 202 (Accepted) zurückgegeben. Für den Fall, dass der mailType nicht valide ist, wird der HTTP-Statuscode 404 (Not Found) zurückgegeben. Falls die Mail aufgrund eines internen Fehlers nicht verschickt werden kann, dann wird der HTTP-Statuscode 500 zurückgegeben.

Database-Service

Der Database-Service stellt die entscheidende Schnittstelle zwischen den verschiedenen Services einer Anwendung und der zugrundeliegenden Datenbank dar. Diese Komponente ermöglicht es, Daten aus der Datenbank zu lesen und auch in die Datenbank zu schreiben. Dabei nutzt der Database-Service als Frameworks Hibernate und JPA.

Hibernate ist ein ORM (Object-Relational-Mapping) Framework für Java. Es spielt eine zentrale Rolle in der Interaktion zwischen der Anwendungslogik und der Datenbank. Hierbei werden Entitätsklassen erstellt, die als Java-Abbildungen der Datenbank-Tabellen dienen.

Es besitzt eine sogenannte SessionFactory, die als zentrale Schnittstelle zur Datenbank fungiert. Die SessionFactory verwaltet Sessions und ermöglicht das Speichern, Aktualisieren und Abrufen von Daten. Transaktionen werden sorgfältig gesteuert, um die Datenintegrität zu gewährleisten. Zudem bietet Hibernate eine eigene Abfragesprache namens HQL (Hibernate Query Language), die es ermöglicht, Abfragen in einer objektorientierten Art und Weise zu formulieren.

JPA, also die Java Persistence API, ist eine Spezifikation für die Verwaltung persistenter Objekte in relationalen Datenbanken. Hibernate implementiert diese Spezifikation. Anhand von JPA werden Standardmethoden bereitgestellt, um Entitäten zu erstellen, zu ändern und abzufragen.

Verbindung zur Datenbank

In Spring Boot werden in der ‘application.properties’-Datei die entsprechenden Verbindungsdaten zur Datenbank hinterlegt. Hierbei müssen Informationen wie die URL, der Benutzername, das Passwort und der Datenbanktreiber angegeben werden. Der Datenbanken-User muss die jeweiligen Berechtigungen für die Datenbank sowie für die Entitäten besitzen.

```
1  spring.datasource.url=${DB_URL}/${DB_NAME}
2  spring.datasource.username=${DB_USERNAME}
3  spring.datasource.password=${DB_PASSWORD}
4  spring.datasource.driver-class-name=org.postgresql.Driver
5  spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
```

Abbildung 3: Beispiel ‘application.properties’ mit den oben definierten Informationen

Da die ‘application.properties’-Datei nicht sicher ist für sensitive Daten, sind diese Informationen entsprechend in eine ‘.env’-Datei ausgelagert.

API-Endpoints

Wichtig: Alle Routen dieses Services benötigen eine entsprechende Authentifizierung!

Hier werden alle Routen zur Verfügung gestellt, die von den anderen Services (Mail-Service, Cards-Service, User-Service, Achievement-Service, Session-Service, Authentication-Service) verwendet werden.

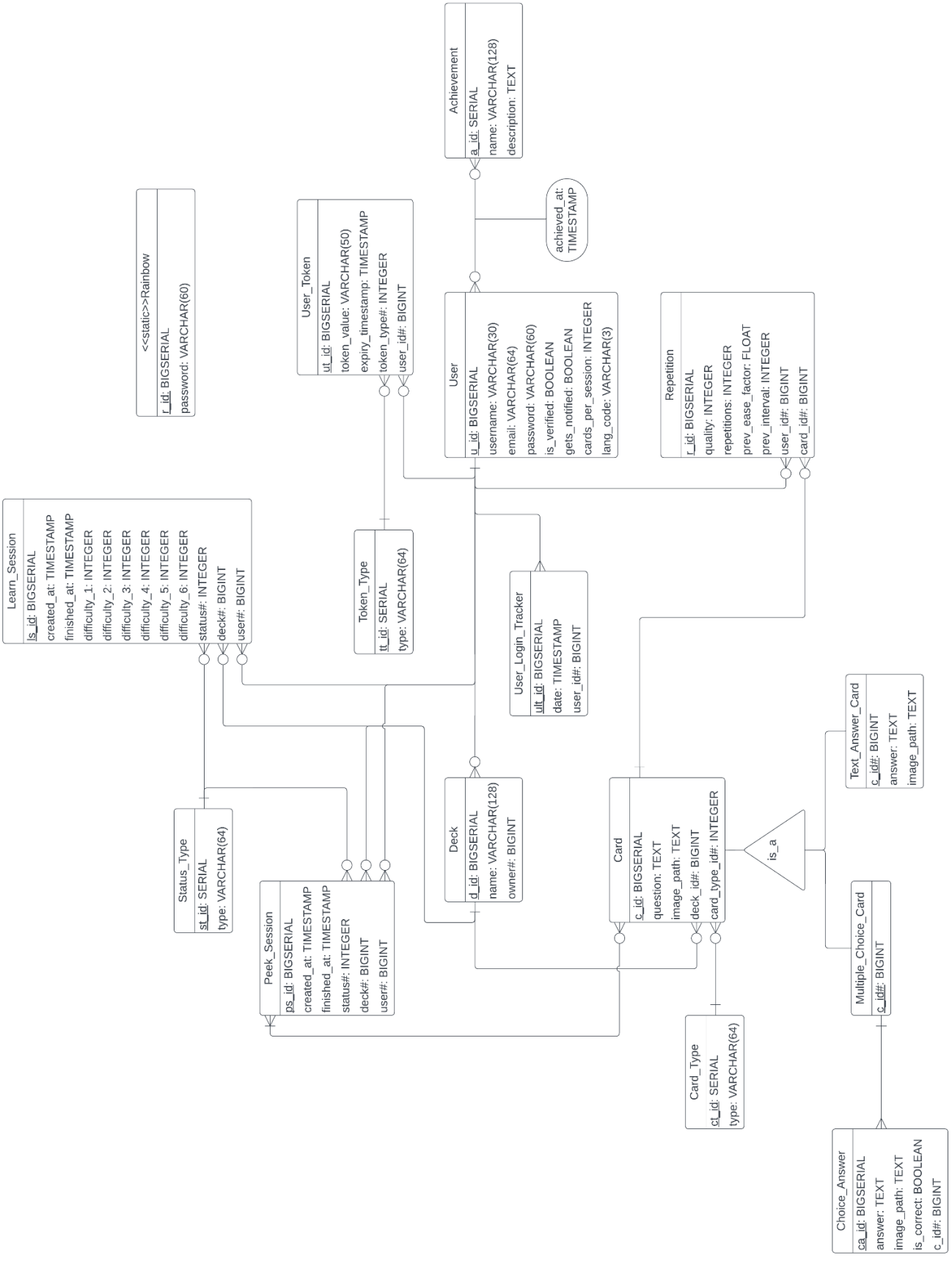

Abbildung 4: Database-Schema

Klassenmodell

Getter und Setter sind entsprechend vorhanden, wurden aber aus Vereinfachungsgründen nicht visualisiert.

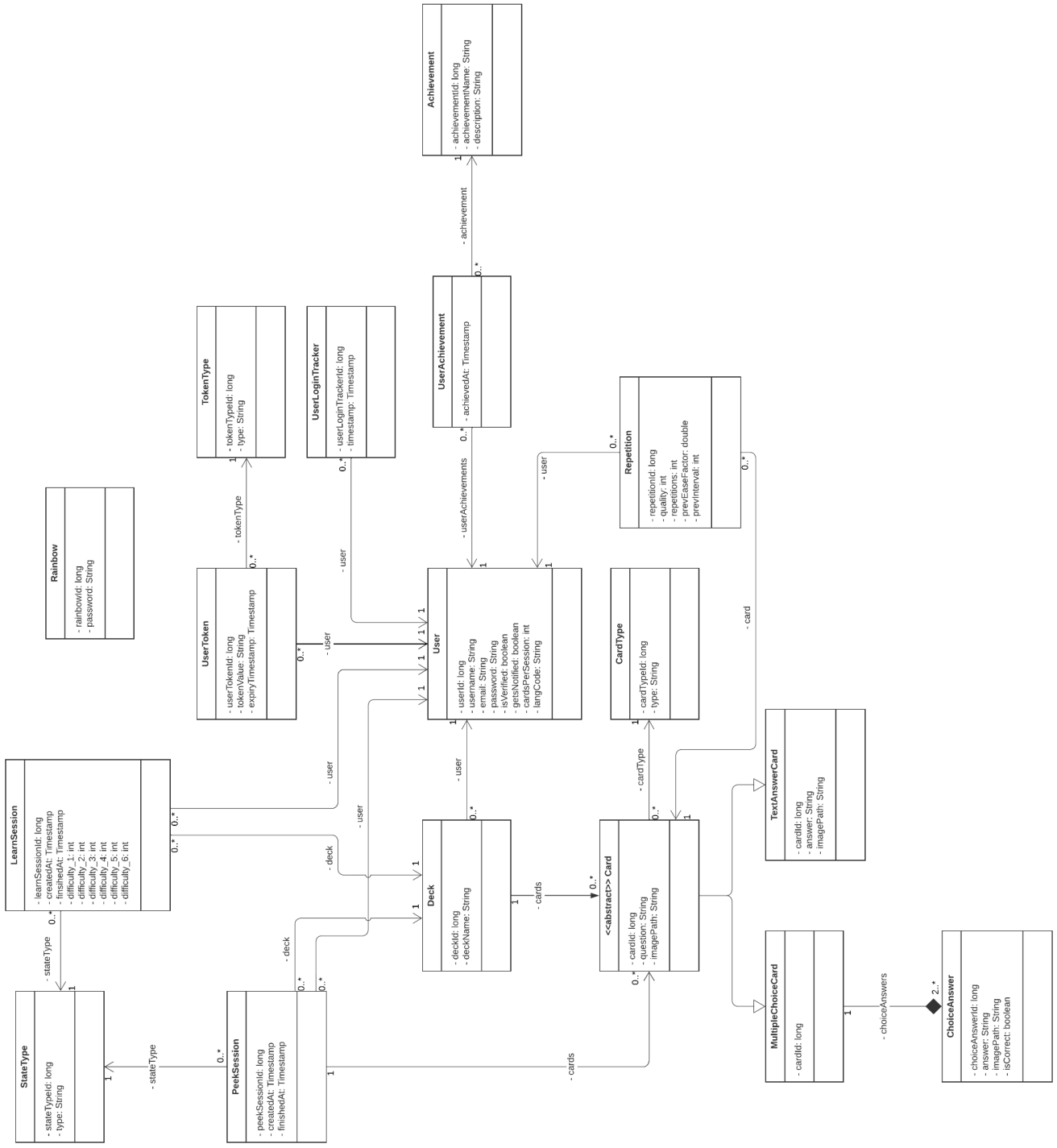

Abbildung 5: Klassendiagramm

Frontend

Techstack

Die HTML-Datei bildet das Grundgerüst. Mit ihr können Buttons, Textfelder, Texte und vieles mehr hinzugefügt werden, um eine einfache, aber klare Seite zu erstellen. In dieser HTML-Datei können TypeScript/JavaScript-Dateien, CSS-Dateien und auch Bilder eingebunden werden, um die Seite weiter zu gestalten.

Die CSS-Datei ist maßgeblich für das Design der Seite verantwortlich. Sie ermöglicht es, das Erscheinungsbild der Website zu gestalten, indem sie Funktionen wie das Anpassen des Hintergrunds oder die Positionierung von Elementen in der HTML-Struktur bereitstellt und vieles mehr.

Die verwendete Programmiersprache für unser Frontend ist TypeScript. Die TypeScriptdatei wird zu einer Javascriptdatei kompiliert und in die HTML eingebunden. TypeScript ist im Vergleich zu JavaScript leichter zu verwenden und bietet auch neue Funktionen. Es erleichtert uns die Erstellung des Frontends. Sie wird außerdem noch dafür verwendet, um Kontakt mit dem Backend zu haben und bei Aktionen, wie bspw. einem Knopfdruck die passenden Informationen zu laden bzw. vom Backend zu erfragen.

Für das Design wird das Bootstrap Framework verwendet. Bootstrap ermöglicht es uns, relativ einfach ein Responsive Design zu bauen, welches sowohl auf dem Handy als auch auf größeren Geräten ansprechend aussieht. Zudem ist es mit allen, zum Zeitpunkt existierenden, Browsern kompatibel.

Angular wird als weiteres Framework für die Struktur sowie auch Ordnung des Frontends verwendet. Sie ermöglicht es, eine einzige HTML-Datei für die gesamte Software/Website zu verwenden und je nach Situation die entsprechenden Komponenten zu laden. Da die Website als SPA (Single Page Application) angeboten wird, muss nicht dauerhaft die gesamte Seite neu geladen werden.

Hinzu verwenden wir noch STOMP, welches Socket-Based Kommunikation ermöglicht.
Sockets werden dazu verwendet, um den User Live-Benachrichtigungen/Updates zu bekommen. Der Socket wird geöffnet, sobald der User unsere Webseite betritt.

Zur Installation und Bereitstellung von verschiedenen Packages verwenden wir npm, welches ein Package-Manager ist, der es uns ermöglicht, schnell Packages explizit für JavaScript herunterzuladen und zu verwenden.

Grundstruktur Komponenten

Jede Seite, auf welcher der User eingeloggt ist, folgt einer festen Komponentenstruktur.

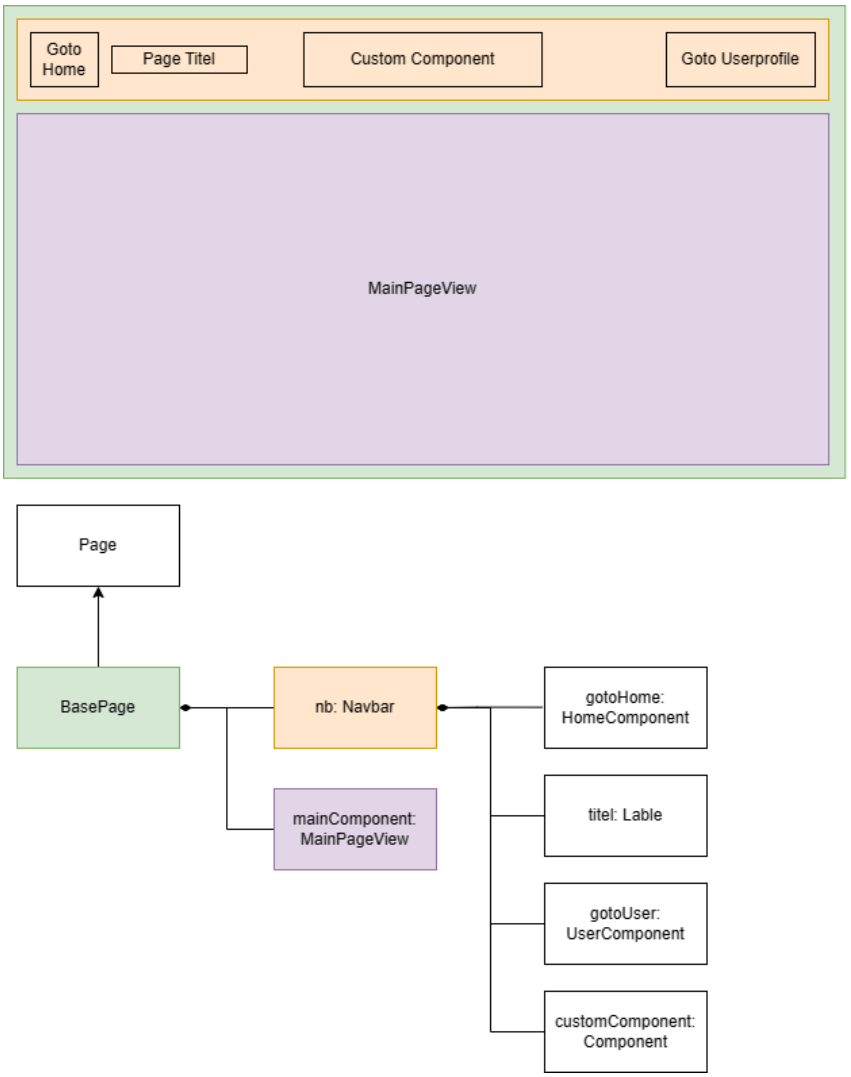

Abbildung 6: Komponentenstruktur

Die Base-Seite ist immer aufgeteilt in die Navbar und dem MainPageView-Content. Die Navbar hat von links nach rechts folgende Komponenten:

- Den ‘GoHome’-Button welcher den Benutzer immer zurück zur Deckview schickt.
- Ein Label, welches den Titel der momentanen Seite anzeigt.
- (Optional) Eine Komponente, welche als Parameter an die Navbar übergeben werden kann, damit man zum Beispiel den bearbeitbaren Titel eines Decks in der Edit-Deckansicht anzeigen kann.
- In der oberen rechten Ecke gibt es einen ‘Go to User Profile’-Button.

In der ‘MainPageView’ wird immer der Inhalt der aktuellen Seite angezeigt. Zum Beispiel für einen eingeloggten User wird hier zuerst die Deckauflistung angezeigt.

Architektur

Die Architektur der Anwendung wird im Wesentlichen durch das verwendete Web-Framework Angular vorgegeben. In diesem Abschnitt werden die für das Verständnis unserer Anwendung relevanten Angular-Konzepte erläutert und deren Anwendung beschrieben.

Grundlegende Funktionsweise von Angular

Eine Angular-Anwendung besteht aus einer Komponentenstruktur und einer klaren Trennung von Logik und Markup. Angular verwendet TypeScript-Komponenten, um die Anwendungslogik zu steuern und HTML für die Ausgestaltung der grafischen Benutzeroberfläche. Jede Angular-Komponente verfügt über eine TypeScript-Datei, die die Logik enthält, sowie eine HTML-Datei, die das Markup definiert.

Angular-Komponenten sind wiederverwendbare Bausteine, die in einer Hierarchie angeordnet sind. Jede Angular-Anwendung hat in der Regel eine Wurzelkomponente, die die gesamte Anwendung repräsentiert. Die Wurzelkomponente wird oft als “AppComponent” bezeichnet. Zusätzlich zu den vordefinierten Komponenten können Entwickler eigene Komponenten erstellen, um Code zu organisieren und wiederzuverwenden. Weitere Komponenten werden an die Wurzelkomponente angehängt und können auch per TypeScript nach bestimmten Auslösern aufgerufen werden.

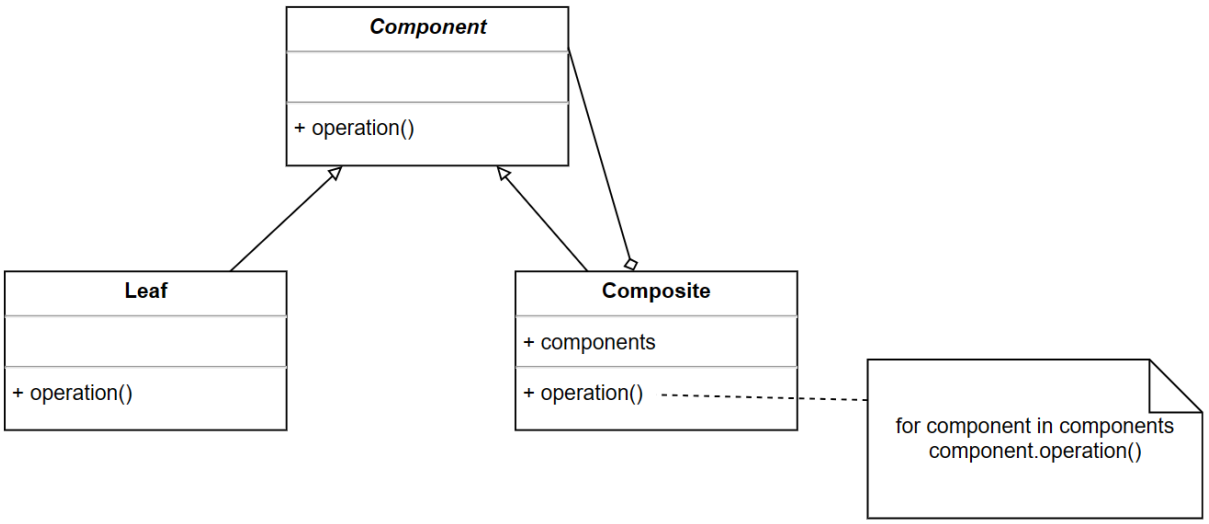

Abbildung 7: Technische Komponenten-Hierarchie (Auszug)

Angular verwendet Services, um Daten zwischen Komponenten auszutauschen und die Anwendungslogik zu kapseln. Diese Services sind TypeScript-Klassen, die in der Angular-Anwendung registriert werden und Methoden sowie Daten bereitstellen, die von verschiedenen Komponenten genutzt werden können.

Ein zentrales Konzept von Angular ist außerdem die Datenanbindung. Sie ermöglicht die automatische Synchronisierung von Daten zwischen Komponenten und der Benutzeroberfläche. Wenn sich Daten in einer Komponente ändern, wird die Ansicht automatisch aktualisiert.

Ereignisse in Angular werden ausgelöst, wenn Nutzer mit der Anwendung interagieren, beispielsweise durch das Klicken auf Schaltflächen oder das Absenden von Formularen. Angular bietet Mechanismen zur Reaktion auf diese Ereignisse, wie die Verwendung von Event Emitters oder das Reagieren auf Ereignisse in Komponenten.

Angular eignet sich besonders gut für die Entwicklung von Single Page Applications (SPAs), bei denen die gesamte Anwendung in einem einzigen HTML-Dokument geladen wird und das Navigieren zwischen verschiedenen Ansichten ohne vollständiges Neuladen der Seite erfolgt.

Angular bietet eine moderne und leistungsfähige Architektur für die Entwicklung von Web-Anwendungen. Mit klaren Strukturen und einer klaren Trennung von Anwendungslogik und Benutzeroberfläche unterstützt es die Entwicklung von robusten und skalierbaren Anwendungen.

Projektstruktur

Die Ordnerstruktur richtet sich nach der in Angular angegebenen Ordnerstruktur. Die Angular CLI generiert hierzu alles in einem Ordnerpfad namens '/src/app'.
Hierbei werden Services in den entsprechenden Serviceordner ausgelagert, um eine gewisse Trennung zu ermöglichen.
Komponenten können entsprechend auch verschachtelt sein, wenn es eine Unterkomponente von einer Root-Komponente ist.
Jede Komponente besitzt 4 Dateien:

- Ein entsprechende CSS Styling-Datei
- Eine HTML-Datei
- Eine für Tests vorgesehene Typescript-Datei
- Eine Typescript-Datei welche zu Javascript konvertiert wird

Die grobe Ordnerstruktur wird ebenfalls von Angular vorgeben. Sie sieht wie im folgenden Beispiel aus:

```
(In /src/app/)
├──node_modules
├ ... <Liste von node_modules>
└──src
   ├──app
   │   ├──(Root Component)
   │   ├──login component
   │   ├──(... andere components)
   │   └──services
   │       ├──UserService
   │       ├──CardService
   │       └──DataService
   │       └──(... andere Services)
   │
   └──assets
```

Abbildung 8: Ordnerstruktur Frontend

Routing

Fast jede Seite hat ihre eigene Route, unter der sie erreichbar ist:

- Login = /login
- Register = /register
- List Card Decks = /
- View Single Deck = /deck/{deck-id}
- View Deck Learn History = /deck/{deck-id}/history
- Edit deck = /deck/{deck-id}/edit
- Learning = /learning/{deck-id}
- Peek Learning = /learning{deck-id}/peek
- View/Edit Profile = /profile

Navbar

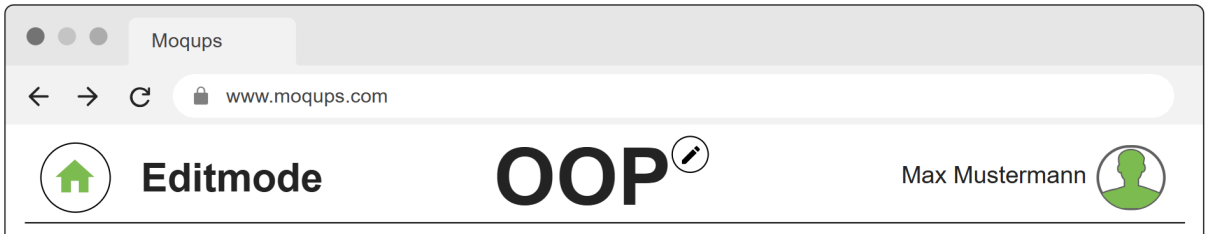

Abbildung 9: Navbar Beispiel

Die Navbar erscheint am oberen Ende jeder Seite, nachdem der User sich eingeloggt hat, die zum Navigieren zwischen den Ansichten/Komponenten benutzt wird.

Login-View

Die Login-Komponente ist eine wichtige Schnittstelle für Benutzer, um auf die Cards-Webanwendung zugreifen zu können. Hier werden die Benutzer authentifiziert und verschiedene Aktionen wie Registrierung, Passwort-Wiederherstellung und die Option “Remember me” ermöglicht.

1. Authentifizierungs-Service-Integration: Die Login-Komponente sendet bei Klick auf den Login-Button die Informationen an das Backend mit der Route ‘api/v1/login’. Falls die Anfrage nicht erfolgreich war, und das Backend den HTTP-Statuscode 401 (Unauthorized) zurückgegeben hat, wird ein entsprechender Toast mit “Die Anmeldedaten waren nicht korrekt, bitte versuche es erneut” angezeigt. Andernfalls wird der User zur Deckansicht-Komponente geroutet.
2. Passwortwiederherstellung: Beim Anfordern eines Password-Resets wird ein Modal geöffnet, worin der User seine E-Mail eingeben soll. Klickt er auf ‘Senden’ wird die POST-Route ‘api/v1/password/reset’ verwendet.
3. Durch das Setzen des Hakens auf “Remember me” wird auf dem Nutzerendgerät ein Cookie gespeichert. Dieses Cookie enthält die Anmeldeinformationen des Benutzers. Wenn der Benutzer die Website erneut aufruft, erkennt das System das Cookie und meldet den Benutzer automatisch an.

Register-View

Die RegisterView-Komponente ermöglicht dem Benutzer, sich einen Account auf der Datenbank zu erstellen. Hiermit kann sich der Benutzer in der LoginView-Komponente anmelden.

1. Registrierung: Mit dem Knopfdruck auf “Register” Button werden die Daten in den Feldern über die POST-Route unter *‘api/v1/register’* als .json übersendet. Falls die Route den HTTP-Statuscode 201 zurückgibt, wird man zur RegisterCompletedView-Komponente weitergeleitet.
2. ALT: Registrierung fehlgeschlagen: Bei einer fehlgeschlagenen Registrierung wird vom Backend ein HTTP-Statuscode empfangen, welcher nicht 201 ist. Dem User wird ein Modal angezeigt, worin steht, dass die Registrierung erfolglos war und er es erneut versuchen sollte.

RegisterCompleted-View

Die RegisterCompleted-Komponente gibt den Erfolg einer Registrierung wieder sowie auch, dass eine Verifikations-E-Mail gesendet wurde.

1. Registrierung erfolgreich: Wenn man auf den “Back to Login” drückt, wird man zurück zur Login-Komponente weitergeleitet.

Deck-View

Die DeckView ist die Hauptansicht, welche man angezeigt bekommt, sollte das Login auf der LoginView-Komponente erfolgreich gewesen sein. In dieser Komponente werden dem Benutzer alle Decks angezeigt und wie viele Karten in diesen verfügbar sind, die entsprechend integrierte Suchfunktion und die Sortierfunktion und Burger-Menü-Komponente stellen weiterführende Funktionen für diese Page bereit.

1. ngOnInit: Beim Laden der Komponente wird ein GET-Request an die Route ‘api/v1/decks’ gestellt, falls Decks existieren, werden dem User die entsprechenden Decks als Tabelle/Liste angezeigt.
2. Wenn anschließend auf einen Tabellen/Listeneintrag geklickt wird, wird er auf die DeckDetailView-Komponente weitergeleitet des jeweiligen Decks weitergeleitet
3. Funktionen des Burger-Menüs
    a. Create Deck: Wenn auf Create Deck geklickt wird, wird unten ein neues Element an die Liste/Tabelle gehängt. Hierbei muss der User einen Decknamen eingeben. Wenn er dies getan hat, wird ein POST-Request an ‘api/v1/decks’ gestellt.
    b. Import Deck: Hier öffnet sich der ‘File Select Context’ vom Browser. Hier kann nun eine ‘.zip’ Datei ausgewählt werden. Anschließend wird an das Backend eine POST-Route an ‘api/v1/decks/import’ gesendet. Falls ein HTTP-Statuscode != created kommt, wird als Toast eine entsprechende Fehlermeldung geworfen.
    c. DeleteDeck: Bei Klick auf den DeleteDeck-Buttons wird die DELETE-Route ‘api/v1/decks/{id}’ aufgerufen.

DeckDetail-View

Die DeckDetailView-Komponente ist die Hauptansicht eines Decks, hierbei sieht der Nutzer die Karten, die noch offen zu lernen sind, die Decksize, das entsprechende Donut-Diagramm, dass über die ‘Doughnut chart component’[3] von Bootstrap realisiert wird und die folgenden Funktionalitäten:

1. Share-Button: Wenn auf den Share Button gedrückt wird, öffnet sich ein Modal bei dem der User die E-Mail des Empfängers eintragen muss. Bei anschließendem Klick auf ‘Send E-Mail’ wird die POST-Request an 'api/v1/decks/{deckid}/share' gestellt.
2. Export-Button: Bei Knopfdruck auf den Export Button fängt ein Download an, bei dem das gesamte Deck mit seinen Karten heruntergeladen und als .zip Datei abgespeichert wird. Die genauen Informationen über das Deck werden über die GET-Route ‘api/v1/decks/{deckId}/export’ geladen.
3. History-Button: Beim Betätigen des History-Button wird man an den HistoryView-Komponente weitergeleitet.
4. StartToLearn-Button & Peek-Button: Wenn auf den StartToLearn-Button gedrückt wird, wird man auf die SessionView-Komponente weitergeleitet
5. Edit-Button: Der Edit-Button wird beim Betätigen den Benutzer an die Edit-Deck-Komponente weiterleiten.

History-View

Die HistoryView-Komponente zeigt dem Benutzer seine bisher abgeschlossenen Sessions an. Sie lädt, wann die Session gestartet wurde, wie viele Karten in der Session abgeschlossen wurden und ob diese überhaupt abgeschlossen wurde oder nicht.

1. ngOnInit: Das Laden dieser Komponente holt sich über die GET-Route ‘api/v1/historys’ die Details aller Sessions, die der User in diesem Deck durchgeführt hat.
2. Detaillierte History Ansicht: Mit Klick auf eine der Historien wird ein Modal angezeigt, worin ein Donut-Diagramm abgebildet wird, das die Bewertungen der Karten visualisiert.. Aufgerufen werden diese Details über die GET-Route ‘api/v1/historys/{id}’
    a. Mit Klick auf den “Back To History” Button schließt sich das Modal wieder.

Search

Die Suchfunktion ermöglicht es dem Nutzer, nach spezifischen Kartendecks zu suchen, um den schnellen Zugriff auf das gewünschte Deck zu erhalten, indem sie Decknamen in das Suchfeld eingeben. Das Bootstrap Autocomplete-Feld zeigt Suchvorschläge basierend auf den eingegebenen Zeichen an, um die Benutzersuche zu optimieren.

Sort

Die Sort-Komponente ermöglicht es dem Nutzer, die Kartendecks absteigend oder aufsteigend alphabetisch, nach Anzahl der Karten und zuletzt gelernt zu sortieren. Dazu wird eine Liste, die alle Kartendecks enthält, abgerufen. Anschließend wird die Liste mithilfe einer Methode nach den gewünschten Kriterien sortiert und zurückgegeben.

[3] https://mdbootstrap.com/docs/standard/extended/doughnut-chart/

EditDeck-View

In der EditDeckView-Komponente wird die Frage angezeigt und eine entsprechende Liste/Tabelle, die die Fragen widerspiegelt. Bei Klick auf eine der Fragen wird der Nutzer auf die entsprechende CardView-Komponente umgeleitet.
Bei Klick auf den Mülleimer neben einem entsprechenden Eintrag wird die DELETE-Route ‘api/v1/decks/deckId}/cards/{cardsId}’ aufgerufen.
Zudem beinhaltet die EditDeckView-Komponente noch ein Burgermenü mit den folgenden Funktionalitäten:

1. AddCard: Die AddCard Funktionalität leitet einen Nutzer standardmäßig auf die CardView-Komponente umgeleitet.
2. DeleteDeck: Bei Klick auf den DeleteDeck-Buttons wird die DELETE-Route ‘api/v1/decks/{id}’ aufgerufen und das Deck gelöscht.

Bei Initialisierung der Komponente wird die GET-Route ‘api/v1/decks/{id}/cards’ aufgerufen.

CardConfig-View

Die CardConfig-View-Komponente ist die obere Hälfte des bei Erstellung einer Karte. In dieser wird dem Nutzer das Question Feld angezeigt, indem die Frage der Karte steht, sowie der Typ der Karte angezeigt. Der Benutzer kann zwischen der Basic- und Multiple Choice Karte entscheiden. Die Komponente baut sich aus einer weiteren Unterkomponente auf. Die Unterkomponente ist hierbei entweder die BasicCardView-Komponente oder die MultipleChoiceCardView-Komponente.

1. Erstellen einer Karte: Die Karte wird beim Erstellen ohne Inhalt geladen. D.h., dass dem Nutzer als Typ der Karte standardmäßig Basic und ein leeres Fragefeld angezeigt wird.
2. Editieren einer Karte: Über die GET-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ werden die Details der Karte geladen. Darunter auch die Details des Question-Feldes. Über die PUT-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ werden die Informationen der Karte entsprechend aktualisiert.

EditBasicCard-View

Die BasicCardView-Komponente ist eine generische CardView, die sowohl für das Editieren einer BasicCard als auch für das Erstellen einer BasicCard verwendet wird.

- Sie besteht lediglich aus einem Fragefeld und einem Bild Upload Button

Hierbei holt sich das Frontend über die GET-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ die entsprechenden Informationen (Sofern eine Karte editiert wird, falls sie neu erstellt wird, werden keine Informationen angefragt).

EditMultipleChoiceCard-View

Mit der MultipleChoiceView-Komponente erstellt oder editiert man eine Multiple Choice Karte. Sie besteht aus einem Answer-Felder. Einer Option ein Bild hochzuladen und die Möglichkeit, ob die Answer korrekt ist oder nicht. Zudem existiert eine kleine Mülltonne, die nur verfügbar ist, falls eine Multiple-Choice-Card mehr als 3 Einträge besitzt.
Hierbei holt sich das Frontend über die GET-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ die entsprechenden Informationen (Sofern eine Karte editiert wird, falls sie neu erstellt wird, werden keine Informationen gefetcht).

Session-View

Die SessionView-Komponente wird je nach Anwendungsfall mit einer Peek-Session Komponente oder Learn-Session Komponente erweitert. Innerhalb der Session-Komponente wird eine Basiccard-Komponente oder eine Multiple-Choice-Komponente dargestellt. Die Session-Komponente ist daher lediglich ein Wrapper.

PeekSession-View

Bei der PeekSession Komponente wird dem User pro Seite eine Karte des gesamten Decks vorgelegt. Hierbei werden alle Karten im Deck durchgegangen. Mit dem “Next” Button kommt man auf die nächste Karte.

1. Die geladene Karte wird als “gelesen” gespeichert. Die Socket-Route ‘topic/sessions/peek/next’ wird dafür genutzt.

LearnSession-View

Die LearnSession Komponente gibt dem User eine n-Anzahl an Karten wieder, abhängig davon, wie viele Karten der User in einer Session im Profil-View ausgewählt hat. Hierbei hat der User 6 Buttons, womit er die Schwierigkeit der Karte auswählt.

1. Schwierigkeit auswählen: Die ausgewählte Schwierigkeit wird mit der Socket Route ‘topic/sessions/learn/next’ gespeichert.

BasicCard-View

Die BasicCard Komponente ist Teil der LearnSession & PeekSession Komponente. Sie wird für das Visualisieren einer Basic-Card verwendet. Dem User wird die Karte mit der Frage angezeigt. Ein Klick auf die Karte dreht die Karte um und zeigt damit die Lösung.

1. Laden der Karte: Die Karteninformationen werden über die GET-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ gefetcht und dargestellt.

MultipleChoiceCard-View

Die MultipleChoiceCard Komponente ist ein Teil der LearnSession & PeekSession. Sie wird in einer Session geladen. Dem User wird die Karte mit der Frage und den Kreuzfeldern, welche betätigt werden können, angezeigt. Ein Klick auf die Karte dreht die Karte um und zeigt damit die Lösung, welche Antworten falsch und welche richtig waren..

1. Laden der Karte: Die Karteninformationen werden über die GET-Route ‘api/v1/decks/{deckId}/cards/{cardsId}’ geholt und dargestellt.

Profile-View

Die Profilansicht ermöglicht es dem User, seine Daten wie E-Mail-Adresse, Passwort und Username zu überarbeiten.

1. Anzeigen von Benutzerdaten: Beim Laden der Profilansicht ruft die Komponente aktuelle Benutzerdaten vom Backend mit der Route ‘api/v1/account’ ab. Die Daten umfassen E-Mail-Adresse, Benutzernamen und möglicherweise erzielte Achievements (lediglich die IDs). Die Achievements werden anschließend über eine Achievement-Komponente geladen.
2. Aktualisierung von Benutzerdaten: Benutzer können ihre E-Mail-Adresse, ihren Benutzernamen und ihr Passwort in der Profilansicht aktualisieren. Bei jeder Änderung werden die aktualisierten Daten an den User Service mit der PUT-Route ‘api/v1/account’ gesendet. Erfolgreiche Aktualisierungen werden mit dem HTTP-Statuscode 200 (OK) quittiert.

SingleAchievement-View

Die Achievements veranschaulichen, wie viele Decks und Karten erstellt wurden, wie oft sich der Benutzer angemeldet hat und wie viele Lern-Sessions der Nutzer umgesetzt hat uvm. Um das Icon eines Achievements zu laden, wird ein GET-Request an die Route ‘api/v1/achievement/{id}/image’ beim Erstellen der Komponente gestellt. Falls der User dann über das Icon hovern sollte, wird ein GET-Request an ‘api/v1/achievement/{id}/data’ gestellt. Hierbei werden dann Informationen, wie achievedAt, Beschreibung und Achievementnamen geladen und entsprechend angezeigt.

Guidelines

Wann ist welcher HTTP-Statuscode zu verwenden?

- 200 (OK): Falls eine Anfrage akzeptiert und verarbeitet worden ist + response
- 201(Created): Neue Daten auf einem Service kreieren.
- 202 (Accepted)
- 204 (No Content): Anfrage akzeptiert, verarbeitet, aber kein Response
- 301 (Moved Permanently)
- 307 (Moved Temporarily)
- 400 (Bad Request): Daten im falschen Format/malformed oder erfüllt Standards nicht
- 401 (Unauthorized): Anfrage ohne Authentifizierung
- 404 (Not Found): Anfrage von Daten, die nicht existieren
- 409 (Conflict): Informationen existiert bereits
- 500 (Internal Server Error): Service nicht erreichbar oder anderweitige Probleme

General Guidelines

Git

Kommentarsprache & Commit-Sprache & Issue-Sprache => Englisch

Issues

- Beschreibende Titel verwenden
- Imperativsätze, falls möglich. Bsp. “Revise the main figure” anstatt “main figure”
- Ziele des Issues klar definieren!
- Falls ein Issue mit anderen in Korrelation steht, write: “Follow-up to #ISSUE_ID”
- Falls ein Issue abgeschlossen ist, einen Abschlusskommentar schreiben und es schließen.
    - Pull Request mit Peer-Review initialisieren, damit der Issue-Branch in den Master gemerged wird.
- Falls ein Issue aus mehreren Teilen besteht, verwende eine ToDo-Liste, um die Aufgabe entsprechend runterzubrechen

Issue-Template

Goal of this issue
definiere das klare Ziel

Resources
Angeben, welche Ressourcen verwendet werden(z. B. wo relevanter Code, wissenschaftliche Arbeiten usw. zu finden sind)

Commits

- Committe nur abgeschlossene Aufgaben
    - Falls dies nicht nötig ist, commite ein WIP: Aufgabe an der gearbeitet wurde
- Beschreibende Commit-Messages! use: “fix”, “refactor”, “add”, “remove”, “update”
- Nicht zu große Commits! Falls ein Revert stattfinden muss, geht vergleichsweise viel Progress verloren

Code

- Backend
    - Controller ist verantwortlich für das handeln der HTTP-Layer
    - Controller sollten keine Business-Logik ausführen!
    - Use REST-Conventions
    - Service: Sollte um Business-Logik gebaut werden.
    - Use Constructor injection!
    - Write meaningful Unit-Tests
    - An die Richtlinien von Java halten
    - Schreib nur Kommentare, wenn es auch sinnig ist!
    - Maximal 200 Zeichen pro Zeile
- Frontend
    - Follow Naming Conventions
    - Klare Ordnerstruktur, Siehe Softwareentwurf
    - One File per Object-Class
    - Use ESLint! If IDE doesn’t has it, ng add @angular-eslint/schematics
    - Reuse Components if possible
    - Use Interfaces
    - Vermeide Callback-Hölle
    - Vermeide type “any”
    - Use Lazy Loading!
    - Schreib nur Kommentare, wenn es auch sinnig ist!
    - Write meaningful Tests
    - An die Richtlinien von Typescript halten
    - Maximal 200 Zeichen pro Zeile

Halte dich an …
DRY (Don’t Repeat Yourself),
KISS (Keep it Stupid Simple), SOLID, YAGNI (You ArenÄt Gonna Need it), BDUF (Big Design Up Front), SOC (Separation of Concerns), …

CI/CD-Pipeline

Stages

Die CI/CD-Pipeline, also Continuous Integration/Continuous Deployment Pipeline, besteht aus mehreren Phasen.

- Build Stage
    - Die Build Stage, baut unser Frontend und unser Backend
- Test Stage
    - Die Test Stage, testet unsere Anwendung, bspw. durch Unit-Tests
- Sonarqube Stage
    - Die Sonarqube Stage testet unsere Anwendung anhand von statischer Codeanalyse
- PushToRegistry Stage
    - Die PushToRegistr Stage pusht unsere Docker-Container in die Container-Registry von GitLab. Dadurch ermöglichen wir ein anschließendes Deployment, z.B. zu einem Server oder lokal via Docker-compose-file.