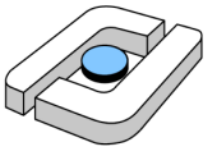


Hochschule Osnabrück

University of Applied Sciences

Fakultät

Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

Umsetzung der Webanwendung StudCar – studentische Mitfahrgelegenheiten inklusive einer REST-API mit Java, Jakarta EE und Quarkus

im Rahmen des Moduls
Software-Architektur – Konzepte und Anwendungen,
des Studiengangs Informatik-Medieninformatik

Autor 1:	Hendrik Purschke
Matr.-Nr.:	982448
E-Mail:	hendrik.purschke@hs-osnab-rueck.de
Autor 2:	Andreas Morasch
Matr.-Nr.:	978554
E-Mail:	andreas.morasch@hs-osnab-rueck.de
Themensteller:	Prof. Dr. Rainer Roosmann

Abgabedatum: 24.07.2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Source-Code Verzeichnis

Abkürzungsverzeichnis

IDE	Integrated Development Environment
REST	Representational State Transfer
CRUD	Create, Read, Update, Delete
DTO	Data Transfer Object
JSON-B	JavaScript Object Notation Binding
ORM	Object Relational Mapping
JDBC	Java Database Connectivity
UI	User Interface
HTML	Hypertext Markup Language
API	Application Programming Interface
SQL	Structured Query Language
ACID	Atomicity, Consistency, Isolation, Durability

1 Einleitung

Ein Thema welches sowohl viele Studierende als auch Lehrende an der Hochschule Osnabrück beschäftigt ist der immer weniger werdende Anteil an Studierenden, die an Präsenzveranstaltungen teilnehmen. Bei vielen Studierenden ist der Grund, wieso diese regelmäßig nicht bei Vorlesungen auftauchen, der weite Anreiseweg. Diejenigen die motiviert genug sind dennoch die Reise anzutreten, werden nicht selten aufgrund von Streiks, Ausfällen oder einfach nur der Unpünktlichkeit vieler öffentlicher Verkehrsmittel ausgebremst.

Passiert dieses Szenario regelmäßig, dann nimmt die Motivation an Präsenzveranstaltungen teilzunehmen noch weiter ab. In dieser Projektarbeit soll eine Lösung dafür entwickelt werden, wie Studierende private Mitnahmegelegenheiten organisieren können, um pünktlich an der Hochschule einzutreffen. Die Lösung dafür ist eine Webanwendung und nennt sich StudCar.

1.1 Vorstellung des Themas

StudCar – studentische Mitfahrgelegenheiten ist eine Webanwendung, die es Studierenden ermöglicht andere Studierende in ihren privaten Fahrzeugen zur Hochschule mitzunehmen. Die Idee ist dabei ähnlich der von BlaBlaCar. Der signifikante Unterschied besteht darin, dass StudCar zum einen vollständig kostenlos ist und sich zum anderen ausschließlich nur für Reisen zur Hochschule Osnabrück eignet.

StudCar unterscheidet zwischen Fahrern und Mitfahrern. Dabei bietet die Anwendung insgesamt vier Möglichkeiten Fahrten zu organisieren.

Fahrer können öffentlich Fahrangebote anbieten und auf private eintreffende Mitnahmeanfragen reagieren oder sie gehen proaktiv auf ein öffentliches Mitnahmegesuch ein.

Mitfahrer haben die Möglichkeit eine öffentliche Mitnahmeanfrage zu platzieren worauf sich Fahrer privat melden können oder sie hinterlegen direkt in einem öffentlichen Fahrangebot eines Fahrers den Wunsch mitgenommen zu werden.

Durch die Abdeckung aller vier Möglichkeiten eine Fahrt zu organisieren, werden alle Interessen sowohl der Fahrer als auch der Mitfahrer berücksichtigt.

1.2 Ziel der Ausarbeitung

Das Ziel der Anwendung besteht darin die rückläufige Anzahl der Studierenden bei Präsenzveranstaltungen wieder steigen zu lassen. Dabei wird im Folgenden die zentrale Frage im Mittelpunkt stehen, wie es StudCar erreichen kann, dass Studierende diesen Service in Anspruch nehmen.

Die schriftliche Ausarbeitung soll außerdem dazu beitragen, dass der Aufbau und die Benutzung der Anwendung verstanden werden. Es wird dabei ein technischer Einblick in die Software gegeben, um nachvollziehen zu können, wieso bestimmte Ansätze gewählt wurden.

Zum Schluss sollte die Anwendung bedient werden können und alle Möglichkeiten eine Fahrt zu organisieren bekannt sein.

1.3 Aufbau der Hausarbeit

Zunächst wird auf die Grundlagen der Anwendung eingegangen. Es wird darüber aufgeklärt mit welcher Programmiersprache, welchen Frameworks und Bibliotheken gearbeitet wurde. Hierunter fällt auch die IDE mit der die Anwendung entwickelt wurde sowie das Versionsverwaltungstool. Weiterhin werden dort auch Rahmenanforderungen und Produktfunktionen detaillierter erläutert.

Im Anwendungskapitel wird dann auf die Entwicklung und Implementation diverser wichtiger Funktionen sowie der Anwendung als Ganzes eingegangen. Wichtige Bereiche der Anwendung sind unter anderem Security, Event, Quarkus Qute als Frontend Templating Engine und die drei verschiedenen Module der Anwendung.

Dafür werden neben Codeausschnitten auch die in der Anwendung eingebaute Prinzipien, welche eine gute Softwarearchitektur ausmachen vorgestellt und erläutert.

Zum Schluss gibt die Zusammenfassung wesentliche Erkenntnisse der Ausarbeitung nochmals wieder und das Fazit evaluiert diese. Es wird dort auch ein Ausblick darüber gegeben, was der Anwendung fehlt und was noch in Zukunft implementiert werden könnte.

2 Darstellung der technischen Grundlagen

In diesem Abschnitt werden die für die Anwendung zu Grunde liegenden Technologien, Frameworks und Bibliotheken genauer erläutert. Zudem wird darauf eingegangen welche Frameworks und Bibliotheken im Allgemeinen für welchen Bereich der Anwendung verwendet wurden.

2.1 IDE, Programmiersprache und Versionsverwaltung

Als IDE wird auf Visual Studio Code zurückgegriffen. Es handelt sich dabei um einen kostenlosen Quelltexteditor von Microsoft. Als Programmiersprache wird Java in der Version 20 verwendet.

Um Java in Visual Studio Code nutzen zu können wird das Erweiterungspaket für Java aus der integrierten Erweiterungsbibliothek heruntergeladen. Dieses umfasst unter anderem Codevervollständigung, Debugmöglichkeiten, Refactoring und Codenavigation [@MS1].

Die Versionsverwaltung erfolgt mithilfe von GitLab auf Basis des Versionsverwaltungstools Git.

2.2 Externe Abhängigkeiten (pom.xml)

StudCar verwendet diverse Frameworks deren Abhängigkeiten in der pom.xml Datei des Projektes definiert werden müssen. Im Folgenden wird über die wichtigsten Abhängigkeiten informiert und verdeutlicht wieso diese relevant für die Anwendung sind.

2.2.1 REST-Schnittstellen mit RESTEasy Classic & Classic JSON-B

Bei RESTEasy Classic handelt es sich um ein Framework, welches diverse Schnittstellen zur Entwicklung von RESTful Webservices bereitstellt. Unter anderem sorgt es dafür, dass serverseitige http-Methoden von außen aufgerufen werden können [@QKS1]. Diese sind insbesondere GET, POST, PUT und DELETE.

Der Datentransfer erfolgt dabei immer im JSON-Format. Um ein JSON-String in eine Java Klasse serialisieren zu können, wurde der Standard JSON-B verwendet. JSON-B erkennt dabei automatisch die Datentypen des JSON-Strings und mappt diese auf das jeweilige im Methodenkopf vorgegebene Data Transfer Object (DTO).

2.2.2 Persistenz mit Hibernate ORM und JDBC PostgreSQL

Die Persistenz erfolgt vollständig über Hibernate. Hibernate ist dabei ein Persistenz Framework für Java für objektrelationale Abbildungen (ORM). Es wurde in diesem Fall in Kombination mit dem JDBC-Driver des relationalen Datenbankmanagementsystems PostgreSQL verwendet. [@QKS2]

2.2.3 UI mit RESTEasy Qute

Qute ist eine Templating Engine, speziell entworfen für die Benutzung im Quarkus Entwicklungsumfeld zur Darstellung von dynamischen Inhalten auf einer Weboberfläche.

Es bietet dabei die Möglichkeit aus einer HTML-Vorlage auf Inhalte zuzugreifen, die bei der Generierung der jeweiligen Seite vom Server in einer TemplateInstance zur Verfügung gestellt werden. Dieser Prozess sorgt dafür, dass die Webanwendung mit dynamischen Inhalten versorgt wird und sich bei einer Änderung der bereitgestellten Inhalte selbständig ändert. [@QKS3]

2.2.4 Security mit OpenID Connect und Keycloak

Für die Absicherung der Schnittstellen wird das OpenID Connect Protokoll verwendet und dafür die von Quarkus bereitgestellte Implementierung. OpenID Connect baut dabei auf dem OAuth 2.0 Framework auf. [@QKS4] Dies hat den Vorteil, dass auch bestehende Nutzerverwaltungen, die den genannten Standard erfüllen verwendet werden können und der Austausch mit wenig Aufwand verbunden ist.

OpenID Connect ermöglicht eine Authentifizierung und Autorisierung mittels eines Authentifizierungsservers. Dadurch wird die Verwaltung der Nutzer inklusive der Passwörter außerhalb der Anwendung durchgeführt und bietet so mehr Sicherheit beim Anmeldeprozess. In diesem Fall wird ein Keycloak Server verwendet, der Open Source ist und von Quarkus unterstützt wird.

Der Keycloak ist neben seiner Funktion eines Authentifizierungsservers auch ein Identity Provider. Es ist also möglich Informationen von Nutzern in diesem zu speichern und bei Bedarf abzurufen. So kann in der Anwendung komplett auf ein Modul für die Verwaltung von Nutzern verzichtet werden.

Keycloak bietet die Möglichkeit der Multitenancy, also der Mehrmandantenfähigkeit. Dazu werden in Keycloak sogenannte Realms eingerichtet, die komplett voneinander abgeschottet sind. Dadurch ist möglich auf einem Server für Unterschiedliche Organisationen Nutzerinformationen zu verwalten, ohne dass diese sich gegenseitig sehen. Im Folgenden wird nur ein Realm des Keycloaks betrachtet und die Möglichkeit anderer bestehender Realms wird außer Acht gelassen. Zusätzlich wird ein Realm in verschiedene Clients unterteilt, die einzeln konfiguriert werden können und Zugriff auf mehr oder weniger Informationen haben, bzw. besondere Rollen für die Nutzer ihres Realms konfigurieren können. Hier wird auch nur ein Client betrachtet, da die Applikation als solch ein Client bezeichnet werden kann und andere Applikationen und Clients keine Rolle spielen.

2.2.5 Schnittstellendefinition mit OpenAPI und SwaggerUI

Für die Auflistung und Beschreibung aller von außen zugänglichen HTTP-Schnittstellen wurde OpenAPI und SwaggerUI verwendet. Es handelt sich hierbei um Standards, die dem Nutzer die Möglichkeit bieten Einsicht auf die HTTP-Schnittstellen einer Anwendung zu erhalten. [@QKS5]

Beide Standards lassen sich ausschließlich bei laufender Anwendung öffnen. OpenAPI über localhost:8080/openapi/?format=json und SwaggerUI über localhost:8080/swaggerui.

2.2.6 Tests mit Rest Assured

Die einzelnen REST-Schnittstellen der StudCar Anwendung wurden mittels der Java Bibliothek Rest-Assured getestet. Dabei stellt Rest Assured Schnittstellen bereit die es ermöglichen auf die Schnittstellen der eignen REST API zuzugreifen und die Response, also die Antwort des Servers, auszuwerten. [@QKS6]

Die Art der Auswertung erfolgt dabei nach dem Prinzip, dass die eingeholte Ausgabe der erwarteten Ausgabe entspricht.

3 Aufbau

Es folgt ein detaillierter Einblick in die Planung und Entwicklung der StudCar Anwendung. Neben den Klassendiagrammen, welche die Struktur und Beziehung der einzelnen Module aufzeigen, wird darauf eingegangen, wie die einzelnen Module umgesetzt wurden.

Dafür werden in diesem Kapitel viele Codeausschnitte zur Darstellung einer beschriebenen Lösung verwendet.

3.1 Aufbau Offer- und Requestmanagement Module

In Anlage A und Anlage B sind jeweils die Klassendiagramme der Module Offermanagement und Requestmanagement zu finden. Da diese vom Aufbau und Kontext sehr ähnlich sind werden sie im Folgenden zusammen erläutert.

Beide Module sind nach dem Prinzip der layered architecture aufgebaut. Das bedeutet, dass es eine technische Unterteilung der Module in einzelne Schichten gibt. Diese Unterteilung in Schichten wird mithilfe von Java packages erreicht. Ziel dieser Architekturart besteht darin, das Design so kohärent zu gestalten, dass eine Schicht lediglich von einer darunterliegenden Schicht abhängig ist. Dabei ist die Kohäsion so zu gestalten, dass es eine starke Bindung innerhalb einer Schicht gibt. Zudem werden Interfaces benutzt, um eine möglichst lose Kopplung zwischen den darüber liegenden Schichten zu schaffen. [@RO1]

Innerhalb einer Schicht gibt es ebenfalls Unterteilungen. Diese sind aber fachlich und nicht technisch. So unterscheidet das Offermanagement in der Domänenschicht zwischen einem Offer dt. Angebot und einer RideRequest dt. Mitnahmeanfrage, da es sich um zwei verschiedene Entitäten handelt, die jedoch beide fachlich zu einem Modul gehören. Die folgende Abbildung verdeutlicht das Prinzip des Unterteilungsansatzes.

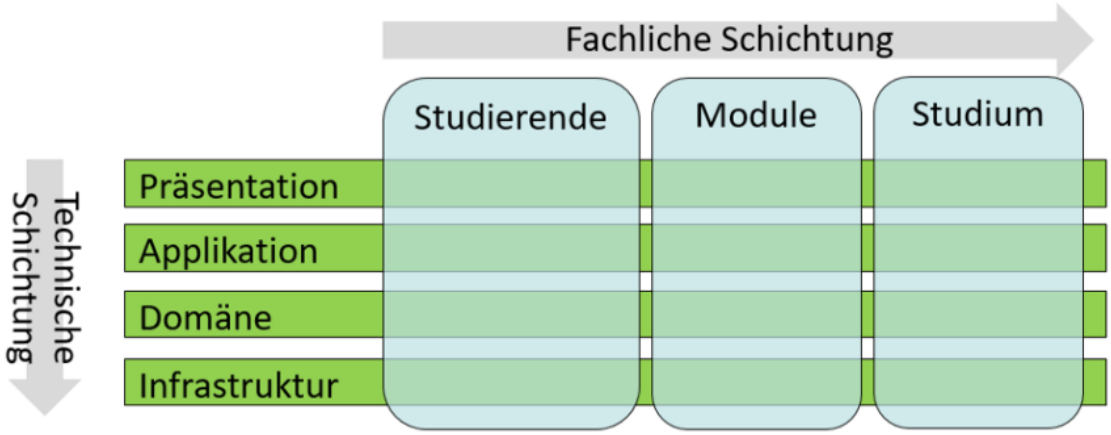


Abbildung 1: Visualisierung der Unterteilungsansätze [@RO1]

Sowohl das Offer- als auch das Requestmanagementmodul sind von oben nach unten in die technischen Schichten boundary, control, entity und gateway unterteilt.

Die boundary ist dabei die Schicht, die für die Kommunikation nach innen und nach außen zuständig ist. Jegliche initiale Kommunikation nach außen findet also in der boundary statt. Hier befinden sich neben Data Transfer Objects auch sämtliche REST-Schnittstellen, auf die ein Nutzer zugreifen kann. Diese REST-Schnittstellen werden in Ressourcen definiert und implementiert.

Die Ressourcen des Offermanagements bieten dabei CRUD-Schnittstellen in Form von HTTP-Methoden für Angebote und Mitnahmeanfragen. Ein Angebot hält dabei eine Liste von Mitnahmeanfragen, welche sich auf dieses Angebot beziehen.

Beim Requestmanagement ist es ähnlich, bloß mit dem Unterschied, dass hier die Mitnahmeanfrage jene Entität ist, welche eine Liste von Mitnahmeangeboten von Fahrern hält.

Die beiden Module unterscheiden sich also insofern, als dass sie hinsichtlich der stärkeren und schwächeren Entität genau gespiegelt sind. Beide Module weisen dabei die gleichen Schnittstellen für eine jeweils andere Entität auf. Wofür die einzelnen Schnittstellen zuständig sind, ist ebenfalls aus der Anlage A und B zu entnehmen.

Die boundary kommuniziert weiterhin lediglich mit der darunterliegenden control. Diese ist primär dafür zuständig die angefragten Informationen über den Katalog der Entität einzuholen oder über eingetroffene Änderungen zu informieren. Diese Aufgabe erledigen Services in der control. In den Services erfolgt ebenfalls das Versenden von Events an andere Module. Aufgrund dessen, erfüllt die control mehr Aufgaben als nur die reine Delegation an andere Schichten und ist somit unverzichtbar. Zusätzliche Operationen, wie Filtern und Sortieren bestimmter Anfragen werden kontextspezifisch auch in den Services umgesetzt.

Die control kommuniziert nach innen ausschließlich mit der entity. Die entity stellt dabei Interfaces, die in den Klassendiagrammen auch Kataloge genannt werden, für das gateway bereit. Die control kann somit indirekt über die Interfaces der entity mit der Datenbank kommunizieren, um Informationen einzuholen oder diese über Änderungen zu informieren. Die entity ist also immer die Schicht, die im Fokus steht und zu der alle anderen Schichten technisch hinzeigen. Hier werden zudem auch individuelle Exceptions definiert die von den Repositories eines gateway im Falle eines Fehlers geworfen und in der boundary gefangen werden. In der entity werden damit die grundsätzlichen Elemente der Anwendung beschrieben, die sich aus der Domäne ergeben. Damit ist diese Schicht langfristig am stabilsten und muss bei äußeren Änderungen am Userinterface oder der Datenbank nicht verändert werden. Deshalb ist es tragbar, dass alle anderen Schichten von dieser abhängen.

Das gateway hat dieselbe Anzahl an Repositories dt. Aufbewahrungsorte wie die entity Entitäten hat. Diese Erkenntnis folgt aus der Tatsache, dass jede Entität im Normalfall persistiert wird. Repositories kümmern sich also um die Persistenz und Verwaltung der einzelnen Entitäten. Somit ist das die Stelle, an der Informationen über Angebote und Mitnahmeanfragen eingeholt oder Änderungen vorgenommen werden.

3.2 Aufbau Events- und Notificationmanagement Module

Wie bereits in Abschnitt 3.1 erwähnt wurde, findet die Kommunikation zwischen voneinander unabhängigen Modulen über Events statt. Dabei werden Events in den Services der control in den einzelnen Modulen erstellt und versendet. Welcher Service auf welches Event zugreift und sendet, regelt die vorher durchdachte fachliche Unterteilung des Moduls. Da es sowohl im Offer- als auch Requestmanagement Modul jeweils zwei Services gibt, kann hier eine fachliche Trennung vorgenommen werden. So werden alle Events, die etwas mit einer Änderung hinsichtlich eines Angebotes im Offermanagement zu tun haben, auch nur in dem OfferService des Offermanagements gesendet.

Jedes Event findet sich dabei in dem Modul Shared wieder, da dort sämtliche Klassen liegen, auf die mehrere Module zugreifen müssen. Dieses Shared Modul ist unter Anlage 0 zu finden.

Während sowohl das Offer- als auch das Requestmanagement jeweils Events verschicken, sobald relevante Änderungen aufgetreten sind, kümmert sich das Notificationmanagement um das Fangen und Verarbeiten dieser Events. Dabei entspricht das Modul Notificationmanagement vom architektonischen Aufbau dem des Offer- und Requestmanagements. Zusätzlich wurde bei diesem Modul bewusst auf die control Schicht verzichtet, da in dem Fall die Aufgabe der control nicht über einfache Delegationsarbeit hinausreicht.

Kernelement des Notificationmanagement Moduls ist der NotificationService in dem von anderen Modulen gefeuerte Events gefangen und verarbeitet werden. Verarbeitung bedeutet in diesem Fall die Konstruktion einer Notification Entität anhand empfangener Daten aus dem Event. Die Notification Entität besteht aus der ID des Empfängers, einem Betreff, einer inhaltlichen Textnachricht, einem Datum sowie einem boolean Wert, der aussagt, ob die Nachricht gelesen wurde oder nicht.

Diese Entität wird dann an über die entity Schicht an das Repository gegeben, um es zu persistieren. Außerdem wird im NotificationService für jedes empfangene Event eine E-Mail an die passende Mailadresse der zu benachrichtigen Person gesendet. Das Notificationmanagement stellt wie alle anderen Module auch eine REST-Schnittstelle für den Zugriff von außen bereit. So lassen sich unter anderem dynamisch zu einer bestimmten Person dessen Nachrichten ausgeben, sodass ein Postfach in der Weboberfläche realisiert werden kann. Die weiteren REST-Schnittstellen sind hierbei das GET auf /notifications/{notificationId} zur

Detailansicht einer Notification sowie das PUT auf denselben Pfad um die Notification auf „gelesen“ zu setzen.

4 Implementation

4.1 Authentifikation mit Keycloak

Damit keine Unbefugten auf die Schnittstellen zugreifen, müssen diese abgesichert werden. Dazu werden Anfragen gegen die Schnittstelle nur mit einem gültigen Access-Token angenommen, welches immer gegenüber dem gegebenen Authentifizierungsserver abgeglichen wird. Bei einer Anfrage ohne Access-Token wird der Nutzer zum Authentifizierungsserver weitergeleitet, wo sich dieser mit seinem Benutzernamen und Passwort anmelden kann und ein Access-Token erhält, mit dem er Zugriff auf die Schnittstelle bekommt.

Quarkus bietet für den Keycloak Server eine Bibliothek, mit welcher der Server in der Applikation einfach konfiguriert werden kann. [QKS4] Außerdem kann ein solcher Server durch die Dev-Services von Quarkus in einem Docker-Container bereitgestellt werden, wodurch die Entwicklung vereinfacht und das Ausführungsumfeld irrelevant wird. Damit der Keycloak Server im Entwicklungskontext bei jedem Start richtig konfiguriert ist wird eine Konfiguration im JSON-Format beim Start des Servers geladen. Darin sind sowohl Standard-Nutzer und ihre Informationen als auch ihre Rollen und Berechtigungen, sowie die verwendeten Token definiert.

Durch die Konfiguration der Anwendung ist es möglich die Überprüfung von Anfragen im Testkontext zu deaktivieren, was automatisierte Tests der Anwendung vereinfacht und die explizite Absicherung von Endpunkten mit Annotationen ersetzt.

4.2 Implementation des Offermanagement Moduls

In diesem Abschnitt wird auf die Implementation des Offermanagement Moduls eingegangen. Das Offermanagement Modul weist starke Ähnlichkeiten zum Requestmanagement Modul auf. Aus diesem Grund wird das Requestmanagement nicht weiter im Detail ausgeführt. Alle Implementationsprinzipien lassen sich auf die gleiche Weise auf das Requestmanagement übertragen. Es werden im Folgenden alle Schichten des Offermanagements analysiert und die wichtigsten Inhalte mit Codeausschnitten verdeutlicht.

4.2.1 Boundary

Die boundary des Offermanagements besteht zentral aus den beiden Ressourcen OfferResource und RideRequestResource. Die OfferResource stellt die Schnittstellen für die Verwaltung von Angeboten nach außen zur Verfügung. Während die RideRequestResource die Schnittstellen für die Verwaltung von Anfragen auf ein entsprechendes Angebot bereitstellt.

Beide Ressourcen haben die gleichen Klassenannotationen. Diese sind in Snippet 1 zu sehen. Dabei sorgt @RequestScoped dafür, dass für jede HTTP-Anfrage eine neue Instanz der annotierten Klasse erstellt wird und diese so lange existiert wie die Anfrage selbst. Die Annotation @Transactional(TxType.REQUIRES_NEW) stellt sicher, dass jede Methode in einer neuen Transaktion ausgeführt wird [JVX1]. @Path gibt an unter welchem Pfad im Browser die Ressource erreichbar ist [JVX2]. @Consumes und @Produces, beide versehen mit MediaType.APPLICATION_JSON, machen bekannt, dass die jeweilige Ressource JSON-Strings als Eingabedaten erwartet und auch JSON-Strings als Ausgabedaten liefert [JVX3].

In Snippet 1 ist zudem die GET-Schnittstelle zu erkennen, welche dem authentifizierten Nutzer eine Liste aller vorliegenden Angebote liefert. Dabei kann der Nutzer einige @QueryParams *dt.* Anfrageparameter mitgeben, die für die Filterung der ausgegebenen Liste verwendet werden.

Die Arbeitsweise der Methode besteht daraus, sich zuerst eine Sammlung aller Angebote über die control zu holen. Die Kommunikation mit der control findet dabei über die Implementation des Interfaces ManageOffers statt. Diese Implementation lässt sich über die Annotation @Inject finden und in die OfferResource injizieren.

Sobald die Sammlung der Offers vorliegt, wird diese in eine Liste von OfferDTOs gemappt. So wird sichergestellt, dass sich die Schnittstelle nach außen nicht ändert, nur weil sich in der Logik der Applikation etwas ändert und auch nur die gewünschten Informationen nach Außen gelangen. Dafür ist die statische Methode *mapOfferToDTO(Offer offer)* aus der Klasse OfferDTO zuständig. Im Anschluss wird die gemappte Liste als Response über *Response.ok().entity(dtos).build();* an den Anfragesteller als JSON-String zurückgeliefert.

```java
@RequestScoped
@Transactional(TxType.REQUIRES_NEW)
@Path("/offers")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class OfferResource {

    private final String NOTFOUNDMESSAGE = "There is no offer with this offerId.";
    private final String NOMORESEATSMESSAGE = "There are no more seats available.";

    @Inject
    ManageOffers manageOffers;

    @Inject
    JsonWebToken principal;

    @GET
    public Response getAllOffers(@QueryParam("sorting") String sorting,
            @QueryParam("driveDate") String driveDate, @QueryParam("destinationCampus") String destinationCampus,
            @QueryParam("seats") Integer seats) {

        Collection<Offer> offers = this.manageOffers.selectAllOffers(sorting, driveDate,
                destinationCampus, seats);
        Collection<OfferDTO> dtos = offers.stream().map(OfferDTO::mapOfferToDTO).toList();
        return Response.ok()
                .entity(dtos)
                .build();
    }
```

Snippet 1: Klassenannotationen und @GET Schnittstelle der OfferResource

Für die Anlage eines neuen Fahrangebotes wird die http-Methode POST *createNewOffer()* verwendet. Diese ist in **Fehler! Verweisquelle konnte nicht gefunden werden.** zu erkennen.

```
@POST
public Response createNewOffer(@Valid NewOfferDTO dto) {

    String driverId = (String) principal.getClaim(claimName:"id");

    Long offerId = this.manageOffers.createOffer(driverId, NewOfferDTO.mapDTOToOffer(dto));
    if (offerId != null) {
        return Response.ok(offerId).build();
    }

    return Response.status(Status.INTERNAL_SERVER_ERROR).build();
}
```

Snippet 2: @POST Schnittstelle für die Erstellung eines neuen Angebotes

Für das Anlageszenario wird von der Methode ein DTO erwartet welches den Ansprüchen eines neuen Angebotes genügt. Darunter fällt das Weglassen von einer ID, da diese später vom EntityManager automatisch vergeben wird. Außerdem wird keine boolean Variable active, die aussagt, ob ein Angebot noch aktiv oder bereits inaktiv ist, erwartet. Der Nutzer muss ebenso keine Liste an Mitfahrern übersenden, da diese erst vorliegt sobald sich Mitfahrer auf das Angebot melden.

Zudem wird eine Validierung der vom Nutzer übergebenen Daten durchgeführt. Dafür sorgt die Annotation @Valid. Diese prüft, ob die vom Nutzer mitgegebenen Daten zum einen vorhanden sind und zum anderen auf das DTO gemappt werden können [JVX4]. Welche Daten vorhanden sein müssen und welche nicht regeln die Annotationen des jeweiligen DTOs.

NewOfferDTO verwendet dabei an jeder Objektvariable außer *String description* die Annotationen @NotNull und oder @NotBlank. Ersteres stellt sicher, dass eine Variable auf jeden Fall vorhanden sein muss. Das dazugehörige Objekt darf also nicht null sein. Zweiteres sorgt dafür, dass eine Variable, die existiert, nicht leer sein darf. Darunter fällt beispielsweise ein String, der nicht aus Leerzeichen bestehen darf. [JVX5]

Nach einer erfolgreichen Prüfung wird das neue Angebot von einem DTO in die passende Entität gemappt. Im Anschluss wird die darunterliegende Schicht über das jeweilige Interface über die ausstehende Änderung informiert. Sobald das Angebot angelegt wurde, steht in der offerId die ID des neu angelegten Angebotes. Diese ID ist dann auch der Inhalt der an den Nutzer als Response zurückgegeben wird.

An dieser Stelle wird auch das JsonWebToken ausgewertet und auf die ID des authentifizierten Nutzers zugegriffen, um das neue Angebot direkt mit einem bestimmten Nutzer zu verknüpfen. Das JsonWebToken ist hier das Access-Token, welches bei jeder Anfrage auf einen

Endpunkt vom Nutzer mitgeschickt und beim Keycloak überprüft wird. Das Token enthält bestimmte Nutzerinformationen, die im Kontext der Anfrage dann vorliegen und ausgewertet werden können. Hier ist jedoch nur die ID des aktuellen Nutzers relevant.

Eine weitere wichtige HTTP-Schnittstelle der OfferResource ist das @PUT. Diese sorgt dafür, dass Änderungen an bestehenden Angeboten vorgenommen werden können. Ein Snippet der der PUT-Schnittstelle des Offermanagements ist in Snippet 3 zu finden. Dafür erwartet die Methode ein OfferDTO mit allen änderbaren Inhalten. Das bedeutet, dass der Nutzer bei einem Änderungswunsch immer alle Daten, inklusive jener die nicht geändert werden sollen, angeben muss. Davon ist die Sammlung der Anfragen auf dieses Angebot jedoch ausgeschlossen, da der Angebotssteller keine Berechtigung hat Anfragen auf sein Angebot zu ändern. Grund für die vollständige Angabe aller Daten ist primär die einfache Handhabung Änderungen zu verarbeiten. Das Repository wird nämlich alle änderbaren Daten auslesen und schlichtweg erneut in die jeweilige Entität schreiben.

Auch diese Methode arbeitet mit der @Valid Annotation, die auf die gleiche Weise funktioniert wie die des NewOfferDTO. Sobald die Validierung abgeschlossen wurde, wird das DTO zunächst in eine Entität gemappt. Diese Entität wird dann über die Schnittstelle der control an das jeweilige Repository gegeben, in der die permanenten Änderungen stattfinden.

Die Antwort des Repositories ist ein boolean Wert, der aussagt, ob eine Änderung durchgeführt wurde oder nicht. Erfolgreich bedeutet in diesem Kontext, dass die Entität mit dieser ID vom EntityManager gefunden wurde.

Bei einer erfolgreichen Änderung erhält der Nutzer nichts weiter als den Status Code 200, andernfalls bekommt dieser eine Nachricht in der darüber aufgeklärt wird, dass das Angebot mit dieser ID nicht gefunden wurde.

```
@PUT
@Path("/{offerId}")
public Response modifyOffer(@Valid OfferDTO dto) {

    String driverId = (String) principal.getClaim(claimName:"id");

    Offer offer = OfferDTO.mapDTOToOffer(dto);
    boolean modified = this.manageOffers.modifyOffer(driverId, offer);

    if (modified) {
        return Response.ok().build();
    }

    return Response.status(Status.NOT_FOUND).entity(NOTFOUNDMESSAGE).build();
}
```

Snippet 3: @PUT Schnittstelle für das Ändern eines Angebotes

Die letzte aus der boundary interessante HTTP-Schnittstelle ist das @DELETE. Wie der Name der Annotation bereits andeutet, sorgt diese Methode dafür, dass ein Angebot gelöscht wird. Dazu benötigt die Methode nichts weiter als die ID des Angebotes. Die Methode ist im Snippet 4 zu sehen.

Anhand der ID wird das Repository erneut über das Interface der control benachrichtigt. Dieses sucht die passende Entität raus und löscht diese. Bei einer erfolgreichen Löschung wird deleted auf true gesetzt, andernfalls auf false.

Wenn das Löschen erfolgreich war, erhält der Nutzer auch hier den Status Code 200, da dieser auch nicht mehr als das erwartet. Bei einer nicht erfolgreichen Löschung konnte die Entität anhand der ID nicht gefunden werden. In diesem Fall bekommt der Nutzer einen Hinweistext zurückgeliefert, indem das Problem beschrieben wird.

```
@DELETE
@Path("/{offerId}")
public Response revokeOffer(@PathParam("offerId") Long offerId) {

    String driverId = (String) principal.getClaim(claimName:"id");

    boolean deleted = this.manageOffers.revokeOffer(driverId, offerId);

    if (deleted) {
        return Response.ok().build();
    }

    return Response.status(Status.NOT_FOUND).entity(NOTFOUNDMESSAGE).build();
}
```

Snippet 4: @DELETE Schnittstelle für das Löschen eines Angebotes

4.2.2 Control

Die control ist zum einen die Schnittstelle für die Kommunikation zwischen boundary und entity. Zum anderen werden hier die Events hinsichtlich einer jeweiligen Entität gefeuert.

Wie in Snippet 5 zu erkennen ist, implementiert OfferService das Interface ManageOffers, welches im selben package liegt und die zu implementierenden Methoden vorgibt. Außerdem ist OfferService mit @ApplicationScoped annotiert und hat somit für die gesamte Laufzeit der Anwendung nur eine Instanz, auf welche zugegriffen werden kann. [JVX6] Da der Service mit keinem eigenen Zustand arbeitet wurde hierbei bewusst auf das @RequestScoped verzichtet.

Zusätzlich hält der OfferService in diesem Fall drei Events, welche in bestimmten Szenarien gefeuert werden. Das OfferChangedEvent wird dann gefeuert, wenn sich ein bestehendes Angebot verändert hat, hierbei werden alle beteiligten Mitfahrer über die Änderung informiert. Das NewRequestCreatedEvent wird gefeuert, wenn auf ein bestehendes Angebot eine neue Mitnahmeanfrage eingetroffen ist. In diesem Fall wird lediglich der Fahrer über dieses Event informiert. Als letztes informiert das OfferRevokedEvent bei einer Löschung des Angebotes alle bereits akzeptierten Mitfahrer über dieses Ereignis.

In der Methode *selectAllOffers* lässt sich erkennen, wie die Anwendung grundsätzlich mit eintreffenden Filterparametern aus der boundary umgeht. Zunächst werden alle Angebote aus der Datenbank geholt. Das passiert über das Interface OfferCatalog welches vom jeweiligen Repository der entsprechenden Entität implementiert wird. Liegen die Angebote vor, werden die Filter, die der Nutzer mitgegeben hat, angewandt. Darunter fällt zum einen der Sortierungstyp für das Datum der Fahrt. Standartmäßig wird die Liste immer nach den frühsten, also am nächsten anstehenden Fahrten sortiert. Der Nutzer kann außerdem sowohl nach einem genauen Datum suchen als auch den Zielcampus und die Anzahl freier Sitze auswählen.

Rückgabetyp der *selectAllOffers* Methode ist daraufhin eine auf die Filterparameter des Nutzers angepasste Collection von Angeboten.

```
@ApplicationScoped
public class OfferService implements ManageOffers {

    @Inject
    Event<OfferChangedEvent> offerChangedEvent;

    @Inject
    Event<NewRideRequestCreatedEvent> newRequestCreatedEvent;

    @Inject
    Event<OfferRevokedEvent> offerRevokedEvent;

    @Inject
    private OfferCatalog offerCatalog;

    @Override
    public Collection<Offer> selectAllOffers(String sorting, String driveDate, String destinationCampus,
            Integer freeSeats) {

        Collection<Offer> offers = this.offerCatalog.selectAllOffers();
        String date = LocalDate.now().format(DateTimeFormatter.ofPattern(pattern:"yyyy-MM-dd"));
        return offers.stream()
                .filter(o -> o.getDriveDate().compareTo(date) >= 0)
                .filter(o -> driveDate == null || driveDate.equals(o.getDriveDate()))
                .filter(o -> destinationCampus == null || destinationCampus.equals(o.getDestinationCampus()))
                .filter(o -> freeSeats == null || freeSeats <= o.getFreeSeats())
                .sorted((a, b) -> sortOffers(a, b, sorting))
                .collect(Collectors.toList());
    }
```

Snippet 5: Event Objekte und selectAllOffers Methode des OfferService

Ein Beispiel wie das Versenden eines Events funktioniert ist in Snippet 6 zu erkennen. Hierbei wird der Fall betrachtet in dem ein Angebot vom Fahrer gelöscht wird. Bei einer erfolgreichen

Löschung wird die Methode *fireOfferRevokedEvent* aufgerufen. Diese Methode konstruiert daraufhin ein neues OfferRevokedEvent und befüllt dieses mit allen zu diesem Event relevanten Daten. Diese sind Fahrer ID, Offer ID, Reiseroute, Datum der Fahrt sowie einer HashMap von Request IDs zu den jeweiligen Mitfahrer IDs. Diese Informationen sind für das Notificationmanagement welches das Event fängt relevant, um weitere Verarbeitungen durchzuführen.

Zum Schluss wird das Event über die Methode *fire* versandt. Hierbei ist wichtig zu verstehen, dass dem Service egal ist, ob das Event von irgendeiner Instanz gefangen wurde oder nicht. Der Service übernimmt hierbei lediglich die Informations- aber nicht die Überwachungsfunktion.

```
@Override
public boolean revokeOffer(String driverId, Long offerId) {
    Offer revokedOffer = this.offerCatalog.selectOffer(offerId);
    boolean revoked = this.offerCatalog.revokeOffer(driverId, offerId);

    if (revoked) {
        fireOfferRevokedEvent(revokedOffer);
    }

    return revoked;
}

private void fireOfferRevokedEvent(Offer offer) {
    OfferRevokedEvent offerRevoked = new OfferRevokedEvent();
    offerRevoked.driverId = offer.getDriverId();
    offerRevoked.offerId = offer.getId();
    offerRevoked.route = offer.getStartLocation() + " to " + offer.getDestinationCampus();
    offerRevoked.date = offer.getDriveDate();
    offerRevoked.requestIdsToPassengerIds = getRequestIdsAndPassengerIds(offer.getRideRequests());
    this.offerRevokedEvent.fire(offerRevoked);
}
```

Snippet 6: Versenden eines Events bei der Löschung eines Angebotes

4.2.3 Entity

Die entity definiert sämtliche Entitäten, die von den Repositories des gateway verwaltet werden sollen. Die entity ist dabei die Schicht eines Moduls zu der alle anderen Schichten kommunikationstechnisch hinzeigen. Änderungen in der entity wirken sich kritisch auf alle anderen Schichten aus. Das ist aber beabsichtigt, da sich so eine möglichst lose Kopplung innerhalb aller anderen Schichten erreichen lässt.

In der entity liegen auch die Interfaces OfferCatalog und RideRequestCatalog auf welche die control zugreift, um mit dem Repository zu kommunizieren. Folglich werden diese beiden Kataloge von den Repositories implementiert um die Verwaltung der Entitäten Offer und RideRequest zu ermöglichen. Welche Entität welche Klassenvariablen hält ist dabei aus dem Klassendiagramm des Anhangs B zu entnehmen.

Des Weiteren enthält die entity das package „exceptions“ in der sämtliche Ausnahmeklassen zu beiden Entitäten definiert sind. Eine Auflistung der eigenen Ausnahmeklassen samt Erklärungstext ist in der nachfolgenden Tabelle 1 zu erkennen.

AcceptInactiveRequestException	Tritt auf, sobald der Fahrer versucht, eine inaktive Anfrage anzunehmen.
AddRequestToInactiveOfferException	Tritt auf, sobald der Fahrer versucht, eine Anfrage anzunehmen, obwohl sein Angebot bereits inaktiv ist.
NoMoreSeatsAvailableException	Tritt auf, sobald der Fahrer versucht, eine anfrage anzunehmen, obwohl keine freien Plätze mehr verfügbar sind.
RejectInactiveRequestException	Tritt auf, sobald der Fahrer versucht, eine inaktive Anfrage abzulehnen.
RideRequestAlreadyInactiveException	Tritt auf, sobald der Fahrer versucht, eine bereits abgelehnte Anfrage abzulehnen.

Tabelle 1: Exceptions aus der entity des Offermanagements

Auslöser für diese Ausnahmen werden in den Repositories erkannt und die entsprechende Ausnahme geworfen. Dabei wird die geworfene Ausnahme bis zur boundary hochgereicht, welche die Ausnahme dann fängt, und verarbeitet, um dem Nutzer ein verständliches Feedback zu liefern. Da es sich bei den Ausnahmen nicht um so genannte Runtime-Exceptions handelt ist der Aufrufer der Methode gezwungen sich mit der Ausnahme auseinander zusetzen.

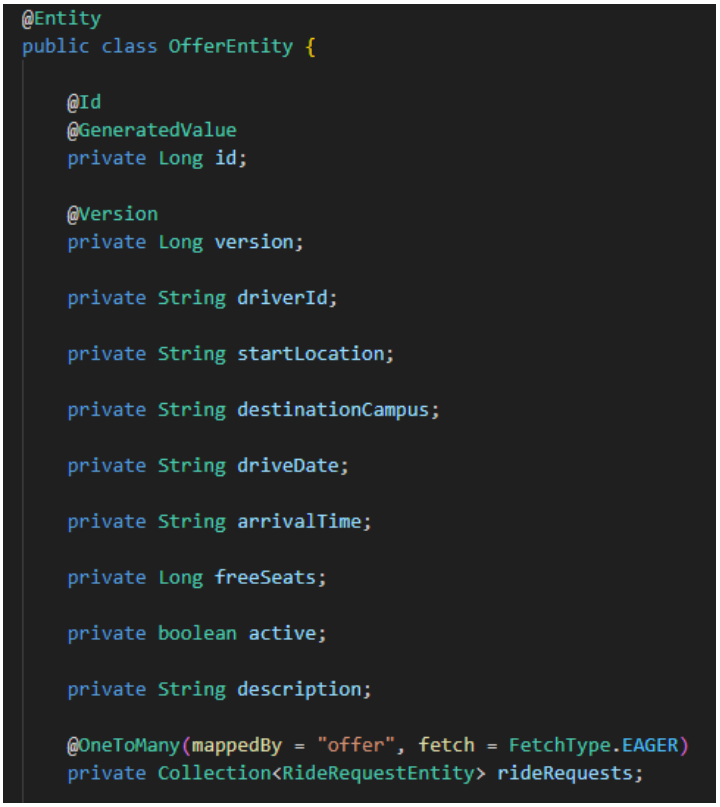
4.2.4 Gateway

Das gateway lässt sich auch als Datenbankendpunkt bezeichnen und kümmert sich um die Persistenz und Verwaltung der Entitäten. Es implementiert dabei die Katalog-Interfaces der jeweiligen Entität aus der entity. Es ist zudem mit @Transactional annotiert was dafür sorgt, dass sämtliche Transaktionen, die auf dieser einzigen Instanz ausgeführt werden, die ACID-Eigenschaften erfüllen und einhalten. Dabei ist die Transaktion jedoch notwendig und wird hier nicht neu erzeugt. Es muss also bereits eine aktive Transaktion existieren, damit eine Methode des Repositories ausgeführt werden kann. So wird das Transaktionsmanagement aus dem gateway herausgehalten und es werden gesamte Anfragen, die potentiell aus mehreren Datenbankzugriffen bestehen als eine zusammenhängende Transaktion betrachtet, die entweder ganz oder gar nicht ausgeführt wird.

Dabei verwalten Repositories nicht die Objekte aus der entity, sondern haben immer eigene Entitäten. Diese Entitäten unterscheiden sich von den Entitäten aus der entity über eine @Entity Annotation wie in Snippet 7 zu sehen ist. Diese macht für die Jakarta persistence Schnittstelle eine verwaltbare Entität kenntlich [JVX7].

Weiterhin werden in allen Entitäten des gateway die Annotationen @Id und @GeneratedValue verwendet. Ersteres definiert das Primärschlüsselobjekt und zweiteres sorgt dafür, dass dieser Primärschlüssel zur Laufzeit automatisch von der Datenbank vergeben wird [JVX8][JVX9].

Mithilfe der @OneToMany Annotation wird in diesem Fall der Datenbank die Beziehungsart zweier Entitäten zueinander mitgeteilt. Hierbei hält ein Angebot mehrere Mitnahmeanfragen. Hingegen ist jede Mitnahmeanfrage auf genau ein Angebot zurückzuführen. Somit ist im Gegenstück, das Angebot, zu welchem die Mitnahmeanfrage gehört, mit @ManyToOne annotiert.



```
@Entity
public class OfferEntity {

    @Id
    @GeneratedValue
    private Long id;

    @Version
    private Long version;

    private String driverId;

    private String startLocation;

    private String destinationCampus;

    private String driveDate;

    private String arrivalTime;

    private Long freeSeats;

    private boolean active;

    private String description;

    @OneToMany(mappedBy = "offer", fetch = FetchType.EAGER)
    private Collection<RideRequestEntity> rideRequests;
```

Snippet 7: Klassenkopf der OfferEntity aus dem gateway

Jedes Repository beinhaltet einen EntityManager über den die eigentliche Verwaltung der Entitäten erfolgt. In Snippet 8 ist dies zu erkennen. Der EntityManager stellt dafür Methoden zur Verfügung die sich aus dem Repository nutzen lassen. Es ist ebenfalls zu sehen, wie der Suchvorgang zur Ausgabe aller Angebote aussieht. Dabei muss dem EntityManager bei einer besonderen Suchen eine Query *dt*. Anfrage übergeben werden. In diesem Fall ist eine besondere Suche, die Suche nach allen vorhandenen Einträgen dieses Entitätstypen. Soll hingegen nach einem bestimmten Eintrag gesucht werden, so stellt der EntityManager die Methode *find(entityClass, primaryKey)* für genau dieses Anwendungsszenario zur Verfügung.

```java
@Transactional(TxType.MANDATORY)
@ApplicationScoped
public class OfferRepository implements OfferCatalog {

    @Inject
    private EntityManager em;

    @Override
    public Collection<Offer> selectAllOffers() {
        TypedQuery<OfferEntity> findAll = em.createQuery(qlString:"SELECT o FROM OfferEntity o",
                resultClass:OfferEntity.class);
        List<OfferEntity> entities = findAll.getResultList();
        return entities.stream()
                .map(OfferEntity::getOfferFromEntity)
                .toList();
    }
```

Snippet 8: Ausschnitt aus der OfferRepository samt selectAllOffers Methode

Falls Änderungen vorgenommen werden sollen, wird die Methode *merge* des EntityManagers verwendet. Diese findet die bestehende Entität nach dessen ID und liefert die Entität an den Aufrufer zurück. Daraufhin lassen sich an der zurückgegeben Entität Änderungen vornehmen. Im Anschluss sorgt der Aufruf von *merge* und die Übergabe der veränderten Entität an *merge* dafür, dass die Änderungen in der Datenbank übernommen werden. Dieser Vorgang ist in Snippet 9 zu finden.

```java
@Override
public boolean modifyOffer(String driverId, Offer offer) {
    OfferEntity entity = em.find(entityClass:OfferEntity.class, offer.getId());

    if (entity == null) {
        return false;
    }
    if (!entity.getDriverId().equals(driverId)) {
        throw new SecurityException(s:"You are not allowed to modify this offer!");
    }

    entity.modifyOfferEntity(offer);
    em.merge(entity);
    return true;
}
```

Snippet 9: Ändern einer OfferEntity in der OfferRepository mit *merge*

Neben *find* und *merge* ist die letzte in dem Repository wichtige Methode des EntityManagers das *remove*. Wie es der Name bereits vermuten lässt, handelt es sich hierbei um die Löschoperation des EntityManagers. Dabei funktioniert der Methodenaufruf ähnlich wie in Snippet 9, nur mit dem Unterschied, dass die Entität an die *remove* Methode übergeben und im Anschluss gelöscht wird.

4.3 Implementation der Events und des Notificationmanagements

Die in der Anwendung geworfenen Events dienen ausschließlich dem Informationszweck. Hat ein Fahrer beispielsweise sein Angebot geändert werden alle betroffenen Parteien außer der Person, welche die Änderung vorgenommen hat, benachrichtigt. Dafür werden, wie bereits in Abschnitt 4.2.2 angedeutet, Events gefeuert.

Diese gefeuerten Events werden nun im NotificationService des Notificationmanagement Moduls gefangen und verarbeitet. Der NotificationService liegt dabei in der entity des Notificationmanagement Moduls.

In Snippet 10 ist ein Ausschnitt des NotificationService zu sehen. Dieses ist mit @Transactional und @ApplicationScoped annotiert und injiziert das Interface NotificationCatalog welches die Methoden für das Repository definiert. Des Weiteren ist im Methodenkopf die Injektion eines Mailer Objektes aus dem io.quarkus.mailer Kontext zu finden. Es ermöglicht das Versenden von E-Mails. Das DateTimeFormatter Objekt wird im weiteren Verlauf dazu genutzt Zeitstempel für den Eintritt der Änderung zu formatieren.

In Snippet 10 ist auch zu erkennen, wie die Vorgehensweise für das Fangen eines Events aussieht. Jede Methode, die ein bestimmtes Event fangen soll, trägt die Annotation @Observes mit dem jeweiligen Event als eintreffenden Übergabeparameter. Diese Annotation sorgt dafür, dass die Methode genau den Eventtypen fängt welchen @Observes zuvor in den Übergabeparametern derselben Methode annotiert hat [JVX10]. Das eintreffende Event hält alle Daten, die für die Konstruktion der entsprechenden Benachrichtigung relevant sind.

Eine Benachrichtigung besteht dabei immer aus der ID der zu benachrichtigen Person, einem Betreff, einer Nachricht, einem Datum sowie einem boolean Wert, der aussagt, ob die Nachricht gelesen wurde oder nicht. Diese Informationen werden in der observierenden Methode extrahiert und an die *notificationBuilder* Methode gegeben, welche daraus ein Notification Objekt erstellt. Zum Schluss wird diese erstellte Notification über den Aufruf von *createNotification* persistiert und jede zu benachrichtigende Person per E-Mail informiert.

```java
@Transactional(TxType.MANDATORY)
@ApplicationScoped
public class NotificationService {

    DateTimeFormatter dtf = DateTimeFormatter.ofPattern(pattern:"yyyy/MM/dd HH:mm");

    @Inject
    NotificationCatalog notificationCatalog;

    @Inject
    Mailer mailer;

    public void onOfferChanged(@Observes OfferChangedEvent event) {
        String headline = "An offer has changed";
        String message;

        for (Map.Entry<Long, String> entry : event.requestIdsToPassengerIds.entrySet()) {
            message = "The offer with ID " + event.offerId
                    + " where you applied as a passenger has changed by the driver. Please " +
                    "check the changes and take action for request with ID "
                    + entry.getKey() + " if needed.";
            Notification notification = notificationBuilder(entry.getValue(), headline, message);
            this.notificationCatalog.createNotification(notification);
            sendMailToUser(notification);
        }
    }
```

Snippet 10: Ausschnitt aus dem NotificationService samt onOfferChanged Methode

Das Notificationmanagement Modul verfügt ebenfalls über eine boundary die den Zugriff von und nach außen steuert. Die boundary ist dabei recht kompakt und hält drei HTTP-Methoden, die im Folgenden kurz erklärt werden.

Das GET auf /notifications liefert eine Liste aller Benachrichtigungen zu diesem Nutzer wieder. Dieses GET wird verwendet, um einem Nutzer alle seine gelesenen und ungelesenen Benachrichtigungen anzuzeigen.

Weiterhin verfügt die boundary über ein zweites GET, nämlich auf /notifications/{notificationId}. Hier wird genau eine Benachrichtigung an den Nutzer zurückgeliefert, nämlich die mit der angegebenen notificationId. Das wird verwendet, um eine einzelne Nachricht anzuzeigen, sobald der Nutzer in dem User Interface draufklickt.

Zum Schluss sorgt das PUT auf /notifications/{notificationId} dafür, dass der „gelesen“ Status verändert wird. Klickt ein Nutzer in dessen Nachrichtenübersicht auf eine seiner Nachrichten, wird diese angezeigt und daraufhin als gelesen markiert.

4.4 Implementation des User Interface mit RESTEasy Qute

Das User Interface wurde mithilfe der Templating Engine RESTEasy Qute realisiert. Die Vorgehensweise dabei ist serverseitig Daten an das Frontend dynamisch bereitzustellen. Daraufhin lässt sich in der entsprechenden HTML-Datei auf diese Daten zugreifen und damit dynamisch die Webseite realisieren. [QKS3]

Um serverseitige Daten an das User Interface bereitzustellen, bedarf es, ähnlich wie bei einem REST-Ansatz auch einer Ressource. Jedoch liefert diese Ressource keinen JSON-String zurück, sondern ein sogenanntes TemplateInstance in dem die Daten für das User Interface stehen.

Diese Ressource liegt ebenfalls wie die anderen Ressourcen in der boundary, jedoch im separaten package *web*. Im weiteren Verlauf wird die OfferWebResource, für das Bereitstellen der Angebotsinformationen in der UI erläutert. Diese ist im Snippet 11 zu finden.

Es lässt sich erkennen, dass die WebResource aus dem gleichen Grund wie die anderen Ressourcen auch mit @RequestScoped und @Transactional sowie einem eindeutigen Pfad, der auf die Ressource verweist, annotiert ist. Der signifikante Unterschied besteht darin, dass diese Ressource als @Produces kein APPLICATION_JSON, sondern ein TEXT_HTML aufweist. Die ersten beiden @Inject Annotationen wurden in bereits in der boundary der OfferResource erläutert und befinden sich aus dem gleichen Grund auch in der WebResource.

Hinzu kommt zusätzlich das @Inject des UserService. Dieser ist vorrangig dazu da anhand von Fahrer IDs die jeweiligen Namen der Fahrer aus dem Keycloak Server zu suchen.

In dem Konstruktor der OfferWebResource wird das zu dieser Ressource dazugehörige Templlate gesetzt. Dieses lässt sich eindeutig über den Dokumentnamen der HTML-Datei in dem *templates* Ordner des *resources* Ordners identifizieren. In diesem Fall soll die Datei offer.qute.html die OfferWebResource repräsentieren. Also wird hier der erste Teil der Datei, nämlich „offer“, als Identifikator verwendet.

Jede WebResource der Anwendung hat nicht mehr als eine mit @GET annotierte Methode. Diese hat immer den Zweck serverseitige Daten in den Dateibereich des Templates zu schreiben. In diesem Beispiel werden alle Angebote basierend auf den potenziell vorliegenden Filterparametern eingeholt. Danach werden die Fahrer IDs mithilfe des injizierten UserService auf ihren richtigen Namen gemappt. Daraufhin kann die UI die Namen der Fahrer anstelle der IDs anzeigen. Hierbei handelt es sich also lediglich um ein Mapping der bereitgestellten Daten nach außen um mehr Benutzerfreundlichkeit zu erreichen. Serverseitig bleiben die IDs weiterhin bestehen.

Außerdem entscheidet die Variable *newRequestsAvailable* darüber ob die eingeloggte Person, sofern diese ein Fahrer mit einem offenen Angebot ist, ob unbeantwortete Anfragen vorliegen. Das macht sich in der UI dadurch bemerkbar, dass in der Navigationsleiste das Menü, über welches der Nutzer zu seinen Angeboten gelangt, rot markiert wird.

Als letztes werden die gesammelten Daten in den Datenbereich des Templates geschrieben.

```java
@RequestScoped
@Path("/view/offers")
@Produces(MediaType.TEXT_HTML)
@Transactional(TxType.REQUIRES_NEW)
public class OfferWebResource {

    @Inject
    ManageOffers offerManager;

    @Inject
    JsonWebToken principal;

    @Inject
    UserService userService;

    private final Template page;

    public OfferWebResource(Template offers) {
        this.page = requireNonNull(offers, message:"page is required");
    }

    @GET
    public TemplateInstance getAllOffers(@QueryParam("sorting") String sorting,
            @QueryParam("driveDate") String driveDate,
            @QueryParam("destinationCampus") String destinationCampus, @QueryParam("seats") Integer seats) {

        String driverId = (String) principal.getClaim(claimName:"id");

        Collection<Offer> offers = offerManager.selectAllOffers(sorting, driveDate,
                destinationCampus, seats);
        offers.forEach(o -> o.setDriverId(userService.getUserName(o.getDriverId())));

        boolean newRequestsAvailable = offerManager.newRequestsAvailable(driverId);

        return page.data(key1:"newRequestsAvailable", newRequestsAvailable, key2:"offers",
                offers);
    }
```

Snippet 11: OfferWebResource für die Bereitstellung von serverseitigen Daten an die UI

Die offers.qute.html kann nun auf diese bereitgestellten Daten zugreifen. Dafür reicht es lediglich das Objekt beim Namen anzusprechen und die jeweiligen Objektvariablen aufzurufen. In Snippet 12 ist zu sehen, dass sich eine for-Schleife über alle Angebote mit *{#for offer in offers}* starten und das Angebot einer jeden Iteration mittels *{offer}* ansprechen lässt.

```
{#for offer in offers}
<div
  class="card"
  style="margin-top: 15px; margin-left: 15px; margin-right: 15px"
>
  <div class="card-header">Fahrer {offer.driverId}</div>
  <div class="card-body">
```

Snippet 12: Nutzung der Collection offers in offers.qute.html

Im gleichen Zuge lässt sich *{#if}* sowie *{#else}* und *{#else if}* gleichermaßen verwenden. Auf diese Weise ist es möglich eine dynamische Webseite auf Basis der zugrunde liegenden Informationen immer anders aufzubauen. In Abbildung 2 ist beispielhaft das User Interface der Mitnahmeangebote zu sehen. Wie sich erkennen lässt wurde darauf geachtet, dass der Nutzer von jeder Stelle und zu jedem Zeitpunkt der Anwendung an eine andere Stelle gelangen kann.

So lässt sich der Homescreen über einen Klick auf den StudCar Schriftzug erreichen. Die eigenen Angebote als Fahrer lassen sich über das in dem Fall rot markierte Task Icon einsehen. Neue Angebote können über das Plus Icon auf der äußeren rechten Seite erstellt werden.

Weiterhin bietet das User Interface in diesem Beispiel auch diverse Filtermöglichkeiten. Nach dem Setzen eines oder mehrerer Filter müssen diese über die Schaltfläche „Finden“ auf der rechten Seite aktiv gemacht werden. Daraufhin lädt die Seite samt gesetzten Query Parametern neu und es werden nur Angebote angezeigt, die auf diese Filterparameter zutreffen.

Unter der Filterleiste sind alle passenden Fahrangebote mit den wesentlichen Informationen aufgelistet. Durch einen Klick auf die Schaltfläche „Mitnahmeanfrage stellen“ gelangt der Nutzer in die Übersicht, in der eine Mitnahmeanfrage zu genau diesem Angebot erstellt werden kann.

Dieses User Interface lässt sich in einer ähnlichen Form auch in den Mitnahmeanfragen, die von den Mitfahrern ausgehen, finden. Aus diesem Grund wird nicht weiter auf die Implementationen der User Interfaces außerhalb der OfferWebResource eingegangen.

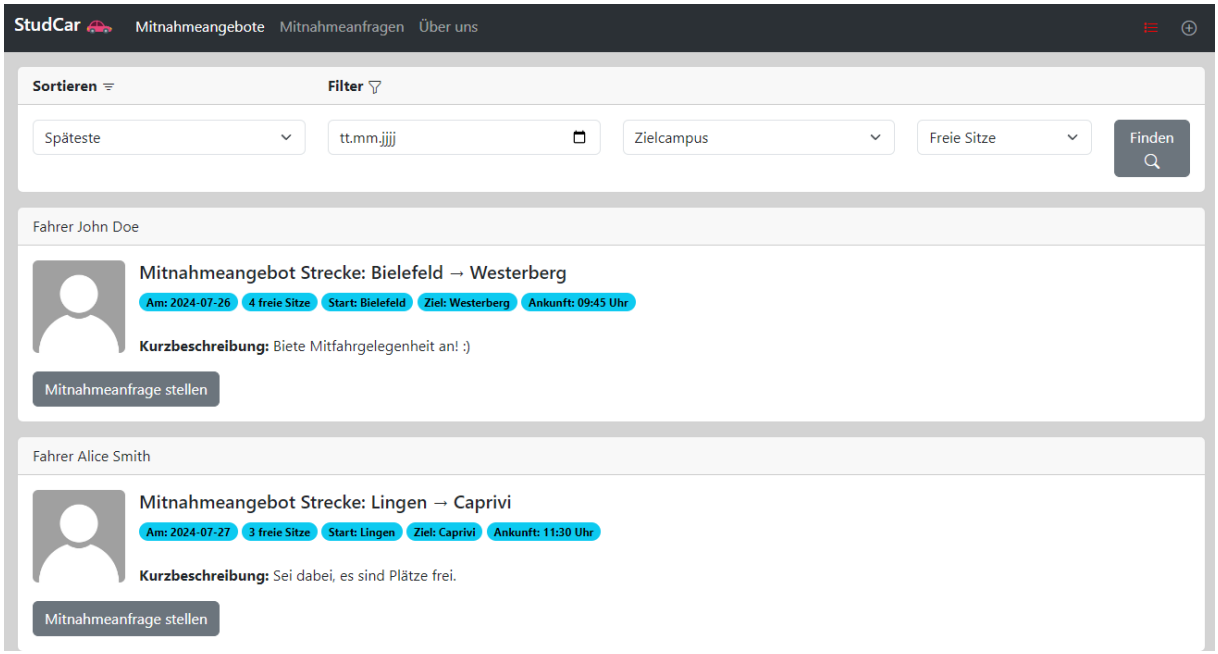


Abbildung 2: Ausschnitt aus dem User Interface der Mitnahmeangebote

4.5 Implementation des UserService

Der UserService stellt Informationen über die Nutzer bereit, die nur im Keycloak verfügbar sind. Der Keycloak stellt dabei eine REST-Schnittstelle für administrative Funktionalitäten zur Verfügung. Auf Nutzerinformationen zuzugreifen ist eine dieser Funktionen. Dazu muss ein REST-Client erstellt werden, der die Zugriffe auf diese externe Schnittstelle des Keycloaks durchführt.

Quarkus bietet für die Administration eines Keycloaks eine Implementation in Form des Keycloak Admin Clients, die Zugriffe auf die REST-Schnittstelle implementiert. Diese Implementation ist jedoch von der RESTEasy-Jackson Bibliothek abhängig, die sonst in der Anwendung nicht verwendet wird und die Abhängigkeit zu anderen Implementationen damit steigen würde. Aus diesem Grund wird auf die bestehende Implementation verzichtet und stattdessen ein Eclipse Microprofile Rest Client verwendet um die Zugriffe ohne starke Abhängigkeiten zu implementieren.

Die Admin-REST-API des Keycloaks ist abgesichert und es wird ein gültiges Access-Token benötigt. Dieses kann durch den Aufruf eines Endpunktes beim Keycloak erhalten werden, falls dieser mit gültigen Anmeldedaten aufgerufen wird. Dazu zählen die Client-ID und das Client-Secret sowie ein gültiger Benutzername mit dem korrekten Passwort. Damit diese Daten schnell und einfach verändert werden können werden die Informationen aus der Konfiguration der Anwendung ausgelesen. Daraufhin müssen diese Daten korrekt formatiert im Body des POST-Requests mitgeschickt werden. Als Antwort gibt es ein JSON-Objekt, mit dem Access-Token.

Nach dem Erhalt des Access-Token kann auf die Schnittstelle des Keycloaks zugegriffen werden, indem der Bearer-Header mit dem Token gesetzt wird. So kann der Zugriff authentifiziert werden. Zusätzlich benötigt der Nutzer die richtigen Rollen im Keycloak, sodass der Nutzer nicht nur authentifiziert, sondern auch autorisiert ist auf die jeweilige Schnittstelle zuzugreifen. Bei der erforderlichen Rolle handelt es sich um die Rolle „view-users“ des „realm-management“-Clients. Mit dieser Rolle ist es dem Nutzer erlaubt alle Nutzer im Keycloak einzusehen. So wird im Keycloak sichergestellt, dass nicht jeder Nutzer Zugriff auf alle Informationen erhält und die Zugriffe können sehr genau gesteuert werden. Demnach kann der Zugriff auf die Administrationsschnittstelle des Keycloaks nicht mit einem beliebigen Nutzer durchgeführt werden, sondern es wird ein Administrationsnutzer benötigt, der die erforderliche Rolle zugewiesen hat.

Sind alle beschriebenen Anforderungen erfüllt gibt der Keycloak ein JSON-Objekt mit den Informationen des angefragten Nutzers zurück. Dabei gibt es Informationen, die jeder Nutzer haben kann, wie Vorname, Nachname und E-Mail und zusätzliche Attribute, die als Key-Value-Pairs abgelegt werden. Die Telefonnummer ist hierbei ein Beispiel für beliebige Informationen, die noch im Nutzer gespeichert werden könnten.

Da die Nutzerinformationen immer auf die gleiche Weise eingeholt und in verschiedenen Modulen verwendet werden, wird der UserService als eine geteilte Komponente direkt an den benötigten Stellen eingebunden. Dies erfolgt über ein Interface, sodass der UserService für den Keycloak durch eine alternative Implementation ausgetauscht werden könnte ohne Änderungen im Code vornehmen zu müssen.

4.6 Implementation von Tests

Getestet wurde die Anwendung mithilfe von Rest Assured. Hierbei werden HTTP-Requests an den Server gesendet, der daraufhin mit einer Response antwortet. Diese Response wird nach dem Prinzip der erwarteten und erhaltenen Ausgabe ausgewertet.

Das Testen erfolgt in separaten dafür vorgesehenen Klassen. So werden beispielsweise sämtliche Tests hinsichtlich eines Angebotes in der Klasse OfferResourceTest getestet. Diese Klasse ist mit @QuarkusTest annotiert was dafür sorgt, dass diese als reine Testklasse bekannt gemacht wird. Jede Methode der Testklasse wird daraufhin mit der Annotation @Test versehen, um kenntlich zu machen, dass es sich hierbei um eine Testmethode ohne Rückgabewert handelt.

In Snippet 13 ist die Testmethode für die Erstellung eines neuen Angebotes zu sehen. Hierbei wird über einen JSON-String, der so aufgebaut sein muss wie ein NewOfferDTO über den body an die boundary übergeben. Zusätzlich wird definiert, dass der erwartete Status Code 200 sein muss.

Wurde das sichergestellt werden alle bestehenden Angebote über die GET-Methode der /offer API angefragt. In den erhaltenen Daten wird dann geprüft, ob das vorher Angebote Angebot tatsächlich korrekt und erfolgreich erstellt wurde. Nur wenn das der Fall ist, ist der Test erfolgreich abgeschlossen. Auf diese Weise wurden auch die anderen Schnittstellen der verschiedenen Ressourcen getestet.

```
@Test
public void testCreateOffer() {
        given()
                        .body("{\"startLocation\": \"Herford\", \"destinationCampus\": \"Lingen\", "
                                        + "\"driveDate\": \"2024-08-01\", \"arrivalTime\": \"08:37\", \"freeSeats\": 2, \"description\": "
                                        + "\"Biete Mitfahrgelegenheit an! Steigt gerne ein.\"}")
                        .header(headerName:"Content-Type", headerValue:"application/json")
                        .when()
                        .post(path:"/offers")
                        .then()
                        .statusCode(expectedStatusCode:200);

        given()
                        .when().get(path:"/offers")
                        .then()
                        .statusCode(expectedStatusCode:200)
                        .body(path:"[3].id", is(value:21),
                                        ...additionalKeyMatcherPairs:"[3].startLocation", is(value:"Herford"),
                                        "[3].destinationCampus", is(value:"Lingen"),
                                        "[3].driveDate", is(value:"2024-08-01"),
                                        "[3].arrivalTime", is(value:"08:37"),
                                        "[3].freeSeats", is(value:2),
                                        "[3].active", is(value:true),
                                        "[3].description",
                                        is(value:"Biete Mitfahrgelegenheit an! Steigt gerne ein."));
}
```

Snippet 13: Die testCreateOffer Methode der OfferResourceTest Klasse

Durch die Identitätsverwaltung mithilfe des Keycloak entsteht bei den Tests eine besondere Schwierigkeit. Die Ressource erwartet, dass ein angemeldeter Nutzer mit einem gültigen Access-Token auf die Schnittstelle zugreift und damit Daten, wie die Nutzer-ID vorliegen.

Eine Möglichkeit wäre für jeden Test ein gültiges Access-Token vom Keycloak abzurufen und damit die Tests zu ermöglichen, was jedoch einen erheblichen Mehraufwand für die Tests bedeutet. Aus diesem Grund wird die Authentifizierung in der Konfiguration der Anwendung für die Tests ausgeschaltet.

Während der Tests werden demnach keine Access-Tokens von der Schnittstelle überprüft. Dennoch muss eine Nutzer-ID in vielen Fällen vorliegen. Um diesem Problem Abhilfe zu schaffen, wird das JsonWebToken mit den Nutzerinformationen gemockt. Es wird dafür eine Mock-Klasse erstellt, die das JsonWebToken-Interface auf simple Weise implementiert.

Ein Objekt dieser simplen Klasse, die vorgegebene Werte zurückliefert, wird dann mit Hilfe von QuarkusMock im Testumfeld bekannt gemacht. Auf diese Weise liegen auch ohne Authentifizierung über den Keycloak die vorgegebene Nutzerinformationen in der Schnittstelle vor und können für die Tests benutzt werden.

5 Abschlussbetrachtung

5.1 Zusammenfassung

Die Projektherausforderung bestand darin eine Anwendung zu entwickeln die es Studierenden ermöglicht Mitfahrgelegenheiten zur Hochschule Osnabrück zu organisieren. Ziel dabei war es die steigenden Fehlzeiten der Studierenden bei Vorlesungsveranstaltungen möglichst auszubremsen. So ergab sich die zentrale Frage wie es StudCar erreichen kann, dass Studierende diesen Service in Anspruch nehmen.

Durch die Darstellung der technischen Grundlagen wurde gezeigt, auf welchen Technologien die Anwendung beruht und wofür diese gebraucht werden. Dadurch wurde die Basis für das weitere Verständnis dieses Berichtes geschaffen.

Das Aufbaukapitel sorgte dann dafür, dass zu den verwendeten Technologien nun auch ein detaillierter Überblick über den Anwendungsaufbau vorliegt. Das wurde mithilfe der Klassendiagrammen in den Anhängen erreicht. Es wurde gezeigt, dass es sich beim Aufbau um das Konzept einer layered architecture handelt welche dafür sorgt, dass es eine möglichst hohe Kohäsion innerhalb einer Schicht gibt. Einhergehend damit wird dadurch zusätzlich eine möglichst lose Kopplung zwischen untereinander liegenden Schichten erreicht.

Das zentrale Element des Berichtes ist das Implementationskapitel. Dort wurde vorrangig darüber aufgeklärt, wie die einzelnen Schichten eines Moduls arbeiten und miteinander kommunizieren. Dieser Aufbau wurde zudem mit verschiedenen Code Snippets belegt. Die Code Snippets spiegeln repräsentativ die wichtigsten Stellen der Anwendung wider. Dabei wurden stets die Annotationen der Code Snippets erläutert. Die Authentifikation mit Keycloak hat verdeutlicht, wieso es wichtig ist, dass jeder Nutzer sich anmelden muss, um auf die für ihn vorgesehenen Daten zugreifen zu können.

Abschließend gab es einen Einblick in die Implementation und den Aufbau des User Interfaces. Dort wurde unter anderem gezeigt wie sich Daten vom Server in eine TemplateInstance laden und auf der Frontend Seite auslesen lassen.

Zum Schluss wurde darauf eingegangen, wie die Anwendung getestet wurde und was einen erfolgreichen Test ausmacht.

5.2 Ausblick

Die Anwendung ist in ihrer jetzigen Form noch nicht perfekt und es gibt mehrere Möglichkeiten diese zu erweitern. Zum Beispiel wäre es eine großartige Möglichkeit sich über die Plattform mit der Fahrgemeinschaft in größerem Umfang austauschen zu können in Form eines Chats unter den Mitgliedern der Fahrgemeinschaft, um sich besser verabreden zu können. Ein erster Schritt in diese Richtung wäre die Bereitstellung von E-Mail-Adressen oder Telefonnummern für die Fahrgemeinschaft, die aktuell nur im eigenen Profil zu sehen ist. Andere Nutzer haben dort jedoch aktuell keinen Zugang.

Außerdem könnten zusätzliche Informationen, wie die Adresse aus dem Keycloak als Identity Provider geholt werden und für die Autovervollständigung des Abholhortes verwendet werden. Ebenso könnte auf diese Weise ein persönliches Profilbild referenziert werden.

Zusätzlich fehlt aktuell eine Auflistung der anstehenden Fahrten. Also eine explizite Auflistung der angenommenen Angebote, sowie der akzeptierten Anfragen. In diesem Zusammenhang wären auch Benachrichtigungen für anstehende Fahrten spannend.

5.3 Fazit

Durch den Fakt der immer größer werdenden Digitalisierung lässt sich festhalten, dass eine gut strukturierte Webanwendung sehr wohl dazu beitragen kann das Interesse vieler Studierenden hinsichtlich der Organisation einer Mitfahrgelegenheit zu wecken. Viele Studierende setzen sich bei Ausfällen von öffentlichen Verkehrsmitteln nicht gerne mit Alternativen auseinandersetzen. Das reicht bis zum Punkt an dem ganze Vorlesungstage gemieden werden.

StudCar ist eine Lösung von Studierenden der Hochschule Osnabrück für die Studierenden der Hochschule Osnabrück und setzt dabei vollständig auf den Zusammenhalt der Studierenden untereinander. Gemischt mit einer nicht kommerziellen Plattform, die auf reiner Wohltätigkeit erstellt wurde, wird es dazu beitragen, dass viele Studierende die regelmäßig von einem solchen Szenario betroffen sind diesen Service in Anspruch nehmen werden. Daraus ergibt sich folglich die Erfüllung des zentralen Zieles, nämlich der Verringerung von Ausfalltagen der Studierenden bei Vorlesungsveranstaltungen.

6 Referenzen

Die Webseiten wurden zuletzt am 22.07.2023 aufgerufen.

[@MS1] Microsoft, Extension Pack for Java, https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack

[@QKS1] Quarkus, RESTEasy Classic, https://quarkus.io/guides/resteasy

[@QKS2] Quarkus, Using Hibernate ORM and Jakarta Persistence, https://quarkus.io/guides/hibernate-orm

[@QKS3] Quarkus, Qute Templating Engine, https://quarkus.io/guides/qute

[@QKS4] Quarkus, Using OpenID Connect (OIDC) and Keycloak to centralize authorization, https://quarkus.io/guides/security-keycloak-authorization

[@QKS5] Quarkus, Using OpenAPI and Swagger UI, https://quarkus.io/guides/openapi-swaggerui

[@QKS6] Quarkus, Testing your application, https://quarkus.io/guides/getting-started-testing

[@RO1] Roosmann, Prof. Dr. Rainer, Vorlesungsskript Domain Driven Design – Taktisches Design, Hochschule Osnabrück, März 2023

[@JVX1] Oracle Docs, Annotation Type Transactional, https://docs.oracle.com/javaee/7/api/javax/transaction/Transactional.html

[@JVX2] Oracle Docs, Annotation Type Path, https://docs.oracle.com/javaee/6/api/javax/ws/rs/Path.html

[@JVX3] Oracle Docs, Using @Consumes and @Produces to Customize Requests and Responses, https://docs.oracle.com/cd/E19798-01/821-1841/gipzh/index.html

[@JVX4] Oracle Docs, Annotation Type Valid, https://docs.oracle.com/javaee/7/api/javax/validation/Valid.html

[@JVX5] Oracle Docs, Annotation Type NotNull, https://docs.oracle.com/javaee/7/api/javax/validation/constraints/NotNull.html

[@JVX6] Oracle Docs, Annotation Type ApplicationScoped, https://docs.oracle.com/javaee/6/api/javax/enterprise/context/ApplicationScoped.html

[@JVX7] Oracle Docs, Annotation Type Entity, https://docs.oracle.com/javaee/7/api/javax/persistence/Entity.html

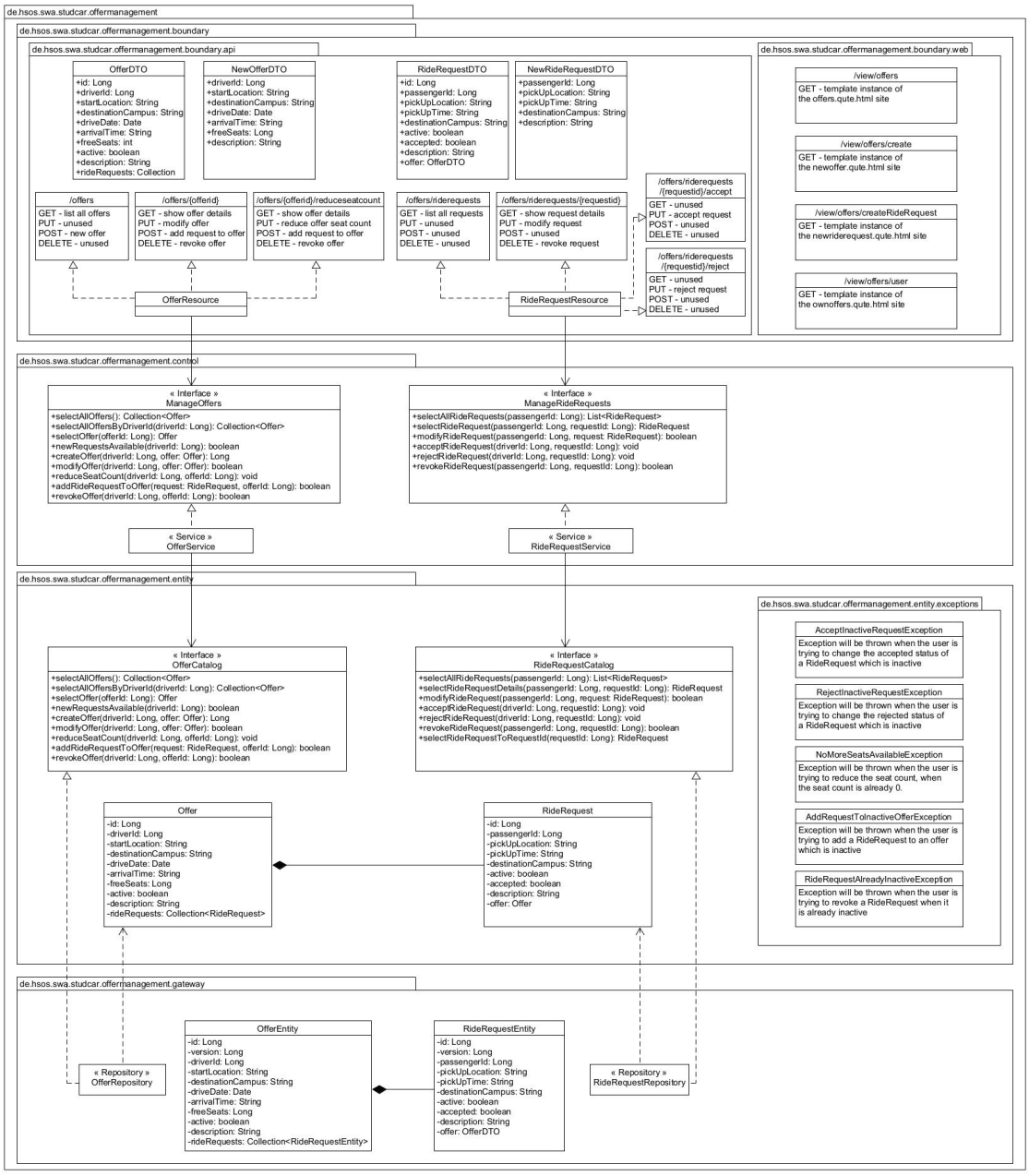
[@JVX8] Oracle Docs, Annotation Type Id, https://docs.oracle.com/javaee/6/api/javax/persistence/Id.html

[@JVX9] Oracle Docs, Annotation Type GeneratedValue, https://docs.oracle.com/javaee/6/api/javax/persistence/GeneratedValue.html

[@JVX10] Oracle Docs, Annotation Type Observes, https://docs.oracle.com/javaee/6/api/javax/enterprise/event/Observes.html

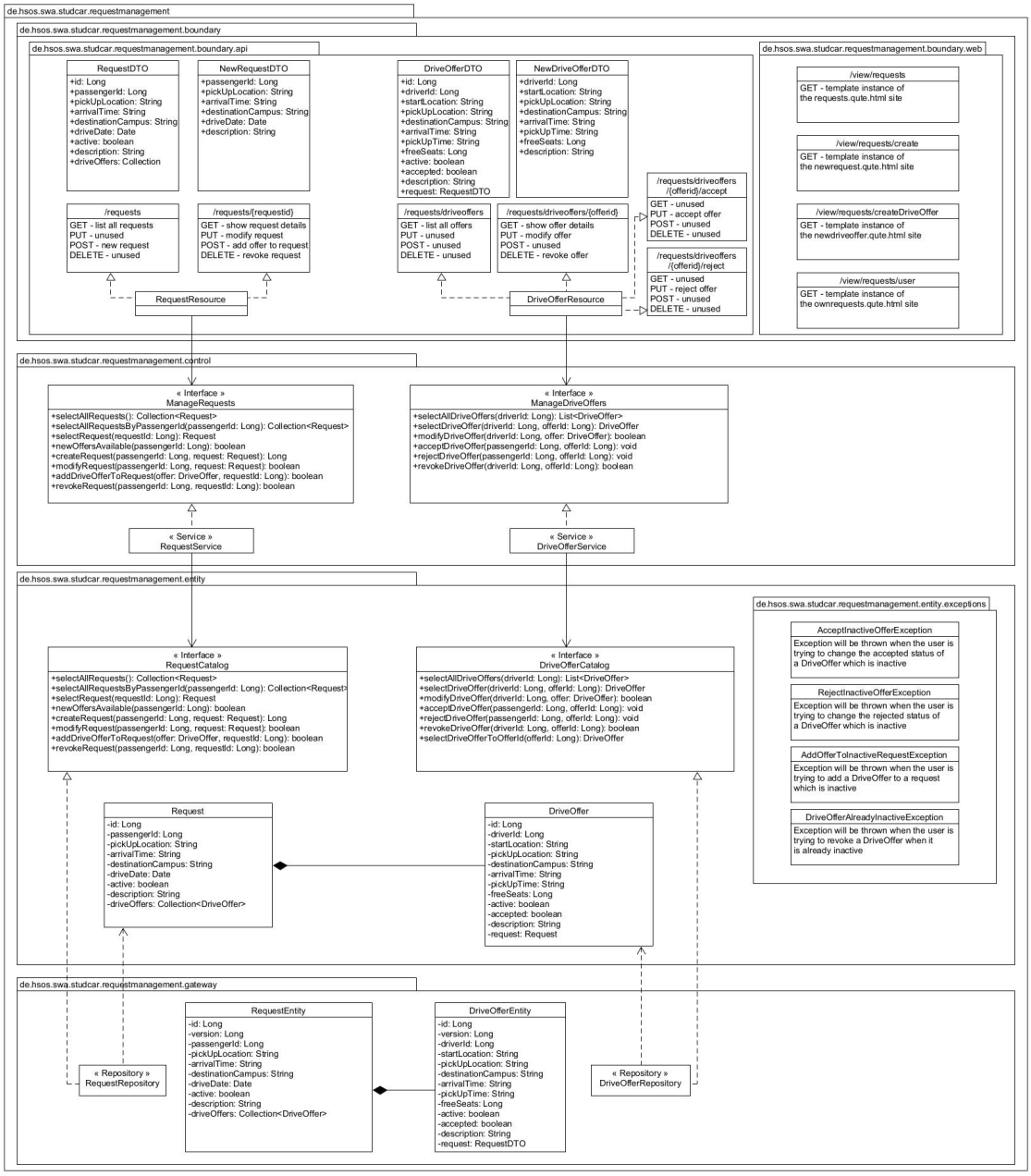
7 Anlagen

A Klassendiagramm Modul Offermanagement



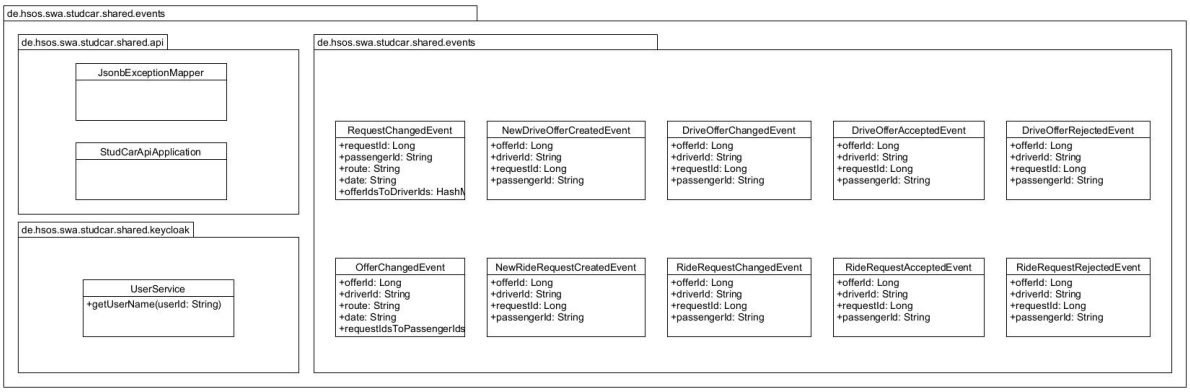
Anlage 1: Klassendiagramm Modul Offermanagement

B Klassendiagramm Modul Requestmanagement



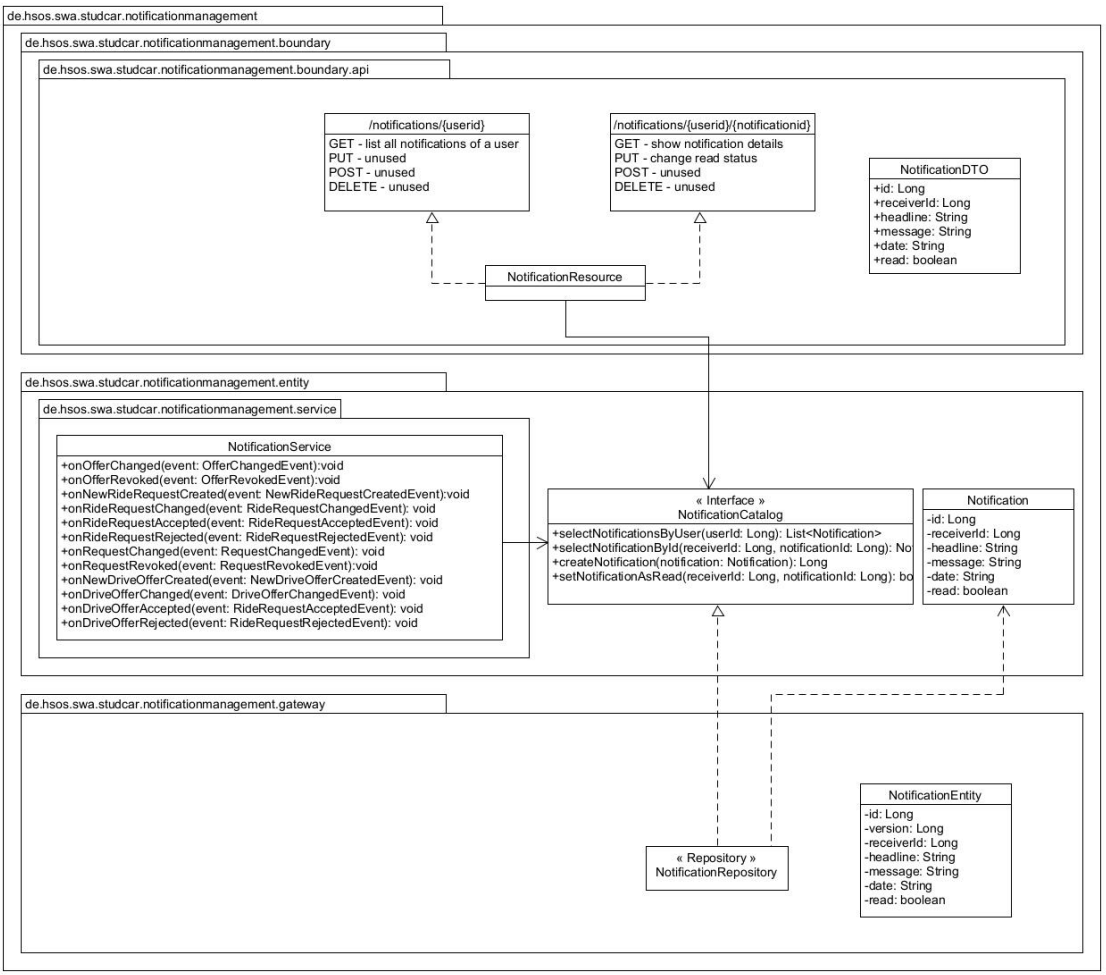
Anlage 2: Klassendiagramm Modul Requestmanagement

C Klassendiagramm Modul Shared



Anlage 3: Klassendiagramm Modul Shared

D Klassendiagramm Modul Notificationmanagement



Anlage 4: Klassendiagramm Modul Notificationmanagement

E Arbeitsaufteilung

Projektbeteiligter	Aufgabe	Umsetzung	Bericht
Hendrik Purschke	Autorisierung und Security mit Keycloak	X	X
Andreas Morasch	Implementation Notificationmanagement Modul	X	X
beide	Erstellen der Klassendiagramme	X	X
	Implementation Offermanagement Modul	X	X
	Implementation Requestmanagement Modul	X	X
	Implementation des User Interfaces	X	X

Anlage 5: Tabelle der Arbeitsaufteilung

Eidesstattliche Erklärung

Hiermit erkläre ich/ erklären wir an Eides statt, dass ich / wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe / haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)

Urheberrechtliche Einwilligungserklärung

Hiermit erkläre ich/ Hiermit erklären wir, dass ich/wir damit einverstanden bin/sind, dass meine/ unsere Arbeit zum Zwecke des Plagiatsschutzes bei der Fa. Ephorus BV bis zu 5 Jahren in einer Datenbank für die Hochschule Osnabrück archiviert werden kann. Diese Einwilligung kann jederzeit widerrufen werden.

....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)

Hinweis: Die urheberrechtliche Einwilligungserklärung ist freiwillig.