



Digitalisierung von Reisen durch Entwicklung einer intelligenten Reiseführer App

Studienarbeit

im Rahmen der Prüfung zum
Bachelor of Science (B.Sc.)

des Studienganges Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Joshua Schulz und Raphael Müßeler

2020

-Sperrvermerk-

Abgabedatum:	18. Mai 2020
Bearbeitungszeitraum:	01.10.2019 - 18.05.2020
Matrikelnummer, Kurs:	4508858, 6801150, TINF15B1
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Gutachter der Dualen Hochschule:	Thorsten Schlachter

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema:

Digitalisierung von Reisen durch Entwicklung einer intelligenten Reiseführer App

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 24. April 2020

Schulz, Joshua; Müßeler, Raphael

Abstract

- English -

This is the starting point of the Abstract. For the final bachelor thesis, there must be an abstract included in your document. So, start now writing it in German and English. The abstract is a short summary with around 200 to 250 words.

Try to include in this abstract the main question of your work, the methods you used or the main results of your work.

Abstract

- *Deutsch* -

Dies ist der Beginn des Abstracts. Für die finale Bachelorarbeit musst du ein Abstract in deinem Dokument mit einbauen. So, schreibe es am besten jetzt in Deutsch und Englisch. Das Abstract ist eine kurze Zusammenfassung mit ca. 200 bis 250 Wörtern.

Versuche in das Abstract folgende Punkte aufzunehmen: Fragestellung der Arbeit, methodische Vorgehensweise oder die Hauptergebnisse deiner Arbeit.

Inhaltsverzeichnis

Abkürzungsverzeichnis	VI
Abbildungsverzeichnis	VIII
Tabellenverzeichnis	IX
Quellcodeverzeichnis	X
1 Einleitung	1
1.1 Motivation	2
1.2 Gliederung der Arbeit	2
2 Grundlagen	4
2.1 RESTful APIs	4
2.2 Frameworks	5
2.3 Build Management Tools	15
2.4 Qualitätssichernde Maßnahmen	17
3 Anforderungen	23
3.1 Visionen und Ziele	23
3.2 Rahmenbedingungen	25
3.3 Geforderte Eigenschaften	26
4 Datengrundlage	33
4.1 Points of Interest	33
4.2 Weiterführende Informationen zu POIs und Städten	36
4.3 Kartendaten	37
5 Konzept	39
5.1 Kommunikationsschema	39
5.2 Server Architektur	43
5.3 Client Architektur	47
5.4 Infrastruktur	49
6 Implementierung	51
6.1 Controller Interfaces	51
6.2 Authentifizierung	52
6.3 Namenszuordnung zwischen den APIs	54
6.4 Umgehung der API Beschränkung für DBpedia	55

7	Evaluation	58
7.1	User Zufriedenheit	58
7.2	Aufbau	58
7.3	Ablauf	58
7.4	Ergebnis	58
8	Fazit	59
8.1	Ausblick	59
	Literaturverzeichnis	XI

Abkürzungsverzeichnis

AOP	Aspect-oriented Programming
API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
DSL	Domain Specific Language
DTO	Data Transfer Objects
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IO	Input/Output
IoC	Inversion of Control
JPA	Java Persistence API
JSON	JavaScript Object Notation
JTA	Java Transaction API
JVM	Java Virtual Machine
MIME	Multipurpose Internet Mail Extension
MVC	Model View Controller
OAI	OpenAPI
ORM	Object-relational Mapping
ORS	OpenRouteService
POI	Point of interest
POJO	Plain Old Java Object
QoS	Quality of Service
REST	Representational State Transfer

RPC	Remote Procedure Call
SDK	Software Development Kit
SOAP	Simple Object Access Protocol
SpEL	Spring Expression Language
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
WWW	World Wide Web
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Abbildungsverzeichnis

2.1	SwaggerHub Darstellung anhand des Projektes <i>Travlyn</i>	7
2.2	Docker Virtualisierung verglichen mit virtuellen Maschinen [19]	13
2.3	Darstellung der für das Projekt <i>Travlyn</i> erstellten Jenkins Pipeline	21
3.1	Darstellung aller geplanten Use Cases mit den assoziierten Nutzerprofilen. . .	28
5.1	Swaggerbeschreibung der Stopentität	40
5.2	Beschreibung des <i>getStop</i> Requests im Swagger UI	42
5.3	Beschreibung des <i>getStop</i> Requests im Swagger UI	43
5.4	Struktur des <i>Travlyn</i> Servers als UML Diagramm. Zu Beachten ist, dass dieses Diagramm stark vereinfacht und nicht vollständig ist, um die Übersichtlichkeit zu wahren.	46
5.5	Aufbau und Struktur der Benutzeroberfläche anhand eines konkreten Use Cases	48
5.6	Ausschnitt aus dem Datenbankschema von <i>Travlyn</i>	50
6.1	Authentifizierungsprozess von <i>Travlyn</i> mittels Spring Security Filter Chain . .	53
6.2	Flussdiagramm zum Ablauf des <i>lazy fetching</i> von DBpedia.	56

Tabellenverzeichnis

4.1	Gegenüberstellung der vorliegenden Alternativen zur Abfrage der POIs	36
-----	--	----

Quellcodeverzeichnis

6.1	Nutzung der beschriebenen Annotationen eines Controllers am Beispiel der City API	51
-----	---	----

1 Einleitung

(Joshua)

In der heutigen Zeit unterliegt die Welt der Medien einer sehr rasanten und starken Veränderung. Es werden immer mehr neuartige und innovative Techniken entwickelt, wie Medien konsumiert werden können, z.B. Augmented und Virtual Reality. Durch diese Techniken wird versucht die Mediennutzung effektiver, intensiver und moderner zu machen. In vielen Bereichen haben diese Techniken bereits Einzug gehalten und sind für eine große Masse von Konsumenten verfügbar, z.B. Virtual Reality gaming mit Hilfsmitteln wie Oculus Rift [1] und anderen Produkten.

Allerdings gibt es ebenfalls Bereiche bei denen die Digitalisierung und Nutzung neuer Techniken nicht komplett ausgenutzt werden und viele weitere Vorteile ungenutzt bleiben, wie z.B. beim Reisen [2]: Viele Menschen nutzen Reiseführer, um sich einen Überblick über ihre Reisedestination zu verschaffen und einen grundlegenden Plan zu erstellen, allerdings findet man diese Reiseführer fast ausschließlich in gedruckter Buchform. Das bedeutet für die Buchform, dass immer schwere Printmedien mit in den Urlaub genommen werden müssen, dass Informationen in den Reiseführern nicht aktualisiert werden können, ohne eine neue Auflage herauszugeben und eine interaktive Gestaltung der Medien nicht möglich ist. Außerdem sind die Seiten in einem Buch begrenzt, d.h. es muss für jede Reise ein neuer Reiseführer erworben werden, der spezielle Informationen zum Reiseziel enthält. All dies sind Nachteile, die durch eine Digitalisierung der Inhalte ausgeglichen werden könnten: Das Smartphone ist heutzutage ein ständiger Begleiter, der ausgenutzt werden kann, um Echtzeitinformationen schnell und immer aktuell zur Verfügung zu stellen. Außerdem können interaktive Elemente wie Karten, Navigation, Audioguides uvm. direkt integriert werden. Es wäre möglich eine Anwendung zu schaffen, die in der Lage ist, für jede beliebige Stadt und Region auf der Welt Informationen bereit zu stellen, ohne dass neue Inhalte erworben werden müssen. Als Geschäftsmodell des Anbieters wäre es denkbar Geld durch den Einsatz von Werbung zu verdienen oder durch Bezahlung von Städten im Gegenzug für besondere Herausstellung innerhalb der Anwendung.

Damit könnte eine Reise entstehen, die unkomplizierter und trotzdem viel aktueller ist als bei Benutzung eines herkömmlichen Reiseführers.

Es könnten viele Services, die aktuell parallel zum Reiseführer genutzt werden (z.B. verschiedene Bewertungsportale und Karten) direkt integriert werden, um alle Informationen auf einen Blick zur Verfügung zu stellen. Ebenso könnte eine Personalisierung der verfügbaren Daten umgesetzt werden. Im Gegensatz zum herkömmlichen Reiseführer, welcher allgemeine und damit u.U. viele für den einzelnen irrelevant Informationen enthält, werden Vorschläge anhand der vom Nutzer gesetzten Vorlieben gemacht und somit nur nützliche Informationen zur Verfügung gestellt.

Insgesamt soll ein digitaler Reiseführer entstehen, der das Reiseerlebnis auf eine bessere, digitalere und einfachere Ebene hebt und noch mehr Spaß am Reisen erzeugen kann.

1.1 Motivation

Wir möchten mit dieser Arbeit einen Beitrag zu einer digitalisierten und technisch geprägten Gesellschaft leisten, indem wir eine Anwendung schaffen, die mit den Nachteilen von gedruckten Reiseführern aufräumt und eine neue innovative Art des Reisen schafft. Damit soll das Reiseerlebnis der Menschen verbessert werden und ihnen die Möglichkeit geben sich noch mehr auf das Erlebte zu konzentrieren, ohne Gedanken an eine komplizierte Planung, bei der viele Medien parallel genutzt werden, zu verschwenden. Außerdem ist diese Arbeit Teil unserer Prüfung zum Bachelor of Science und dient zur praktischen Anwendung der bisher im Studium erlernten Fähigkeiten und zum Ausprobieren neuer innovativer Techniken um unser Wissen zu erweitern.

1.2 Gliederung der Arbeit

In dieser Arbeit soll die durchgeführte Entwicklung detailliert beschrieben und festgehalten werden. Dazu werden alle wichtigen Arbeitsschritte aufgeschlüsselt und die

Ergebnisse entsprechend dargestellt. Die entwickelte App trägt den Namen *Travlyn* und wird im Verlauf dieser Arbeit nur noch so bezeichnet. Dieser Name entstand bei Diskussionen der Teammitglieder und wurde für markenfähig befunden.

An erster Stelle steht die Beschreibung der verwendeten Techniken, Prozessen und Begriffen. In diesem Teil werden alle, für diese Arbeit, relevanten technischen Grundlagen genannt und Verweise auf weiterführende Literatur geliefert um, eine gemeinsame Ausgangslage zu schaffen. Anschließend werden die Anforderungen an die zu entwickelnde App geschildert und Ziele gesetzt, die erreicht werden sollen. Danach wird auf ein sehr zentrales Thema dieser Entwicklung eingegangen: Die Datenquellen. Ein Reiseführer lebt von den präsentierten Daten und somit ist die Auswahl der Datenquellen anhand der Anforderungen von großer Bedeutung. Ein ähnliches Gewicht hat das Erstellen eines Übergeordneten Konzepts für die Anwendung und dessen Spezialisierung auf kleinere Komponenten der Software. Das Ergebnis dieses Prozesses wird im Anschluss an die Datenquellen ausführlich dargestellt. Nach der Konzeption folgt die Durchführung und Implementation, von der nur einige zentrale und/oder kritische Punkte dargestellt werden. Abschließend soll das entstandene Ergebnis anhand bestimmter Verfahren evaluiert und gegen die zu Beginn definierten Ziele validiert werden. Nach dieser Evaluation mündet diese Arbeit in einem Fazit, welches einen Ausblick auf zukünftige Erweiterung und Verbesserungen gibt und zentrale Punkte, die das Team aus diesem Projekt mitgenommen hat, darstellt.

2 Grundlagen

In der vorliegenden Arbeit werden einige Technologien betrachtet und verwendet, welche im Folgenden zum weiteren Verständnis der Arbeit erläutert werden. Dazu zählen zum einen Paradigmen und Entwurfsmuster, und zum anderen deren Umsetzung unter Verwendung verschiedener Frameworks und Build Management Tools.

2.1 RESTful APIs

Representational State Transfer beschreibt ein architektonisches Modell, um Web Services zu erstellen. Sogenannte REST Web Services bieten Interoperabilität zwischen Computersystemen im Internet – geeignet für die Kommunikation von Maschine zu Maschine. So lässt sich REST auch als eine Abstraktion der Struktur und des Verhaltens des World Wide Webs beschreiben.

Neben REST gibt es weitere Alternativen, wie SOAP oder RPC; der Vorteil von REST besteht jedoch darin, dass durch das WWW ein Großteil der Infrastruktur für die Kommunikation bereits vorhanden und implementiert ist. Während bei acsRPC in der URI Methodeninformationen enthalten sind, gibt eine URI in der REST Architektur ausschließlich Ressourcen an und kodiert die Funktionalität mittels HTTP Methoden. Dieser Ansatz entspricht dem Konzept einer URI, da bei einer HTTP Anfrage ebenso nur Ressourcen und keine Funktionalität gefragt ist.

Eine REST API muss insgesamt sechs Eigenschaften besitzen:

Client-Server Architektur Bei REST gilt im Allgemeinen, dass eine Client-Server Architektur vorliegen soll: Der Client konsumiert die vom Server bereitgestellten Dienste.

Zustandslosigkeit Eine RESTful API hat keine Zustände, sondern ist so konzipiert, dass alle benötigten Informationen in einer REST-Nachricht enthalten sind. Dies begünstigt außerdem die Skalierbarkeit eines solchen Dienstes, da es auf diese

Weise einfacher ist, alle eingehenden Anfragen auf mehrere Instanzen zu verteilen.

Caching Server sowie Client können Antworten zwischenspeichern. Es muss jedoch vorher explizit definiert werden, welche Antworten zwischengespeichert bzw. welche nicht zwischengespeichert werden, um zu verhindern, dass ungültige Daten – Daten, deren Lebensdauer überschritten ist oder welche nicht mehr aktuell sind – versendet werden.

Einheitliche Schnittstelle Die REST API muss eine einheitliche Schnittstelle zur Verfügung stellen, welche den von [3] definierten Anforderungen entsprechen muss. Dies vereinfacht die Nutzung der API.

Mehrschichtige Systeme Die Struktur einer RESTful API soll mehrschichtig sein, so dass es ausreicht, dem Client lediglich die oberste Schicht als Schnittstelle anzubieten. Die Architektur der API wird dadurch simplifiziert und die dahinterliegenden Schichten der Implementierung bleiben verborgen.

Code on Demand Fielding beschreibt diese Eigenschaft als optional: Der Server kann, durch das Übertragen von ausführbarem Code, die Funktionalität des Clients zeitweise erweitern oder anpassen. Vorstellbar wäre beispielsweise eine Übertragung von bereits kompilierten Komponenten oder Client-seitigen Skripten. Dies gilt es jedoch mit Vorsicht zu genießen, da diese Funktionalität auch Sicherheitslücken bergen und somit eine geeignete Angriffsmöglichkeit bieten kann.

2.2 Frameworks

2.2.1 OpenAPI

Der OpenAPI Standard ist Open Source und dient der Beschreibung von RESTful APIs. Bis 2016 war OpenAPI Teil des Swagger Frameworks, wurde aber schließlich als separates Projekt unter Aufsicht der sog. *OpenAPI Initiative* [4] ausgelagert.

Mit der deklarativen Ressourcenspezifikation von OpenAPI können Clients Dienste verstehen und konsumieren, ohne über Kenntnisse der eigentlichen Server-Implementierung bzw. Zugriff auf den Servercode zu verfügen. Dies erleichtert die Entwicklung Client-

seitiger Applikationen, die RESTful APIs verwenden. Die OpenAPI-Spezifikation ist zudem sprachunabhängig und lässt sich in jeder Beschreibungssprache (YAML, XML etc.) definieren.

Zu dieser Spezifikation der API gehören unter anderem folgende Metadaten:

- Name der API
- Version
- Kurzbeschreibung
- Kontakt
- Lizenz

Wichtig ist hierbei vor allem die Version. Diese ermöglicht es, auch auf ältere Versionen der API zuzugreifen, sodass nach einem Update, nicht jeder Konsument dieser API zum einen die Version ändern und zum anderem eventuell auch Code anpassen muss.

Darüber hinaus lassen sich Schemata beschreiben, die das Datenmodell für Konsumenten repräsentieren. Diese Schemata sind mittels der JSON Schema Specification definiert. Es können verschiedene Objekte definiert werden, die mehrere Attribute mit einem definierten Datentypen haben. Dieser Datentyp kann wiederum ein Schema eines Objektes sein, sodass eine Verschachtelung von Objekten möglich ist.

Der Hauptteil der API Beschreibung entspricht der Definition der eigentlichen Schnittstelle. Hierbei können Pfade definiert werden, welche die API öffentlich zur Verfügung stellt. Pro Pfad – bzw. Ressource – lassen sich die möglichen HTTP Operationen definieren. Diese Definition beinhaltet die geforderten Parameter und möglichen Rückgabewerte bzw. -codes inkl. einer Beschreibung. Bei einer erfolgreichen Anfrage können die bereits definierten Schemata verwendet werden, um den Rückgabotyp zu definieren. So lassen sich alle möglichen Anfragen an die API klar definieren, sodass fehlerhafte Anfragen nicht zugelassen werden.

Die definierten Pfade lassen sich zudem in Tags zusammenfassen, sodass eine festgelegte Struktur in den Anfragen gewährleistet und außerdem die Komplexität der API reduziert wird.

2.2.2 Swagger

Swagger ist ein Framework, das sich des OpenAPI Standards bedient. Swagger bietet ein Tooling an, mit dessen Hilfe APIs spezifiziert und beschrieben werden können. Neben einem entsprechenden Editor, bietet Swagger die Möglichkeit, aus der Open-API Spezifikation Code zu generieren und zwar unter Verwendung unterschiedlicher Frameworks – z.B. eine vollständige Spring Applikation –, wobei nur noch die eigentliche Implementierung der Businesslogik erforderlich ist.

Des Weiteren stellt Swagger eine Web-basierte Benutzeroberfläche zur Verfügung, welche nicht nur die direkte Anbindung von Live APIs ermöglicht, sondern auch eine visuelle Dokumentation der API darstellt. [5]

Swagger stellt zudem die Plattform SwaggerHub öffentlich zur Verfügung. Auf dieser Plattform können API Spezifikationen veröffentlicht werden. Somit sind die Spezifikationen vieler verschiedener APIs auf einer Plattform zusammengefasst und können leicht eingesehen werden. Abbildung 2.1 zeigt die Darstellung der Plattform SwaggerHub.



Abbildung 2.1: SwaggerHub Darstellung anhand des Projektes *Travlyn*

2.2.3 Spring

Spring ist ein Open Source Framework, welches in Java geschrieben wurde. Es gilt als de facto Standard bei der Entwicklung von RESTful API, da es in der Open Source Welt viel Zuspruch und Verwendung gefunden hat. Zudem integriert Spring mit fast allen Java Umgebungen und ist somit nicht nur für Anwendungen im kleinen Maßstab, sondern eben so für Anwendungen in großen Unternehmen geeignet. [6]

Bei der Entwicklung dieses Frameworks wurden die unter anderem in [7] beschriebenen Design Prinzipien mithilfe der folgenden Module und deren entsprechender Funktionalität umgesetzt:

Dependency Injection

Der von Martin Fowler 2004 definierte Begriff der *Dependency Injection* in [8] ist eine Präzisierung oder Spezialisierung des Begriffs *Inversion of Control*. IoC bezeichnet ein Paradigma, welches den Kontrollfluss einer Applikation nicht mehr der Anwendung, sondern dem Framework – in diesem Fall Spring – überlässt. Ein Beispiel für IoC sind Listener (Beobachter Muster [9]).

Dependency Injection beschreibt ein Entwurfsmuster, bei welchem festgelegte Abhängigkeiten nicht zur Kompilierzeit, sondern zur Laufzeit bereitgestellt werden. Dies lässt sich einem Beispiel erläutern: Besteht bei der Initialisierung eines Objektes eine Abhängigkeit zu einem anderen Objekt, so wird diese Abhängigkeit an einem zentralen Ort hinterlegt. Wenn nun die Initialisierung dieses Objektes erfolgt, beauftragt es den sog. Injector (dt. Injezierer), die Abhängigkeit aufzulösen. [8]

In Spring bietet der IoC Container mittels Reflexion ein konsistentes Werkzeug zur Konfiguration sowie Verwaltung von Java Objekten. Diese durch den Container erstellten Objekten heißen *Beans*. Die Konfiguration des Containers erfolgt entweder über eine XML Datei oder über Java Annotationen. [6]

Aspektorientierte Programmierung

AOP beschreibt ein Paradigma und ermöglicht die klassenübergreifende Verwendung generischer Funktionalität. Dies führt zu einer starken Modularisierung und

sorgt für eine klare Trennung zwischen der Basislogik der Kernanwendung und der Businesslogik (Cross-cutting Concern). [10]

Das Schreiben von Logs stellt ein Beispiel für ein Cross-cutting Concern dar, da eine Logging-Strategie alle protokollierten Klassen und Methoden erfasst und somit durchaus mit der Basis- sowie der Businesslogik in Berührung kommt.

Transaktionsmanagement

Ein weiteres Beispiel für AOP ist sog. Transaktionsmanagement. Eine Transaktion bezeichnet in der Informatik eine logische Einheit, mit dessen Hilfe Aktionen auf einer Persistenz ausgeführt werden können. Dabei ist sichergestellt, dass sobald die Transaktion fehlerfrei und vollständig abgeschlossen ist, der Datenbestand weiterhin konsistent ist. Im Umkehrschluss bedeutet das, dass eine Transaktion entweder vollständig oder gar nicht ausgeführt wird. [11]

Das von Spring bereitgestellte Transaktionsmanagement stellt eine Abstraktion der Java Plattform dar und ist in der Lage mit globalen und verschachtelten Transaktionen sowie sog. Savepoints – ein Punkt innerhalb einer Transaktion, zu welchem im Fehlerfall zurück gesprungen werden kann – zu arbeiten. Außerdem lässt sich diese Abstraktion in fast allen Java Umgebungen einsetzen. Die von Java bereitgestellte Java Transaction API (JTA) [12] hingegen unterstützt nur globale und verschachtelte Transaktionen und erfordert zudem immer einen Applikationsserver.

Model View Controller

Das MVC (Model View Controller) Pattern ist ein weit verbreiteter Mechanismus zur Entwicklung von Benutzeroberflächen. MVC stellt ein Design Pattern dar, dass Kapselung sowie eine Struktur für eine Architektur von Benutzeroberflächen bietet und bei welcher jeder Bereich eine definierte Aufgabe hat. Eine Verletzung der Zuständigkeitsbereiche ist zu vermeiden. [9]

Das Pattern besagt, dass die Architektur von Benutzerschnittstellen in folgende Bereiche aufgeteilt ist: Das *Model* ist für den Zugriff auf die Datenbank und die Beschaffung von Daten zuständig. Häufig ist das Model auch für die Aufbereitung der Daten zuständig. Somit liegt die meist aufwändige Logik nicht beim Client, sondern bei den

Servern, welche zumeist auch mit besserer Hardware ausgestattet sind.

Ein *Controller* definiert die Art und Weise, wie die Benutzerschnittstelle auf die Eingaben des Benutzer reagiert. Des Weiteren ist ein Controller für das Aktualisieren der Daten im Datenmodell, aber auch auf dem View zuständig.

Der *View* bestimmt ausschließlich, wie die Benutzeroberfläche aussehen soll. Er enthält – in den meisten Implementierungen – keine Logik, sondern ist lediglich eine Definition und Anordnung der Benutzeroberflächenelemente.

Spring definiert für alle Verantwortlichkeiten eigene Strategie-Interfaces, wie beispielsweise das Controller Interface, welches alle eingehenden HTTP Requests definiert und darüber hinaus auch behandelt. Dadurch wird die strikte Kapselung der Verantwortlichkeiten, welche vom MVC Entwurfsmuster gefordert sind, gewährleistet und zwar auf Framework Ebene.

2.2.4 Hibernate

Hibernate [13] ist ein Persistenz- und Object-relational Mapping (ORM) Framework, das ebenfalls unter einer Open Source Lizenz veröffentlicht und in Java geschrieben ist. Hibernate bietet eine Abstraktionsstufe gegenüber relationalen Datenbankimplementationen. Mithilfe der Sprache *Hibernate Query Language* und dem entsprechend konfigurierten Dialekt (z.B. MySQL Dialekt, MariaDB Dialekt etc.) werden die entsprechenden Statements erzeugt und schließlich ausgeführt. Dies ermöglicht den einfach und schnellen Umstieg von einer Datenbankimplementation auf die andere, ohne Anpassung der sich im Code befindlichen Queries.

Object-relational Mapping

Eine häufig eingesetzte Technik der Persistierung sind relationale und meist auch SQL [14] basierte Datenbanken wie beispielsweise MySQL oder MariaDB. Wenn jedoch die Anwendung, welche die Businesslogik enthält, der Objektorientierung folgt, kommt es zu einem Widerspruch – dem sog. *Object-relational impedance mismatch* –, welcher in den unterschiedlichen Paradigmen begründet liegt. So beschreibt die Objektrelationale Abbildung eine Technik, bei welcher sich Objekte einer objektorientierten Sprache in einer relationalen Datenbank persistieren lassen. [15]

Java bietet mit der sog. Java Persistence API (JPA) [16] eine Abstraktion genau zu diesem Zweck, dessen sich Hibernate auch bedient. Mittels Annotationen lassen sich Objekte mit Attributen und Methoden – zumeist Plain Old Java Objects (POJOs) – auf Entitäten abbilden. Diese Annotation definieren, welche Tabelle auf welches Objekt und welche Spalte auf welches Attribut abgebildet wird. Es lassen sich außerdem die Relationen der Entitäten auf die Assoziationen der Objekte abbilden. Hibernate bzw. JPA unterstützt 1:1, 1:N sowie N:N Relationen. Somit wird ein vollständiges Abbild der Persistenz in der Anwendung geschaffen.

Die einzige Vorgabe bei der Definition der Objekte ist, dass ein parameterloser Konstruktor existieren muss. Hibernate greift auf die Attribute der Klasse mittels Reflection zu.

Transaktionsmanagement

In Hibernate erfolgt der Zugriff auf die Persistenz über sogenannte *Sessions*. Eine Session repräsentiert eine physische Verbindung zwischen der Persistenz und der Anwendung und bietet Methoden für alle Datenbestandsoperationen. Der Lebenszyklus einer Session ist durch den Beginn und das Ende einer logischen Transaktion begrenzt. Es ist konfigurierbar, ob Hibernate das Sessionmanagement übernimmt, oder die Anwendung selbst die Sessions öffnet und wieder schließt. So werden auch parallele Datenbankverbindung und damit auch eine Performance Verbesserung ermöglicht.

Um eine Session mittels des Sessionmanagements zu erstellen, wird sich der *SessionFactory* bedient, von welcher meist nur eine Instanz in der Applikation existiert. Diese beinhaltet die Konfiguration, die den Verbindungsaufbau und die Verbindung selbst definiert.

Eine Session ermöglicht es eine *Transaction* – Abstraktion der Implementation von JTA – zu starten und zu beenden. Wie bereits in Abschnitt 2.2.3 beschrieben, lässt sich hierbei das Transaktionsmanagement sehr gut integrieren, sodass Spring das Starten und Beenden von Transaktionen handhabt. Dies führt zu einer Simplifizierung des Codes, der zum einen lesbarer und zum anderen einfacher wird.

2.2.5 Docker

Docker ist eine Open Source Virtualisierungssoftware, die der Isolierung von Anwendungen dient und von Docker Inc. bereitgestellt wird. Die Motivation hinter der Verwendung von Docker liegt im Deployment Prozess begründet:

Der noch vor einigen Jahren vorherrschende Auslieferungsprozess war zumeist eine Installationsanleitung, welche die einzelnen Anweisungsschritte beinhaltete, die es auszuführen galt, um die Applikation zu starten. Da dies jedoch auf Dauer zu komplex wurde, wurden Anwendungen mittels virtueller Maschinen ausgeliefert. Hierbei bestand der Vorteil darin, dass keine Abhängigkeiten o.Ä. installiert, sondern lediglich die virtuelle Maschine mit dem ausgelieferten Image gestartet werden musste. Jedoch beinhaltet ein solches Image ein vollständiges Gastbetriebssystem, dessen Größe die der Anwendung um einiges überstieg und damit nicht effizient und geeignet war. Des Weiteren stellt die Dauer des Starts einer virtuellen Maschine auch ein Problem dar.

Um nun all diesen Problemen entgegenzuwirken, wurde Docker ins Leben gerufen. So werden alle Abhängigkeiten einer Anwendung in einem sog. Docker Image zusammengefasst, aus welchem wiederum Instanzen – sog. Docker Container – erzeugt werden. Diese leichtgewichtigen Images können sehr einfach ausgeliefert werden und sind darüber hinaus in der Lage, sehr schnell zu starten. [17]

Der Unterschied zwischen Docker und einer virtuellen Maschine besteht darin, dass, anstelle eines vollständigen Gastsystems, ein Docker Container sich des Betriebssystems des Hosts bedient, dem sog. Host OS. Mittels der Docker Engine wird der Zugriff auf den Kernel des Betriebssystems sichergestellt und diese bietet zudem die Möglichkeit, Container zu erstellen, zu stoppen oder zu starten. Abbildung 2.2 stellt diesen Unterschied graphisch dar. [18]

Docker Stacks

Des Weiteren bietet Docker die Möglichkeit, mehrere Images zusammenzufassen und zu kombinieren. In sog. *docker-compose* Dateien – diese Dateien werden im YAML Format gespeichert – können mehrere Docker Images konfiguriert werden. Diese konfigurierten Docker Images erzeugen dann Instanzen – also Container –, welche jeweils



Abbildung 2.2: Docker Virtualisierung verglichen mit virtuellen Maschinen [19]

Zugriff aufeinander haben. So kann ein sog. Docker Stack – bestehend aus mehreren Images – als ganze Einheit gestartet werden und auch beendet werden.

Eine Anwendung eines solchen Stacks könnte beispielsweise die Kombination aus einem Backend-Image einer Anwendung und einem dazugehörigen Datenbank-Image sein. Dies bietet den Vorteil, dass die ganze Applikation inklusive Datenbank bzw. deren Konfiguration einfach ausgeliefert werden kann und dieser Stack keine weiteren Abhängigkeiten zu anderen Applikationen – in diesem Falle eine Datenbank Applikation – auf dem Server hat. Ein weiterer Vorteil dieser Technik ist, dass die Daten der Applikation auf der Datenbank von allen anderen Applikationen und Docker Containern auf dem Server abgekapselt sind. Dies trägt zum Schutz der Daten bei. In Kombination mit Hibernate ist dann sogar die Änderung der Datenbankimplementierung ein leichtes.

DockerHub

Eine Plattform, die ebenfalls von Docker Inc. bereitgestellt wird, ist der sog. Docker-Hub [20]. Dieser stellt eine Registry für Docker Images und Repositories dar und bietet dem Nutzer die Möglichkeit selbst erstellte Images hochzuladen und somit anderen Nutzern zur Verfügung zu stellen.

Ferner bietet Docker eine Versionsverwaltung, welche es ermöglicht, verschiedene Versionen eines Images zu erstellen. So sind die verschiedenen Versionen eines Images auf dem DockerHub einsehbar.

2.2.6 Android

Google stellt für die Entwicklung von Client-Applikation auf dem Android Betriebssystem für mobile Endgeräte ein Software Development Kit (SDK) [21] zur Verfügung. Dies ermöglicht die Entwicklung von Android-Apps in den Programmiersprachen Kotlin, Java und C++.

Die Entscheidung, welche Programmiersprache zu verwenden, beschränkte sich auf Java und das auf der Java Virtual Machine (JVM) basierende Kotlin, da die bereits genannte Frameworks ebenfalls in Java geschrieben sind und so Einheitlichkeit herrscht. Die in [22] beschriebenen Argumente sprechen für die Verwendung von Kotlin bei der Entwicklung von Android Apps. Da Kotlin zwar ähnlich, aber nicht identisch zu Java ist, folgt aus dieser Entscheidung zudem ein Lernprozess und eine Weiterbildungsmaßnahme – es sind keine Kenntnisse und Erfahrungen über Kotlin vorhanden –, welche im Rahmen einer wissenschaftlichen Arbeit wie dieser ausdrücklich gefordert sind.

Kotlin

Kotlin ist eine plattformunabhängige und statisch typisierte Programmiersprache, die von JetBrains entwickelt wurde. Beim Kompilieren wird der Quellcode in Bytecode übersetzt, der auf der JVM laufen kann. Seit 2017 wurde Kotlin offiziell von Google für die Entwicklung von Android Apps unterstützt [23].

Ein Vorteil von Kotlin gegenüber Java sind sog. Coroutines. Coroutines erleichtern die asynchrone Programmierung, indem die in Java verwendeten Callbacks – meist durch anonyme Klassen oder seit Java 8 durch Lambdas – ersetzt werden. Coroutines lassen sich als leichtgewichtige Threads bezeichnen und verhalten sich ähnlich wie Jobs: Auf einem Thread können mehrere Coroutines existieren und unabhängig agieren. Der Vorteil ist hierbei, dass diese deutlich performanter bzgl. Start und Erhalt

sind. Zudem können Coroutines auch den aktuellen Thread zur Laufzeit mehrfach wechseln. Dies ermöglicht es, beispielsweise asynchron Daten von einer API abzurufen, und diese auf dem aktuellen UI Thread anzuzeigen und zwar innerhalb einer Coroutine, ohne den UI Thread zu blockieren.

Zu diesem Zweck hat Kotlin drei verschiedene Kontexte eingeführt unter denen eine Coroutine laufen kann:

1. **Main:** Unter dem *Main* Kontext laufen alle Operationen, die mit dem UI interagieren.
2. **IO:** Mithilfe des *IO* Kontextes werden Input/Output Operationen, wie beispielsweise der Zugriff auf eine Datei oder eine Web Ressource, ausgeführt.
3. **Default:** Coroutinen mit diesem Kontext stellen schwere Berechnungen an, und blockieren damit nicht den UI Thread.

Das Schlüsselwort *suspend* in Methodenköpfen definiert, dass diese Methode in der Lage ist, asynchron zu arbeiten. Diese Methoden können entweder von weiteren *suspend* Methoden aufgerufen werden, oder aber innerhalb einer Coroutine.

2.3 Build Management Tools

2.3.1 Maven

Maven ist ein von der Apache Software Foundation entwickeltes Build-Management-Tool. Es ist Open Source und basiert auf Java. Das Ziel von Maven ist zweiteilig: Auf der einen Seite soll eine Applikation mittels Maven automatisiert gebaut werden und auf der anderen Seite dient Maven der Verwaltung von Abhängigkeiten.

In einer XML basierten Konfigurationsdatei werden – gemäß des Paradigma *Convention over Configuration* – ausschließlich Ausnahmen definiert, wie Abhängigkeiten auf externe Module und Komponenten, benötigte Plugins oder die Build Reihenfolge. Alle für das Projekt benötigten Bibliotheken und Plugins werden von Maven dynamisch von einem oder mehreren Repositories heruntergeladen und in einem lokalen Cache zwischengespeichert. [24]

Der Build Prozess einer Maven Applikation besteht aus folgenden Schritten:

1. *archetype*: Alle Abhängigkeiten des Projektes werden aufgelöst und bei Bedarf heruntergeladen.
2. *validate*: Prüfung, ob die Projektstruktur valide und vollständig ist.
3. *compile*: Das Projekt wird in dieser Phase kompiliert.
4. *test*: Alle sich in dem Projekt befindlichen Tests (siehe Unterabschnitt 2.4.1) – diese werden in einem von Maven definierten Verzeichnis gespeichert – werden ausgeführt und evaluiert.
5. *package*: Das Kompilat wird mit anderen nicht kompilierbaren Dateien zum Zwecke der Weitervergabe verpackt, meist in Form einer Jar-Datei.
6. *integration-test*: Alle sich in dem Projekt befindlichen Integration Tests (siehe Abschnitt 2.4.1) werden ausgeführt.
7. *verify*: In dieser Phase wird eine Gültigkeitsprüfung des Software-Pakets durchgeführt.
8. *install*: Das Projekt wird im lokalen Zwischenspeicher installiert, sodass die Verwendung dieses Projektes durch andere ermöglicht wird.
9. *deploy*: Das Projekt wird im zentralen Repository von Maven installiert, sodass es zum Herunterladen zur Verfügung steht.

2.3.2 Gradle

Gradle ist ebenso, wie Maven, ein Build-Management-Tool. Statt XML wie bei Maven verwendet Gradle eine Groovy basierende domänenspezifische Sprache. Außerdem sind Gradle Skripte direkt ausführbarer Code und keine Projektbeschreibung oder -definition. Die von Gradle definierten Build Prozesse unterscheiden sich jedoch nur minimal von den von Maven verwendeten.

Mitte 2013 hat Google Gradle als Build-Management-Tool für Entwicklung von Android Applikationen festgesetzt. So wurde das Android Gradle Plugin eingeführt,

welches es ermöglicht, die Applikation außerhalb der für die Android Entwicklung bereitgestellten IDE *Android Studio* zu bauen. [25]

2.4 Qualitätssichernde Maßnahmen

2.4.1 Tests

Eine der am häufigsten verwendeten Maßnahmen, um die Qualität von Code zu bewerten, sind automatisierte Tests. Tests dienen dem Zweck, dass Tests automatisch ausgeführt werden, sobald eine Änderung im Versionskontrollsystem vorliegt und somit auf eventuelle, durch die Änderung entstandene Fehler hinweisen. Dies ermöglicht es, noch vor der Veröffentlichung einer Applikation diese durch Tests aufgedeckten Fehler zu beheben und spart unter Umständen auch Kosten, da diese Fehler sonst im Produktivbetrieb erst aufgefallen wären. [26] Es existieren verschiedene Ansätze Tests zu implementieren. Im Folgenden werden die für diese Arbeit relevanten Ansätze beschrieben.

Unit Tests

Unit Tests beschreiben automatisierte Tests, die nur kleinste Einheiten der Software bzw. der Logik testen. Dabei werden alle Abhängigkeiten dieser kleinsten Einheit gemockt – Mocking ist die Simulation eines bestimmten Zustands einer Software –, sodass ausschließlich die Funktion der kleinsten Einheit getestet wird. Diese kleinste Einheit sind zumeist Methoden verschiedener Klassen. Die Intention von Unit Tests ist also das Prüfen der funktionalen Einzelteile einer Software auf Korrektheit.

Ein in Java häufig für die Implementierung von Unit Tests eingesetztes Framework ist JUnit. JUnit ist Open Source und wurde u.A. von Kent Beck und Erich Gamma entwickelt. JUnit bietet einige nützliche Funktionalitäten: So können pro Test Klasse beispielsweise Methoden deklariert werden, die jeweils vor und nach einem Test oder aber auch vor und nach allen sich in dieser Klasse befindlichen Tests ausgeführt werden. Außerdem lassen sich Tests seit JUnit 5.0 Tests taggen, um beispielsweise nur Tests eines vorher definierten Anwendungsfalles auszuführen. [27]

Spring unterstützt die Implementierung von JUnit Tests durch die Bereitstellung eines Testframeworks, mit dessen Hilfe der Spring Server gemockt werden kann.

Integration Tests

Da bei Unit Tests nur einzelne Einheiten getestet werden und nicht jedoch deren Integration, werden Tests, die mehrere Komponenten einer Software testen, Integration Tests genannt. Ein Beispiel für einen solchen Test ist der Test einer Benutzeroberfläche in Verbindung mit einem dahinterstehenden Server. Dabei wird der Server und die dazugehörige Datenbank in einer Testumgebung gestartet, sodass der Produktivbetrieb dabei nicht gestört wird.

Dies setzt natürlich voraus, dass es ein Framework zum Testen von Benutzeroberflächen gibt. Das von Google bereitgestellte Test Framework Espresso stellt genau diese Funktionalität zum Testen von UIs bereit.

Testabdeckung

Um nun die Qualität von Software mittels Tests zu messen, wurde die sog. Testabdeckung (engl. Code Coverage) eingeführt. Diese Metrik beschreibt den Anteil des Quellcodes, der während eines Testlaufes durchlaufen wird. Die Testabdeckung wird prozentual gemessen. Wenn die Testabdeckung einer Applikation hoch ist, bedeutet dies, dass der Anteil des durchlaufenen Codes hoch ist und somit die Wahrscheinlichkeit für einen Fehler geringer ist und umgekehrt. Eine hohe Abdeckung ist jedoch kein Garant für fehlerfreien Quellcode, da auch Tests fehlerhaft sein können, oder nicht unbedingt den Fehler hervorrufen. [28]

Man unterscheidet bei der Testabdeckung unter Anderem zwischen den in [29] beschriebenen Metriken:

1. **Class Coverage:** Anteil der durchlaufenen Klassen innerhalb einer Applikation
2. **Function Coverage:** Anteil der durchlaufenen Funktionen, Methoden oder Subroutinen innerhalb der Applikation

3. **Statement Coverage:** Anteil der durchlaufenen Anweisungen innerhalb der Applikation
4. **Branch Coverage:** Anteil der durchlaufenen Zweige von jeder Kontrollstruktur (*if*- und *case*-Anweisungen sowie Schleifen) innerhalb einer Applikation.

Die oben beschriebenen Metriken reichen für diese Arbeit aus, um eine Aussage über die Qualität des Codes bzgl. der Testabdeckung zu treffen.

2.4.2 Code Reviews

Um sicher zu stellen, dass der Code nahezu fehlerfrei ist, werden häufig sog. Code Reviews durchgeführt. Hierbei setzen sich der Entwickler, der diesen Code geschrieben hat sowie ein oder mehrere weitere Entwickler zusammen und lesen den Code gemeinsam. Somit werden zum einen Fehler oder Performance-Verbesserungen erkannt, die der Entwickler nicht im Blick hatte und zum anderen wird der Code auf Verständlichkeit geprüft. Da Code häufig mehrfach gelesen und abgeändert wird, ist dies ein essentieller Bestandteil der qualitätssichernden Maßnahmen. Dies erspart Entwicklern häufig Zeit, da verständlicher Code viel leichter zu lesen ist.

GitHub bietet genau zu diesem Zweck mittels sog. Pull Request – eine Vorgehensweise in der Versionsverwaltung, den Code aus einem Feature Branch und dem Hauptbranch zusammenzuführen – anderen Entwicklern die Möglichkeit zu geben, den Code zu begutachten und zu verbessern.

2.4.3 SonarQube

SonarQube ist eine in Java geschriebene Plattform, welche Quellcode hinsichtlich technischer Qualität analysiert und bewertet. Die Ergebnisse dieser Analyse werden über eine Weboberfläche dargestellt. [30]

SonarQube analysiert den Code hinsichtlich folgender Qualitätsmerkmale:

- **Code Duplikationen:** Duplikationen im Code gilt es innerhalb eines Projektes (und zumeist auch außerhalb) zu vermeiden. Sobald es Duplikate im Code gibt – sei es durch die Unwissenheit oder aber die Absicht eines Entwicklers –, steigt

der Aufwand für die Wartung der Duplikationen, da Änderungen an mehreren Stellen gepflegt werden müssen.

- **Testabdeckung:** siehe Abschnitt 2.4.1
- **Komplexität:** Mit der Komplexität von Code wird im Bereich der qualitätssichernden Maßnahmen die Anzahl der Verschachtelungen – beispielsweise von Kontrollstrukturen *if*, *for* und *while* – bezeichnet. Da Code mit vielen Verschachtelungsebenen schwer zu lesen und nachzuvollziehen ist, sollte die Komplexität möglichst gering sein. Um komplexe Methoden verständlicher zu machen, werden häufig Teile in weitere Methoden ausgelagert, sodass ferner auch die Wiederverwendbarkeit und die Kapselung gefördert wird.
- **Potentielle Fehler:** SonarQube erkennt potentielle Fehler, die nicht vom Compiler erkannt werden wie beispielsweise den Fall, dass eine Schleife nicht terminiert, sodass diese unendlich lange läuft.
- **Code Richtlinien:** SonarQube prüft den Code auf die von SonarQube definierten Richtlinien. Diese Richtlinien sind häufig Performance Verbesserungen, Schließen von Sicherheitslücken oder Verbessern der Lesbarkeit.

Alle definierten Regeln und Richtlinien können durch den Anwender angepasst oder deaktiviert werden. Außerdem lässt sich unter anderem die Anzahl der Regelverstöße definieren, um festzulegen, wann der Build durch SonarQube fehlgeschlagen ist und wann nicht. Gleiches gilt für die Testabdeckung.

2.4.4 Continuous Integration

Continuous Integration (CI) ist ein Automatisierungsprozess in der Entwicklung. Wenn Änderungen zum Versionskontrollsystem hinzugefügt und veröffentlicht werden, startet automatisiert auf einem CI Server ein bestimmter Prozess. Dieser Prozess beinhaltet häufig Kompilation und Testen der Applikation. Dies bietet den Vorteil, dass Fehler noch schneller entdeckt und behoben werden können. Außerdem ist eine Effizienzsteigerung in der Entwicklungsphase zu erkennen [31].

Ein Tool, das genau dies umsetzt, ist Jenkins. Jenkins ist eine Open Source Software, welche als Nachfolger der Software Hudson gilt. Jenkins' Multibranch Pipeline bietet

die Möglichkeit, die einzelnen Schritte dieses Prozesses auf den unterschiedlichen Branches des Versionskontrollsystems auszuführen. Dies ermöglicht es z. B., dass die Zusammenführung dieses Branches mit dem Hauptbranch fast fehlerfrei erfolgt. Falls ein Prozess fehlschlägt, gilt die ganze Ausführung der Pipeline als fehlgeschlagen. [32]



Abbildung 2.3: Darstellung der für das Projekt *Travlyn* erstellten Jenkins Pipeline

Abbildung 2.3 zeigt die für diese Arbeit erstellte Pipeline. Die Abbildung besteht aus drei Schritten: Zunächst werden Frontend und Backend kompiliert. So können Syntax- und Kompilierfehler erkannt werden. Im darauffolgenden Schritt werden die sich sowohl im Frontend als auch im Backend befindlichen Tests ausgeführt. Anschließend die Analyse der Code Qualität mittels SonarQube (siehe Unterabschnitt 2.4.3) durchgeführt.

2.4.5 Continuous Delivery

Continuous Delivery (CD) bezeichnet die kontinuierliche und im Besonderen automatisierte Auslieferung einer Applikation. Ziel ist, dass, nachdem ein Build die Pipeline erfolgreich durchlaufen hat, der Build Server – z.B. Jenkins – Artefakte der Applikation speichert und zur Verfügung stellt. Diese Artefakte sind häufig das Kompilat der Applikation, die sich Kunden und Nutzer herunterladen können. Artefakte kön-

nen aber ebenso gut Docker Images sein, mit deren Hilfe der Auslieferungsprozess simplifiziert wird. [33]

3 Anforderungen

(Joshua)

In diesem Kapitel sollen die Anforderungen an die zu erstellende Applikation, sowie der zu erreichende Scope beschrieben werden um eine abschließende Bewertung durchführen zu können, die das Erreichte mit den Zielen vergleicht.

Zuerst wird die Begrifflichkeit Anforderung geklärt. In der Softwareentwicklung gibt es einige verschiedene Definitionen von „Anforderungen“. In dieser Arbeit soll der Definition gefolgt werden, welche von Helmut Balzert in seinem Buch zu den Basiskonzepten und des Requirements Engineering erarbeitet wurden [34]. In seinem Werk werden Anforderungen wie folgt definiert.

Anforderungen: „Anforderungen (requirements) legen fest, was man von einem Softwaresystem als Eigenschaften erwartet“ [34] (Seite 455). Mit der Annahme, dass „man“ alle Stakeholder (Personen, die ein Interesse an der Entwicklung und/oder der erstellten Software haben) beinhaltet. [34]

Des Weiteren sollten vor der Festlegung der Anforderungen „Visionen und Ziele“ und die Rahmenbedingungen, in denen die Software existieren sollen, festgelegt werden. Erst danach sollten Eigenschaften, welche in funktionale und nicht-funktionale Eigenschaften unterteilt werden können, definiert werden. In den folgenden Unterkapiteln wird nach diesem Prinzip vorgegangen, um eine konsistente und sinnvolle Anforderungslage zu schaffen.

3.1 Visionen und Ziele

An erster Stelle der Anforderungen steht eine Vision für das zu erstellende Produkt. In diesem Fall wurden bereits einige wichtige Punkte in der Einleitung der Arbeit zusammen gefasst, die zu einer kurzen und prägnanten Vision führen:

„Durch *Travlyn* sollen Nutzer in der Lage sein, ihre Städtereisen ohne das Mitführen von papierbasierten Reiseführern oder die Nutzung von multiplen mobilen Diensten zu bewältigen, ohne dabei einen Informationsverlust oder eine Beeinträchtigung des Reisespaßes hinnehmen zu müssen.“

Anhand dieser Version, die beschreibt, was erreicht werden soll aber nicht wie, werden konkrete Ziele abgeleitet. Es wurde entschieden, dass diese Ziele dem „SMART“ Prinzip folgen sollten.

SMART Ziele: Die Art und Weise messbare Ziele zu setzen kann einer festgelegten Struktur folgen. In diesem Fall soll die Struktur, welche Peter Drucker in seinem Buch „The Practice of Management“ (1954) erarbeitet hat, genutzt werden. Allerdings hat Drucker nie eine genaue Erklärung zu der Bedeutung des Akronyms „SMART“ abgegeben, deswegen wird folgende allgemein akzeptierte Variante gewählt [35]:

- Specific (Spezifisch)
- Measurable (Messbar)
- Attainable (Attraktiv)
- Realistic (Realistisch)
- Timely (Terminiert)

Folgend diesem Prinzip sind folgende Ziele entstanden:

- Ein Nutzer soll sich vor einer Reise über *Travlyn* folgende Informationen zu seiner Zielstadt einholen können: Name, Lage, Beschreibung und ein Bild, welches einen ersten Eindruck der Stadt vermittelt.
- Die vermittelten Informationen und durchgeführten Schritte zum Entwurf einer Reise werden anhand der vom Nutzer zur Verfügung gestellten Informationen personalisiert. Außerdem kann jeder Nutzer seine persönliche Reise designen und muss sich damit nicht auf festgelegte Routen beschränken, was zu einer hohen Individualisierung führt.
- Vor und während der Reise wird *Travlyn* den Nutzer entlang einer vorher festgelegten Route durch die Stadt leiten und Informationen, wie Beschreibungen, Bil-

der und Kosten anzeigen. Die Route beinhaltet Stops an interessanten Orten der Stadt, wie Sehenswürdigkeiten und spannenden Aktivitäten.

- Während der Führung durch *Travlyn* werden einzelne Texte per Machine Learning zu einer zusammenhängenden Führung zusammengefasst und von Android vorgelesen um den Eindruck eines realen Stadtführers zu schaffen.
- Der Nutzer kann Reisepläne teilen und Pläne von anderen Nutzern einsehen können, um sich inspirieren zu lassen und seinen eigenen Plan an den existierenden Plänen orientieren zu können.

3.2 Rahmenbedingungen

Laut Balzert stellen Rahmenbedingungen „organisatorische und/oder technische Restriktionen für das Softwaresystem und/oder den Entwicklungsprozess“ da [34] (Seite 459).

3.2.1 Organisatorisches

Für *Travlyn* liegen folgende organisatorische Rahmenbedingungen vor: Der Anwendungsbereich der Software liegt im privaten Umfeld, genauer gesagt im Bereich des privaten Reisens. Die Zielgruppe sind alle Personen, die für relativ kurze Zeiträume in größere Städte reisen und diese mit ihren Sehenswürdigkeiten entdecken wollen und sich zu diesen weiter informieren wollen. Damit liegt eine mobile Benutzung unter ständiger Beobachtung des Nutzers vor. Während der Reise/der durch *Travlyn* geführten Tour wird die Anwendung ohne Unterbrechung laufen.

3.2.2 Technisches

Die technischen Anforderungen werden an dieser Stelle in zwei Abschnitte aufgeteilt, da sich die Rahmenbedingungen für den Server und den Client stark unterscheiden.

Für den Server wird festgelegt, dass er in einem Docker-Container[18] läuft, welcher unabhängig von dem darunterliegenden Betriebssystem ist. Peripherie wird es an diesem Rechner keine geben, da er nur per Remote Zugriff von außen gesteuert werden wird. Die wichtigste Rahmenbedingung ist, dass die Hardware, auf welcher der Server läuft ständig mit dem Internet verbunden sein muss, um eine dauerhafte Verfügbarkeit zu gewährleisten

Für den Client wird festgelegt, dass er auf einem mobilen Smartphone, welches im Akkubetrieb operiert, läuft. Auf dem Gerät muss Android als Betriebssystem laufen und die Version soll 8.0 (Oreo) nicht unterschreiten. Das Gerät stellt eine klassische mobile Peripherie zur Verfügung, welche z.B. eine virtuelle Tastatur, einen GPS-Sensor und einen Lautsprecher/Kopfhöreranschluss beinhaltet. Auch für den Client muss eine konstante Internetverbindung existieren, um sicher zu stellen, dass der Server ständig erreicht werden kann und die entsprechenden Informationen abgefragt werden können.

3.3 Geforderte Eigenschaften

Nachdem die Ziele und die Rahmenbedingungen definiert sind, können die erwarteten Eigenschaften erarbeitet werden. Bei jeder Eigenschaft sollte geprüft werden, ob diese im Sinne eines der gesteckten Ziele sind und zum Erreichen der Vision beitragen und ob diese im Sinne der Rahmenbedingungen erreichbar und realistisch ist. Wenn eine dieser beiden Prüfungen nicht positiv erfüllt wird, sollte über eine Redefinition der Eigenschaft nachgedacht werden, denn in der aktuellen Form ist sie nicht zielführend und kann mit dem vorliegendem Rahmen ggf. nicht umgesetzt werden.

Wie im vorherigen Verlauf beschreiben, werden Eigenschaften häufig in funktional und nicht-funktional aufgeteilt [34]. Im Folgenden wird dieser Trennung gefolgt.

3.3.1 Funktionale Eigenschaften

Unter funktionalen Eigenschaften wird alles spezifiziert, was ein System tun und explizit nicht tun/können soll. Laut Balzert können diese Eigenschaften in statische,

dynamische und logische Eigenschaften aufgeteilt werden[34]. Wir haben uns explizit gegen eine solche Gliederung entschieden, um dieses Kapitels nicht zu sprengen.

Zur Visualisierung der erwarteten funktionalen Eigenschaften wurde ein Use Case Diagramm erstellt.

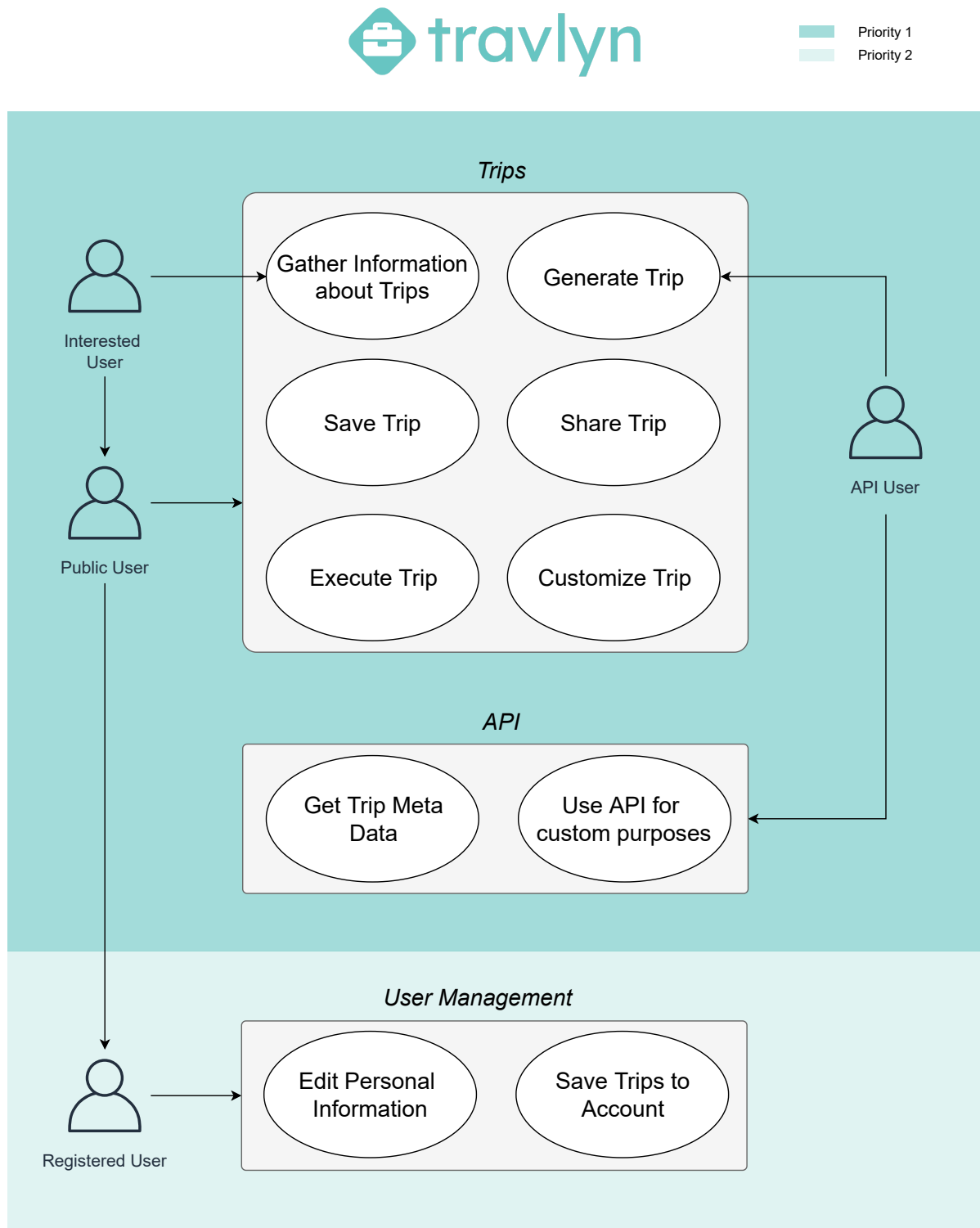


Abbildung 3.1: Darstellung aller geplanten Use Cases mit den assoziierten Nutzerprofilen.

In Abbildung 3.1 wird der Begriff *Trip* geprägt, welcher sich durch die gesamte Softwareentwicklung und Arbeit ziehen wird.

Trip: Ein Trip ist eine durch den Server erstellte Abfolge von sehenswerten Punkten in einer Stadt. Damit ist jeder Trip eindeutig einer Stadt zugeordnet. Neben den Punkten und der Route, um sich entlang der Punkte zu bewegen, werden weitere Metainformationen zu den Punkten geliefert, die einen Besuch noch interessanter machen.

Abbildung 3.1 zeigt neben den Use Cases auch die Nutzerprofile, denen die Use Cases zugeordnet sind. Für *Travlyn* sind vier Nutzerprofile geplant. Der *interessierte Nutzer* benutzt die App eher zufällig und ohne die Intention sie aktiv für eine Reise einzusetzen. Sein Ziel ist es sich über die Funktionalität und bereits bestehende Trips zu informieren. Sollte der Nutzer in dieser Phase ansprechende Informationen finden und sich entscheiden *Travlyn* für die nächste Reise einzusetzen entwickelt er sich zum *Öffentlichen Nutzer*, welcher ohne sich anzumelden oder zu registrieren alle Aktionen auf Trips ausführen kann und die volle Funktionalität zur Durchführung von Trips nutzt. Darauf aufbauend ist eine freiwillige Registrierung möglich, um persönliche Daten zu hinterlegen und alle eigenen Trips in an einem persönlichen Platz zu speichern. Alle Nutzer die sich dafür entscheiden liegen im Profil *registrierter Nutzer*.

Da die RESTful API, die im Zuge dieser Softwareentwicklung erstellt wird, öffentlich angeboten wird, ist es möglich diese API in fremden Applikationen zu eigenen Zwecken zu nutzen. Aus diesem Anwendungsfall bildet sich das Profil *API Nutzer*. Es ergeben sich aus Abbildung 3.1 z.B. Funktionen, wie die Abfrage von öffentlichen Trips, um diese in einer eigenen Applikation an zu bieten und als Inspiration zu Nutzen. Ein vorstellbares Szenario wäre die Nutzung auf der Website einer Stadt zu Werbezwecken. Eine weitere benutzbare Funktion wäre die Ausnutzung der in *Travlyn* vorliegenden Informationssammlung für eigene Zwecke wie Machine Learning o. Ä..

Alle in Abbildung 3.1 dargestellten Use Cases wurden auf ihre Kompatibilität zu Zielen und Rahmenbedingungen geprüft und für passend befunden. Zusätzlich wurden sie bereits grob priorisiert. Es ist wichtig zu erwähnen, dass viele der dargestellten Use Cases im Laufe der Entwicklung zu kleinen Einheiten aufteilt werden, die hier im Sinne der Übersichtlichkeit nicht dargestellt sind.

Abschließend ist zu erwähnen, dass die funktionalen Anforderungen der in der Einführung geschilderten Funktionalität folgen soll.

3.3.2 Nicht-Funktionale Eigenschaften

Neben den geschilderten funktionalen Eigenschaften gibt es weitere Anforderungen, welche sich nicht direkt auf die Funktionalität und den Funktionalitätsumfang auswirken. Diese Eigenschaften werden auch Quality of Service (QoS) genannt und beinhalten häufig Eigenschaften wie Genauigkeit, Verfügbarkeit und Konsumierbarkeit. Es ist zu erwähnen, dass einige dieser Anforderungen miteinander in Konflikt stehen und ein geeigneter Mittelweg gefunden werden muss bzw. bestimmte Kompromisse eingegangen werden müssen, z.B. schränkt die Eigenschaft der möglichst hohen Sicherheit meist die Eigenschaft der Benutzbarkeit oder der Speichereffizienz ein [34].

Für diese Softwareentwicklung soll für die nicht-funktionalen Eigenschaften als Orientierung die internationale ISO/IEC 25010 Norm gelten. Diese Norm setzt einen Standard für die Qualitätskriterien und -bewertung von Software und ist in drei Bereiche aufgeteilt [36] [37]:

- **Quality In Use Model:** Dieser Teil beschreibt alle Merkmale, welche die Interaktion zwischen Mensch und System beschreiben, wie z.B. Effektivität und Freiheit von Risiken.
- **Product Quality Model:** Der zweite Teil der Norm beschreibt acht Charakteristika, welche ein Softwareprodukt erfüllen sollte, u.a. Wartbarkeit, Sicherheit und Benutzbarkeit.
- **Data Quality Model:** Der dritte und letzte Teil beschreibt Ansprüche, welche an das Datenmodell gestellt werden sollten um eine möglichst konsistente Benutzung der Daten zu ermöglichen.

Diese Norm ist sehr umfangreich und für ein Studienprojekt nicht vollständig umsetzbar, ohne den Zeit- und Aufwandsrahmen zu sprengen. Deshalb wurde einige wichtige Punkte ausgewählt auf die ein besonderes Augenmerk gelegt werden soll. Zusätzlich wurden einige themenspezifische Anforderungen hinzugefügt. Daraus ergibt sich folgende Liste:

- **Benutzbarkeit:** Durch die intensive mobile Benutzung der *Travlyn* App sollte diese für alle gängigen Gerätetypen eine gute Nutzungserfahrung bieten, die Nutzer überzeugen kann und die Reise angenehm verlaufen lässt. Explizit soll an dieser Stelle die Performance der Software genannt werden, welche bei anderen Applikationen häufig für eine schlechte Benutzbarkeit sorgt. Außerdem sollen ansprechende User Interfaces gestaltet werden, die den Nutzer zur Verwendung von *Travlyn* einladen und z.B. über Spracheinstellungen für möglichst viele Personen anpassbar sind.
- **Sicherheit:** Der Schutz persönlicher Daten wird immer wichtiger und soll auch bei *Travlyn* nicht vernachlässigt werden. Sowohl die API als auch die Client Applikation sollen nicht anfällig für gängige Angriffe, wie Injection oder Denial of Service Angriffe (bzw. Distributed denial of service) [38] sein und sicherstellen, dass persönliche Daten nur an authentifizierte Nutzer ausgeliefert/angezeigt werden.
- **Wartbarkeit:** Die Software sollte über die bereits beschriebenen qualitätssichernden Maßnahmen und durch eine entsprechende Architektur gut wartbar und erweiterbar sein. Um dieses Ziel zu erreichen soll auf eine möglichst modularisierte Architektur geachtet werden. An dieser Stelle ist der Verzicht auf *code ownership* besonders hervorzuheben, d.h. alle Teilnehmer des Entwicklungsteams haben Einblicke in alle Teile des Codes und können im Notfall Korrekturen und Erweiterungen vornehmen. Es ist nicht erwünscht, dass einzelne Code Teile nur von einer Person geschrieben, gepflegt und gewartet werden. Außerdem soll durch den Server eine ausführliche Dokumentation der Benutzung angefertigt werden, um auftretende Fehler identifizieren und beheben zu können. Hierzu sollen alle Aktionen und ihr Ergebnis in geeigneter Weise persistiert werden.
- **Datenquellen:** Da diese Software im Rahmen eines Studienprojekts mit sehr begrenzten Ressourcen erstellt wird, können keine kostenpflichtige Services eingebunden werden. Außerdem sollen schwierige Lizenzfragen vermieden werden, indem komplett auf Opensource Dienste gesetzt wird, bei denen die Nutzung der Daten vollständig erlaubt und frei ist. In den folgenden Kapiteln wird genauer auf dieses Thema eingegangen und die verschiedenen Alternativen um Daten zu beschaffen genauer vorgestellt.

- **Zuverlässigkeit:** Der Nutzer soll sich auf *Travlyn* verlassen können. Es ist wichtig, dass vor allem während einer Reise alle Funktionen zur Verfügung stehen und verhindern, dass der Nutzer alleine gelassen bzw. gezwungen wird auf andere Diensten zurückgreifen zu müssen. Außerdem sollten die Ausfallzeit für die API und den Client so gering wie möglich sein.

4 Datengrundlage

(Joshua)

Die Datengrundlage ist für einen Reiseführer sehr wichtig, da der gesamte Sinn eines Reiseführers darauf basiert, Informationen aufzubereiten und an den Leser/Nutzer weiter zu geben. Aus diesem Grund wurden für diese Arbeit mehrere Datenquellen zu unterschiedlichen Themen herausgesucht und verglichen. Im Folgenden sollen diese Alternativen und die angestellten Überlegungen sowie die endgültige Entscheidung, welche Daten für *Travlyn* verwendet werden sollen, aufgezeigt.

4.1 Points of Interest

Travlyn soll laut Spezifikation Trips erstellen können, welche eine Abfolge von interessanten und sehenswerten Punkten in einer Stadt ist. Diese Point of interest (POI) sollen über ein Application Programming Interface (API) in die App integriert werden. Folgende Anforderungen sind an die Informationen und die API gestellt:

- Es soll eine möglichst vollständige API gewählt werden, um zu viele Abhängigkeiten zu verhindern. Allerdings wird dies für den Informationsbedarf von *Travlyn* kaum möglich sein, sodass wahrscheinlich eine geschickte Kombination gewählt werden muss, welche *Travlyn* einzigartig macht.
- Die abgefragten Daten sollten für einen kommerziellen Nutzen zugelassen sein, damit während der Entwicklung keine schwierigen Lizenzfragen auftreten können und ggf. höhere Datenvolumen durch Nachfragen erreicht werden können. Dies ist das wichtigste Entscheidungskriterium.
- Da diese Arbeit ein Studienprojekt ist, für welches sehr begrenzte Ressourcen zur Verfügung stehen sollte die API kostenfrei benutzbar sein.
- Die API sollte Daten für möglichst viele Städte/Orte zur Verfügung stellen, damit *Travlyn* möglichst überall eingesetzt werden kann.

- Zu den einzelnen POIs sollten neben dem Namen und der Position weitere Daten wie Beschreibungen, Öffnungszeiten und ggf. Bilder bereitgestellt werden.

4.1.1 Google Places API

Google ist einer der größten Anbieter von ortsbasierten Services/Diensten und stellt eine API für POIs zu Verfügung [39]. Diese API hat sehr weit gefächerte Funktionen, die von einer einfachen Suche über ausführliche Details zu interessanten Orten bis hin zu „user check-in“ an einzeln Orten reichen. Diese große Funktionalität wäre für *Travlyn* sehr wertvoll. Allerdings sind die Google APIs nicht frei zugänglich und die Anzahl der Requests ist u.U. stark eingeschränkt [40]. Außerdem ist die Nutzung der erhaltenen Daten nur in Verbindung mit anderen von Google bereitgestellten Services erlaubt [41], somit wäre die ganze *Travlyn* Applikation an Google gebunden.

4.1.2 OpenrouteService

Openroute service [42] wird vom Heidelberger Institut für Geoinformationstechnik angeboten. Es handelt sich um eine Crowd Sourced API, d.h. sie wird durch Benutzer über OpenStreetMap (OSM) [43] gespeist und ist damit frei zugänglich. Durch die Nutzung von OSM ergibt sich der weitere Vorteil, dass die API für Orte weltweit nutzbar ist. Leider sind die gelieferten Informationen nicht sehr umfangreich und beinhalten häufig keine genauere Beschreibung und keine Bewertung o.Ä.. Weiterhin sind Crowd Sourced Informationen meist nicht offiziell verifiziert und könnten u.U. falsch sein. Die Beschränkungen für diese API sind relativ gering (500 POIs requests pro Tag), allerdings können diese auf Nachfrage erhöht werden (z.B. für Bildungszwecke).

Crowdsourcing: Beim Crowdsourcing wird das Wissen, die Kreativität oder die Arbeitskraft der Masse ausgenutzt. Jeder leistet einen kleinen Teil und zusammen ergibt sich ein großes Ganzes. Typische Beispiele sind z.B. Wikipedia oder die Klassifikation von Daten zum Machine Learning. Allerdings können diese Daten von jedem bewusst oder unbewusst verfälscht werden und sie sind sehr schwer zu verifizieren [44].

4.1.3 Foursquare

Die Firma Foursquare bietet ebenfalls eine API an [45], über die Informationen zu interessanten Orten gelesen werden können. Die API ist weltweit einsetzbar und liefert sehr viele Informationen zu einzelnen Orten, wie Ratings, kurze Beschreibungen oder Adressen von denen Bilder in der gewünschten Auflösung abgefragt werden können. Die Anzahl der möglichen Requests kann durch die Registrierung einer Kreditkarte (trotz der kostenlosen Nutzung) auf ca. 100.000 pro Tag gesteigert werden. Allerdings können die kostenfreien Varianten dieser API nicht für kommerzielle Zwecke genutzt werden und die abgefragten Daten dürfen nicht länger als 24 Stunden persistiert werden.

4.1.4 Evaluation der Alternativen

Für das Projekt wurde aus den obigen Alternativen gewählt. Zu diesem Zweck wurde die in Tabelle 4.1 dargestellte Entscheidungsmatrix aufgestellt, welche die Eigenschaften der einzelnen APIs vergleicht und die Gewichtung der Eigenschaften darstellt.

Anhand der Entscheidungsmatrix fiel die Entscheidung auf den OpenRoute service, da die beiden Eigenschaften, welche am höchsten Gewichtet sind, nämlich Kommerzieller Nutzen und Lizenzbedingungen von dieser API am besten erfüllt werden: Für diese beiden Kriterien steht Google als sehr teuer Dienst dar, welcher zwar für kommerziellen Nutzen zugelassen ist, aber in einem Studienprojekt praktisch nicht bezahlbar ist. Außerdem sind die Lizenzbedingungen so einschränkend, dass dies ein viel zu hohes Risiko birgt. Foursquare ist im Bereich Lizenzfragen zwar besser geeignet, allerdings ist auch dieser Dienst nur bei Zahlung eines hohen Betrags und Nutzung des Enterprise Models für kommerziellen Nutzen zugelassen. So sind diese beiden Services bereits ausgeschieden und es bleibt nur noch der OpenRoute service, welcher sowohl für kommerziellen Nutzen zugelassen ist und gleichzeitig kostenfrei bleibt. Leider muss dafür ein Nachteil im Bereich „Informationsfülle“ hingenommen werden, da die abgefragten Informationen bei weitem nicht so ausführlich sind wie bei den anderen beiden Diensten.

Wie aber in den Anforderungen für die Datenquelle bereits geschildert ist es nötig eine geschickte Kombination von Datenquellen zu wählen. Im Folgenden werden weitere Datenquellen vorgestellt mit welchen der durch diese Entscheidung entstandene Nachteil ausgeglichen werden kann.

	Google Places API	Foursquare	Openroute service
Kosten Gewichtung: Wichtig	Sehr teuer, siehe [39]	kostenfrei für personal use, für Abo incl. kommerzieller Nutzung min \$599	kostenfrei
Kommerzieller Nutzen erlaubt? Gewichtung: KO-Kriterium	Ja	Nein, im kostenfreien Account	Ja
Informationsfülle Gewichtung: Mittelmäßig wichtig	Sehr hoch	Hoch	Mäßig
Crowdsourcing Gewichtung: Weniger wichtig	Nein	Nein	Ja
Lizenzbedingungen Gewichtung: Essenziell wichtig	Sehr restriktiv, siehe [41]	Nennung der Datenherkunft ist Pflicht	Praktisch keine

Tabelle 4.1: Gegenüberstellung der vorliegenden Alternativen zur Abfrage der POIs

4.2 Weiterführende Informationen zu POIs und Städten

Um die eingeschränkte API Openroute service auszugleichen und dem Nutzer weitere Informationen zu seinem Reiseziel und zu besuchenden Orten zu bieten müssen weitere APIs angefragt werden.

Die wahrscheinlich bekannteste und ausführlichste Datenquelle ist Wikipedia. Dies ist eine Crowd Sourced Enzyklopädie die von allen Nutzern gespeist werden kann. Aus diesem Grund ist die Nutzung direkt im Internet aber auch per API Zugriff kostenlos und frei nutzbar. Allerdings ist zu beachten, dass die enthaltenen Informationen durch jeden verändert und ggf. gefälscht werden können und Wikipedia deshalb keine sichere Quelle für wissenschaftliche Arbeiten o.Ä. darstellt. Für den in dieser Arbeit vorliegenden Use Case wurde entschieden, dass dieses Risiko annehmbar ist und der Vorteil der sehr großen Wissensbasis das Risiko überwiegen.

Für den Zugriff auf Wikipedia gibt es unterschiedliche Möglichkeiten, im Folgenden werden zwei APIs beschrieben, die ausprobiert worden sind:

- **MediaWiki:** Hinter Wikipedia und vielen anderen Wiki-Seiten steht die selbe Software: MediaWiki [46]. Diese Software bietet eine sogenannte *MediaWiki action API*, die viele Informationen zu allen Artikeln eines Wiki zurückliefern kann. Leider sind die Daten nicht über einen zentralen Aufruf abrufbar sondern es werden mehrere Abfragen in folge benötigt, um z.B. die URL eines der Bilder des Artikels zu ermitteln.
- **DBpedia:** Die zweite API, die mithilfe eines Prototypes getestet wurde ist *DBpedia* [47]. DBpedia stellt die strukturierten Informationen aus Wikipedia in strukturierter Form zur Verfügung. Auch diese API folgt dem Crowd Sourcing Prinzip und ist frei zugänglich. Zusätzlich können dort alle Informationen über einen zentralen Zugriff abgerufen werden, indem die benötigten Daten über URL-Parameter spezifiziert werden können. Außerdem bietet diese API die Daten in aufbereiteter Form an: Links können direkt aufgelöst werden, es wird das selbe Thumbnail ausgeliefert welches im original Artikel ausgewählt ist und es kann auf alle Daten der kompakten Infobox in der oberen rechten Ecke strukturiert zugegriffen werden.

Durch die einfachere Handhabung der *DBpedia* API wurde für den weiteren Verlauf entschieden auf diese API zu setzen und alle Informationen zu POIs und Städten von diesem Zugang abzufragen. Damit können erweiterte Infos geladen werden, die dem Nutzer während seines Trips kontinuierlich angezeigt werden können, um die Erfahrung weiter zu verbessern.

4.3 Kartendaten

Neben den POIs werden zur Aufbereitung der Informationen weitere Daten benötigt. Allen voran sind Kartendaten essenziell, um dem Nutzer Orientierung und eine Übersicht über die Lage der ausgewählten POIs zu geben. Der *Travlyn* Client soll auf Mobilgeräten laufen, im Besonderen auf Android Geräten, wie in Kapitel 3 beschrieben. Aus diesem Grund muss ein Kartenformat gewählt werden, welches auf entsprechenden Geräten angezeigt werden und genutzt werden kann.

Auch bei diesem Thema stehen sich grundlegend zwei große Anbieter gegenüber: Google [39] und OpenStreetMap [43]. Aufgrund der ausführlichen Beschreibung im vorangegangenen Kapitel soll an dieser Stelle auf eine weitere Ausführung verzichtet werden. Auch bei dieser Entscheidung kann die Wahl nur auf OpenStreetMap fallen, da dies eine Crowd Sourced Datenbasis ist, welche größtenteils frei von Lizenzfragen in den meisten Umfeldern genutzt werden kann. Abgesehen vom Kostenfaktor ist dies der größte Vorteil gegenüber Services von Google, die allerdings meist im Funktionsumfang und der gebotenen Qualität überlegen sind.

Somit standen einige Android Dienste zur Auswahl, welche auf OpenStreetMap Karten operieren. Die Wahl fiel schlussendlich auf *osmdroid* [48]. Die Bibliothek *osmdroid* kann als alternative zum Android MapView, welcher auf Google Karten operiert, genutzt werden. Sie bietet diverse Features wie das Herunterladen von Karten, Icons und anpassbare Overlays. Besonders die Möglichkeit Karten auch offline anzeigen zu können bietet für die *Travlyn* Applikation einen Vorteil, der ausgenutzt werden kann, wenn die Netzqualität unterwegs nicht ausreicht, um Kartendaten in angemessener Zeit herunterzuladen.

Die Bibliothek kann über *Gradle* (siehe Unterabschnitt 2.3.2) eingebunden werden und ist damit leicht zu benutzen. Da diese Bibliothek Open Source ist und frei auf GitHub zur Verfügung steht, ist die Nutzung kostenfrei. Außerdem gibt es aktuell eine starke Weiterentwicklung und regelmäßige Releases. Hiervon versprechen wir uns bald weitere unter verbesserte Funktionen, die wir für *Travlyn* nutzen können.

5 Konzept

(Joshua)

In diesem Kapitel soll beschrieben werden, wie die in Abschnitt 2.2 beschriebenen Frameworks und Technologien in diesem Projekt zusammen spielen sollen. Hierzu wird zum einen auf die Architektur des Servers, der die API zur Verfügung stellt, eingegangen und zum anderen beschrieben wie der mobile Client aufgebaut ist. Als letztes wird aufgezeigt, wie beide Applikationen miteinander kommunizieren.

5.1 Kommunikationsschema

Da bei diesem Projekt zwei unabhängige Applikationen entstehen (zum einen der mobile Client und zum anderen der Server), ist es sehr wichtig die Schnittstelle zwischen beiden möglichst früh festzulegen. Zudem ist es möglich aus der Beschreibung der Schnittstelle ein Code-Gerüst zu generieren, siehe Unterabschnitt 2.2.2. Aus diesen Gründen wurde entschieden mit dem Kommunikationsschema zu beginnen.

5.1.1 Entitäten

Die entstandene Swagger Beschreibung der API beinhaltet die grundlegende Struktur der Entitäten, die in unserer Applikation genutzt werden sollen. Dazu zählen unter anderem:

- **Stop:** Die POIs, die dem Nutzer angezeigt und zur Auswahl gestellt werden sollen werden in der Stop Entität gespeichert und kommuniziert. Diese Entität beinhaltet alle notwendigen Informationen, wie den Namen des interessanten Punktes, die geografische Lage und die Kategorie, welchem der Punkt zugeordnet ist. Des Weiteren sind Informationen zu Bewertungen enthalten, die von Nutzern abgegeben wurden. Damit bildet die Stopentität die Grundlage für einen *Trip*, welcher aus einer Folge von *Stops* aufgebaut ist.



Abbildung 5.1: Swaggerbeschreibung der Stopentität

Abbildung 5.1 zeigt für die Stop Entität beispielhaft, wie eine Entitätsbeschreibung auf dem von Swagger bereitgestellten UI aussieht. Alle Elemente, die mit einem roten Stern gekennzeichnet sind, sind zwingend erforderlich um eine solche Entität anzulegen. Für jedes Attribut ist eindeutig festgelegt, welchen Typ es hat, ein Beispiel und eine kurze Beschreibung, welche die Benutzung erleichtern soll. Wie an den Attributen *category* und *ratings* zu erkennen, können die einzelnen Entitäten untereinander Verschachtelt werden, um eine konsistente und vollständige Typisierung zu erreichen.

- **Trip:** Ein *Trip* ist wie bereits erwähnt eine Abfolge von Stop Entitäten. Entsprechend ist der zentrale Bestandteil einer Trip Entität eine Sammlung von Stop

Entitäten. Daneben gibt es weitere Informationen zu der zugeordneten Stadt, zum Veröffentlichungsstatus des Trips und zu ggf. vorliegenden Ratings.

- **City:** Jede Stadt, welche in *Travlyn* besucht werden kann, wird in einer City Entität gespeichert. Diese Entität beinhaltet Informationen wie Name, Beschreibung und ein die Adresse eines aussagekräftigen Bildes der Stadt. Einer Stadt sind alle in dieser Stadt liegenden Trips zugeordnet.
- **User:** Die User Entität beinhaltet alle klassischen Informationen zu einem Benutzer der App, dazu zählen z.B. seine E-Mail Adresse, seine technische ID und sein Name. Vor allem für das in Unterabschnitt 3.3.1 beschriebene Nutzungsprofil *Registered User* ist diese Entität von zentraler Bedeutung, da eine Identifizierung des Nutzers notwendig ist um Trips zu speichern oder persönlich zugeordnete Informationen anzulegen und zu verwalten.

Neben den hier beschriebenen Entitäten gibt es noch einige weitere kleinere Entitäten in diesem Projekt, wie z.B. das *token* oder wie bereits in der Stop Entität verwendet das *rating* und die *category*. Diese Entitäten und alle genaueren Beschreibungen (wie Abbildung 5.1) sind im Sinne der Übersichtlichkeit nicht dargestellt, können aber bei Abruf der API eingesehen werden ¹.

5.1.2 Requests

Neben den beschriebenen Entitäten beinhaltet die Swaggerbeschreibung Schemata, wie bestimmte Informationen von der API abgerufen werden können. Diese Beschreibung bietet jedem die Möglichkeit die API zu erkunden und schreibt damit genau vor, welche Services der Server bereitstellen muss aber auch wie der Client bestimmte Informationen erfragen kann.

Im Folgenden wird einer der Requests genauer beschreiben und die entsprechenden Elemente im von Swagger bereitgestellten UI gezeigt. Neben diesem beispielhaften Request gibt es viele weitere Requests mit unterschiedlichsten Spezifikationen, welche bei Bedarf auf der in Unterabschnitt 5.1.1 bereits beschriebenen Website Abge-

¹<https://travlyn.rafael-muesseler.de/travlyn/travlyn/1.0.0/swagger-ui.html>

rufen werden können und sogar mit direktem Zugriff auf die implementierte API getestet werden können.

- **/stop/{stopId}**: Durch die Angabe einer technischen *stopId* kann über diese Schnittstelle eine Stop Entität zu erfragen. Im Swagger UI wird der Request mit seiner Beschreibung und allen erforderlichen Parametern angezeigt.



Abbildung 5.2: Beschreibung des *getStop* Requests im Swagger UI

Außerdem werden die möglichen Antworten der API auf diesen Request angezeigt, um zu verdeutlichen was der Client genau zu erwarten hat und welche Fehler bei der Benutzung der Schnittstelle auftreten können.

Wie in Abbildung 5.3 zu sehen wird die Antwort im JSON Format zurückgegeben, falls kein Fehler aufgetreten ist. In diesem Fall ist eine Antwort im selben Format zu erwarten, wie unter „Example Value“ dargestellt. Dies ist die JSON Darstellung der bereits vorgestellten Stop Entität (siehe Unterabschnitt 5.1.1). Es sind alle Attribute, wie z.B. die Id oder der Name des POI, wiederzuerkennen und in diesem Fall mit den gleichen Beispielwerten belegt, die in der Entitätsbeschreibung festgelegt wurden. Neben der erfolgreichen Antwort mit HTTP Code 200 sind alle Fehlercodes, die von der API zurückgegeben werden können kurz beschreiben. Zusätzlich zu den hier gezeigten Codes ist der Code 500 (*Internal Server Error*) zu berücksichtigen, welcher immer auftreten kann.

Responses

Response content type

application/json

Code	Description
200	<div>successful operation</div> <div><div>Example Value Model</div><div><pre>{ "average_rating": 0.98, "category": { "id": 123, "name": "tourism" }, "description": "This is a description about the Statue of Liberty", "id": 123, "latitude": 123.456, "longitude": 123.456, "name": "Statue of Liberty", "pricing": 50, "ratings": [{ "description": "This is a description of a Rating.", "id": 123, "rating": 0.75, "user": { "email": "test@email.com", "id": 123, "name": "Test User", "token": { "id": 123, "ip_address": "192.168.0.1", "token": "re7sr75a<7dfg8df6g84bcd5flv6a8sx" } }] }</pre></div></div>
401	<div>You are not authorized to perform this action</div>
403	<div>Forbidden</div>
404	<div>Not Found</div>

Abbildung 5.3: Beschreibung des *getStop* Requests im Swagger UI

5.2 Server Architektur

Wie in Unterabschnitt 2.2.2 beschrieben ist es möglich aus der oben beschriebenen Swagger Beschreibung ein komplettes Code-Gerüst für eine Spring Applikation zu

generieren. Diese Funktionalität wurde sich für das vorliegende Projekt zunutze gemacht. Damit entstand eine Spring Applikation, welche die in Unterabschnitt 2.2.3 beschriebenen Patterns verwendet und implementiert.

In Abbildung 5.4 ist eine Übersicht über die grobe Struktur der Server Applikation zu sehen. Dieses UML Diagramm ist stark vereinfacht und beinhaltet nur beispielhafte Klassen, die symbolisch für einen größeren Verbund stehen sollen. So existiert wie bereits beschreiben nicht nur die Stop API und die dazugehörigen Interfaces und Klassen sondern noch viele weitere wie z.B. die User oder Trip API. Trotzdem ist erkennbar, dass sich die Serverarchitektur in mehrere Bereiche aufteilen lässt:

- **Spring-Klassen:** Zur Nutzung von Spring werden einige Klassen benötigt, um bestimmte Konfigurationen und Dienstprogramme aufzusetzen. Diese Klassen werden an sehr vielen Stellen über *Dependency Injection* (siehe Unterabschnitt 2.2.3) in Spring und andere Teile der Architektur eingebunden. Besonders hervorzuheben ist an dieser Stelle die Klasse *TravlynServer*, welche die Methode enthält, um die komplette Springapplikation zu starten.
- **Controller:** Controller liegen ebenfalls unter starker Kontrolle des Spring Frameworks. Alle Requests, die auf der API eingehen werden von Spring an den entsprechenden Controller übergeben. Diese sind für die Verarbeitung der Anfragen verantwortlich und extrahieren Parameter, allerdings beinhalten sie keinerlei Businesslogik, sondern delegieren die Aufgaben an die zentrale Serviceklasse. Diese ist ebenfalls über *Dependency Injection* eingebunden, um die Verwaltung der Beziehung dem Framework zu überlassen. Nach der Ausführung der Anfrage werden die entsprechenden Informationen zurückgegeben und für eine entsprechende HTTP-Antwort gesorgt. Durch dieses Verfahren können serverinterne Informationen wie detaillierte Fehlernachrichten vor dem Nutzer der API versteckt werden, um Angriffe zu erschweren.

Neben der funktionalen Bedeutung für die Anwendung beinhalten die Controller Dokumentation entsprechend zu der Swagger-Beschreibung. Diese sorgt dafür, dass die *Travlyn* API selbständig von Nutzern erkundet werden kann.

- **Service:** Die Klasse *TravlynService* ist das Herzstück des Servers und beinhaltet die meiste Businesslogik. Sie steht zum Teil unter der Kontrolle des Spring Fra-

meworks, muss aber komplett händisch implementiert werden. In dieser Klasse werden alle Operationen, die der Server ausführen muss (z.B. Login eines Nutzers, Sammeln der POI's für eine Stadt oder erzeugen eines Trips) implementiert. Diese Klasse hält alle anderen Komponenten, wie die Controller, den Datenbankzugriff und weitere externe Funktionalitäten zusammen und kontrolliert ihr zusammenwirken.

- **DTOs und Entitäten:** Auf dem Server werden Objekte in zwei unterschiedlichen Formaten gehalten, welche nicht unter der Kontrolle vom Spring aber zum Teil unter der Kontrolle von Hibernate stehen. Zum einen existieren sog. Data Transfer Objects (DTO)s. Diese beinhalten ausschließlich die Informationen, die als Ergebnis einer API-Abfrage übermittelt werden. Zum anderen sind Datenentitäten entstanden, wie sie von Hibernate genutzt werden. Die Entitäten besitzen die gleiche Struktur, wie die zugrundeliegenden Datenbanktabellen und damit alle vorliegende Informationen. Dieses Pattern wird genutzt um bestimmte Informationen zu verstecken und eine Trennung zwischen der Sicht des Servers auf die Daten und der Sicht des API Nutzers zu erreichen. In der Literatur wird dieses Pattern als *Interface Segregation Principle* genannt und ist Teil des *SOLID* Konzepts [49]. Beide Objekttypen sind über entsprechende Methoden ineinander umwandelbar.
- **Hilfsklassen:** Neben allen bereits genannten Funktionalitäten müssen weitere externe Funktionen eingebunden werden. Im Fall von *Travlyn* hauptsächlich fremde APIs, die alle nötigen Daten zu POIs, Städten und Karten zur Verfügung stellen. Diese Zugriffe werden im Service benötigt, um die angefragten Operationen ausführen zu können. Dieser Teil der Applikation steht nicht unter der Kontrolle eines Frameworks und wurde händisch integriert.

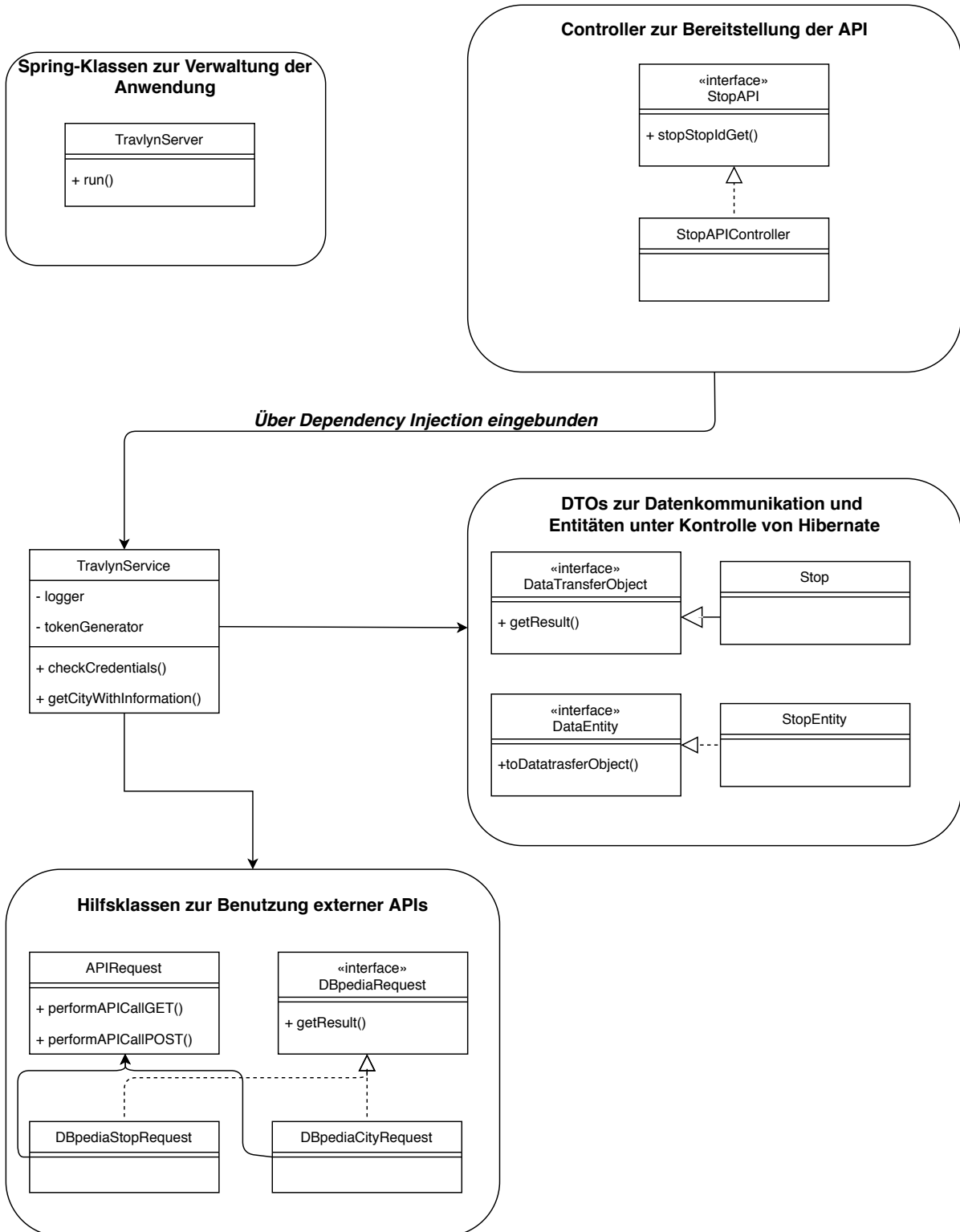


Abbildung 5.4: Struktur des *Travlyn* Servers als UML Diagramm. Zu Beachten ist, dass dieses Diagramm stark vereinfacht und nicht vollständig ist, um die Übersichtlichkeit zu wahren.

5.3 Client Architektur

Neben der automatischen Generierung von Code, welche das Spring Framework nutzt, ist es mit Swagger ebenfalls möglich, Code für den Client zu generieren. Da diese Generierung viel Arbeit und Einrichtungsaufwand spart, wurde auch das Grundgerüst für den *Travlyn* Client automatisch generiert.

5.3.1 Design Guidelines

Für die Entwicklung von Benutzeroberflächen für mobile, Android-fähige Geräte, stellt Google Design Guidelines zur Verfügung. Das sog. Material Design [50] ist eine Menge an Richtlinien und Vorgaben, wie Anwendung auf dem Android Betriebssystem entworfen werden sollen. Da dies nur Richtlinien sind, ist eine Einhaltung nicht zwingend erforderlich, jedoch bringt es einige Vorteile mit sich:

Ein Benutzer assoziiert einzelne Komponenten einer Anwendung – wie beispielsweise ein Radiobutton – mit einer bestimmten Aktion. Wenn nun alle Applikationen die verschiedenen Komponenten mit die gleichen Aktionen belegen, erleichtert dies die Bedienung einer einzelnen Applikation. So wird durch die Berücksichtigung dieser Richtlinien Konsistenz gewährleistet. Dabei wird auch das mentale Modell des Benutzers weiter entwickelt.

Darüber hinaus erleichtert es auch zumeist die Entwicklung dieser Applikationen, da viele Komponenten durch die Verwendung des Android SDKs bereits implementiert sind, aber auch leicht wieder verwendet und bei Änderungen durch das Framework ohne großen Aufwand angepasst werden können. [51]

5.3.2 Aufbau und Struktur der Benutzeroberfläche

Aufbau und Struktur der Benutzeroberfläche orientiert sich im Allgemeinen an den in Unterabschnitt 5.1.2 beschriebenen möglichen Anfragen. So werden die zu einem Use Case gehörigen Aktionen unter einen Menüpunkt gefasst, welcher sich im ausklappbaren Menü – diese Komponente der Applikation heißt *NavigationDrawer* und lässt sich über den in der Suchleiste befindlichen Knopf (siehe Abbildung 5.5) öffnen

– der Applikation befindet.

Um jedoch den Aspekt der Struktur etwas weiter im Detail zu beleuchten, wird im Folgenden beispielhaft ein ausgewählter Use Case beschrieben.

Beim Start der Applikation erscheint eine Karte (dargestellt durch OSMdroid Abschnitt 4.3) sowie eine Suchleiste, welche dem Zweck dient, nach Städten zu suchen. Bei erfolgreicher Suche nach einer Stadt erscheint ein Dialog, welcher zusätzliche Informationen zur Stadt sowie deren Sehenswürdigkeiten anzeigt. Von dort aus ist es möglich auf eine Detailansicht der Sehenswürdigkeiten abzuspringen sowie alle zu dieser Stadt gehörigen öffentlichen Trips anzuzeigen und zu favorisieren. Dieses Szenario ist in Abbildung 5.5 dargestellt.

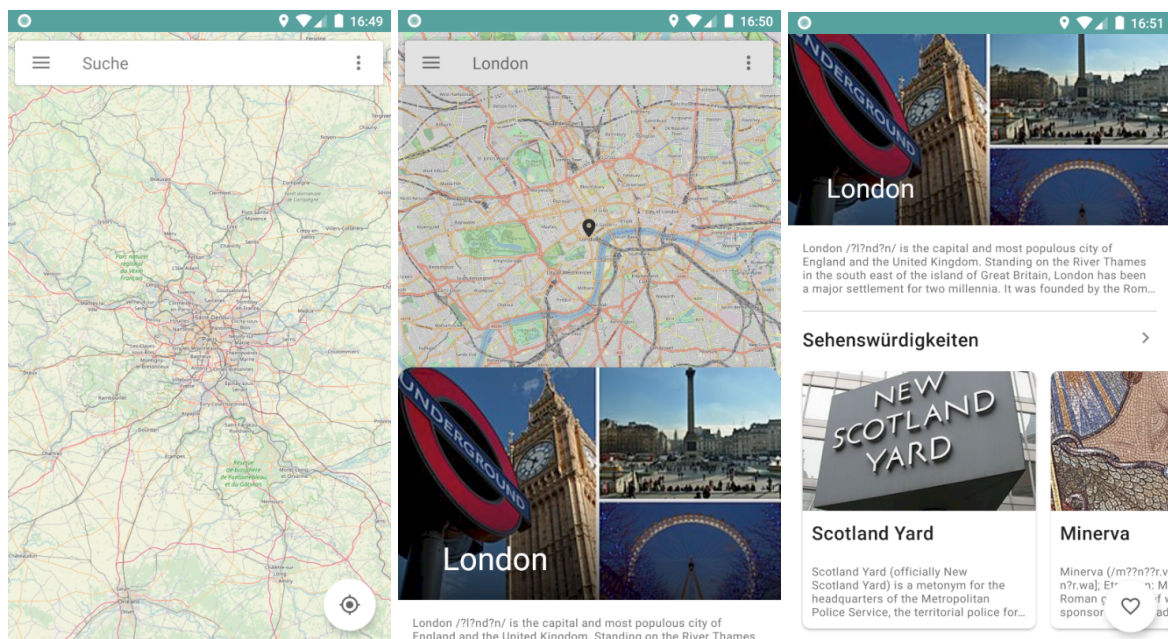


Abbildung 5.5: Aufbau und Struktur der Benutzeroberfläche anhand eines konkreten Use Cases

Alle weiteren Use Cases werden mithilfe der von Google bereitgestellten Benutzeroberflächenkomponenten und gemäß des Material Designs erstellt, wie beispielsweise die Erstellung oder Bearbeitung von Trips.

5.3.3 Kommunikation zum Server

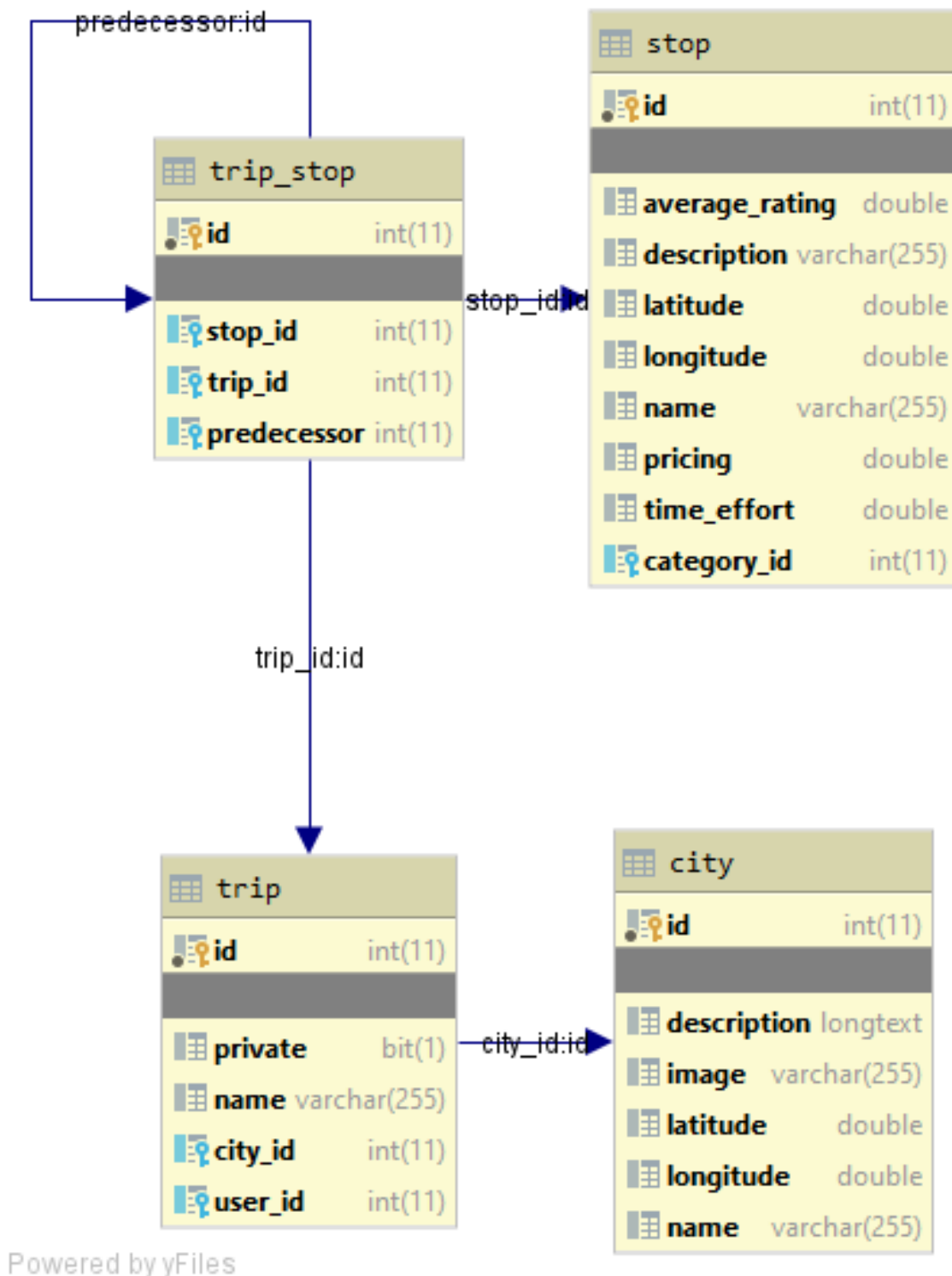
Wie bereits erwähnt, lässt sich ebenso die Kommunikation zum Server durch Swagger generieren. Dabei wird für jeden Endpoint der API eine Methode generiert, welche genau die Objekte entgegennimmt, die auch von der API gefordert sind. Auch lassen sich in gleicher Weise die DTO Objekte generieren, sodass die vollständige Kommunikation zwischen Server und Client gewährleistet ist. Um die Kommunikation zu beschleunigen, werden einige Entitäten bzw. Daten nach der Anfrage an den Server lokal zwischengespeichert.

5.4 Infrastruktur

Wie bereits in Unterabschnitt 2.4.5 beschrieben wird der *Travlyn* Server über ein Docker Image (siehe Unterabschnitt 2.2.5) ausgeliefert. Außerdem wird mittels einer *docker-compose* Datei eine Datenbank angebunden. Für das Projekt *Travlyn* wurde entschieden, eine relationale Datenbank – in diesem Falle eine *MariaDB* – zu verwenden.

Das relationale Datenbankschema ähnelt dem von der API exponierten Entitäten Schema (siehe Unterabschnitt 5.1.1), jedoch mit kleineren Unterschieden. So enthält beispielsweise das Datenbankschema der *User*-Entität noch zusätzlich zu den bereits beschriebenen Daten, ein gehashtes Passwort sowie einen dazugehörigen *Salt* – eine zufällige Zeichenfolge, die an das Passwort vor dem Hashen angehängt wird, sodass bei unbefugtem Zugriff auf die Datenbank verhindert wird, dass gleich Passwörter gleiche Hashes erzeugen –, um die Vertraulichkeit des Passworts zu gewährleisten.

So werden auch Daten vor der Exponierung aufbereitet. Beispielsweise werden die zu einem *Trip* gehörigen *Stops* bereits in der richtigen Reihenfolge zurückgegeben, während auf der Datenbank die Tabelle der N:N Relation zwischen *Trip* und *Stop* noch ein weiteres Attribut besitzt: *predecessor* (dt. Vorgänger). Ist der Vorgänger *null*, so ist dieser *Stop* der erste im *Trip*. Der nächste *Stop* hat die ID des Vorgängers als Attribut gespeichert, welcher wieder als Vorgänger für den nächsten *Stop* fungiert usw. Abbildung 5.6 zeigt einen Ausschnitt des Datenbankschemas von *Travlyn* mit Fokus auf der Relation zwischen der *Trip*- und *Stop*-Entität.

Abbildung 5.6: Ausschnitt aus dem Datenbankschema von *Travlyn*

6 Implementierung

In diesem Kapitel werden einige Implementationsdetails dargestellt und erläutert, welche entweder einen kritischen Aspekt des Projekts darstellen oder von hoher Relevanz für den Erfolg des Projektes sind.

6.1 Controller Interfaces

Wie bereits erwähnt sind die sog. Controller in einer MVC Architektur für den Behandlung von Benutzereingaben zuständig. Im Falle einer RESTful API – hier durch Spring umgesetzt – dienen Controller dem Zweck, alle eingehenden HTTP Anfragen zu behandeln. Dafür bietet Spring Annotationen, mit dessen Hilfe eindeutig definiert werden kann, wie die Anfrage auszusehen hat. Diese Definition eines sog. Endpoints beinhaltet immer einen Pfad, unter welchem der Endpoint erreicht werden kann, und eine HTTP Methode. Es kann zusätzlich noch festgelegt werden, welche HTTP Header bei einer Anfrage angegeben werden sollen und welcher MIME Type zurück gegeben wird.

Listing 6.1 zeigt beispielhaft, wie eine solche Definition eines Endpoints aussehen kann. Die eigentliche Beschreibung des Endpoints wird durch die Annotation `@GetMapping` beschrieben.

Um jedoch eine ausführlichere Beschreibung des Endpoints zu erstellen, können die von Swagger bereitgestellten Annotation verwendet werden. So lassen sich beispielsweise die Parameter, die Rückgabewerte aber auch die Authentifizierungsmethode beschreiben. Außerdem lassen sich noch Beschreibungen einfügen, um die Benutzung der API zu erleichtern und zu dokumentieren. Für die City API wird ein String als Parameter benötigt, mit dessen Hilfe die entsprechende Stadt zurückgegebene werden kann. Außerdem wird ein API Key benötigt, um sich zu authentifizieren.

```
1 @Api(value = "city")
2 public interface CityApi {
```

```
3  @ApiOperation(  
4      value = "Get City by search term",  
5      nickname = "getCity",  
6      notes = "",  
7      response = City.class,  
8      authorizations = {@Authorization(value = "ApiKeyAuth")},  
9      tags = {"city"})  
10 @ApiResponses(value = {  
11     @ApiResponse(code = 200, message = "successful operation", ↵  
12         ↳ response = City.class),  
13     @ApiResponse(code = 401, message = "You are not authorized ↵  
14         ↳ to perform this action")})  
15 @GetMapping(  
16     value = "/city",  
17     produces = {"application/json"})  
18 @PreAuthorize(value = "hasRole(" + ROLE_API_USER + ")")  
19 ResponseEntity<City> getCity(@ApiParam(value = "Name of the ↵  
20     ↳ city that should be searched for", required = true, ↵  
21     ↳ defaultValue = "", example = "Düsseldorf") @Valid ↵  
22     ↳ @RequestParam(value = "query") String query);  
23 }
```

Listing 6.1: Nutzung der beschriebenen Annotationen eines Controllers am Beispiel der City API

6.2 Authentifizierung

Travlyn stellt insgesamt zwei verschiedene Authentifizierungsrollen bereit. Wie Abbildung 3.1 zeigt, gibt es zum einen die Rolle des API-Benutzers und die des registrierten Benutzers. Dem API Benutzer ist es erlaubt, Informationen wie beispielsweise über Trips anzufragen, wohingegen ein registrierter Benutzer auch die Möglichkeit hat, einen Trip zu erstellen.

Der Prozess der Authentifizierung basiert bei *Travlyn* auf sog. Tokens. Sobald sich ein Benutzer registriert oder anmeldet, wird ein Token generiert – das Token besteht aus



Abbildung 6.1: Authentifizierungsprozess von *Travlyn* mittels Spring Security Filter Chain

32 Zeichen und Ziffern und besitzt zudem ein Ablaufdatum, welches bei Überschreiten das Token als ungültig gekennzeichnet –, auf der Datenbank abgespeichert und zurück an den Benutzer gesendet. Um nun Endpoints anzufragen, welche eine Authentifizierung benötigen, wird dieses Token im `Authorization-Header` im Format `Bearer <token string>` gesendet. Auf dem Server erfolgt nun die Validierung des Tokens. So kann außerdem festgestellt werden, wer eine bestimmte Aktion ausgeführt hat.

Spring stellt für diesen Prozess einen Mechanismus bereit, um diese Authentifizierung möglichst einfach umzusetzen: Bevor eine Anfrage an einen bestimmten End-

point zugelassen und die entsprechende Controller-Methode (wie bei Listing 6.1 beispielhaft gezeigt) aufgerufen wird, durchläuft diese Anfrage eine Reihe von Filtern. Diese sog. Filter Chain beinhaltet unter Anderem auch Authentifizierungsmechanismen, die sich eigens definieren lassen. So wurde für *Travlyn* ein Filter namens `AuthenticationTokenFilter` entwickelt, der dazu dient, das Token aus dem HTTP-Header zu extrahieren und zu validieren.

Wenn das Token valide ist und auch der korrekten Benutzerrolle – API- oder registrierter Nutzer – zugeordnet werden kann, wird die Anfrage an die entsprechende Controller Methode weitergeleitet und der entsprechende Nutzer im sog. `SecurityContext` gespeichert, sodass auf die Rolle des Benutzers und auch auf den Benutzer selbst beim Prozessieren der Anfrage zugegriffen werden kann. Ist dies nicht der Fall, so wird diese Anfrage als nicht-authentifiziert markiert. Abbildung 6.1 visualisiert den Authentifizierungsprozess, wie er bei *Travlyn* umgesetzt wurde.

Um die zugelassenen Benutzerrollen für einen Endpoint der API festzulegen, stellt das Modul Spring Security unter anderem die Annotation `@PreAuthorize` zur Verfügung. Als Parameter wird ein in der Spring Expression Language (SpEL) geschriebener Ausdruck übergeben, wie beispielsweise `hasRole(API_USER)` in Listing 6.1. So lässt sich die gesendete Authentisierung des Benutzers automatisiert überprüfen.

6.3 Namenszuordnung zwischen den APIs

(Joshua)

Wie in Kapitel 4 beschrieben werden für *Travlyn* mehrere verschiedene APIs benutzt um alle relevanten Informationen zu sammeln. Leider ergeben sich bei einer derartigen Parallelbenutzung häufig Inkompatibilitäten, gerade wenn die APIs nicht von demselben Anbieter stammen.

Im Fall von *Travlyn* besteht eine solche Inkompatibilität zwischen der OpenRoute-Service API, welche die Positionen und Namen der POIs liefert und der DBpedia API, welche die entsprechenden Informationen von Wikipedia bezieht: OpenRoute-Service liefert die Namen der Punkte in der Landessprache z.B. „Schloss Karlsruhe“. Dem gegenüber können bei DBpedia nur englische Namen abgefragt werden, wie z.B.

„Karlsruhe Palace“. Des Weiteren stehen selbst in englischsprachigen Ländern einige Namen in Konflikt wie z.B. „Coca Cola London Eye“ auf ORS zu „London Eye“ auf DBpedia. Durch diese Konflikte fehlen viele Informationen in der *Travlyn* App und müssen manuell nachgepflegt werden.

Da diese Arbeit nicht darauf ausgelegt ist eine zur Veröffentlichung bereite App zu erstellen, sondern Konzepte und Wege aufzuzeigen, welche verwendet werden können wurde diese Nachpflege lediglich für einige Sehenswürdigkeiten in Karlsruhe durchgeführt und die entsprechende Infrastruktur für eine ausgedehntere Pflege geschaffen. Sollte die App produktiv veröffentlicht werden sind mehrere Möglichkeiten der Pflege vorstellbar: Es werden entsprechende Agenturen gebucht um die Zuordnung manuell in einer Datenbank zu pflegen und für möglichst viele Städte zur Verfügung zu stellen. Außerdem könnte auf einen nutzerbasierten Ansatz gesetzt werden, bei dem die Nutzer die Möglichkeit haben Fehler und fehlende Informationen nachzupflegen, sobald sie auffallen. Dies hätten den Vorteil, dass es für den Hersteller kostenlos ist und gut skaliert. Wahrscheinlich wäre eine Kombination aus beiden Möglichkeiten der beste Weg.

6.4 Umgehung der API Beschränkung für DBpedia

Der Umgang mit DBpedia für dieses Projekt wurde neben dem oben beschriebenen Mappingproblem auch durch die Abfragebeschränkung erschwert. Trotz eines registrierten persönlichen API Schlüssels stehen der *Travlyn* App nur 100 Abfragen pro Minute Verfügung. Außerdem gelten weitere strenge Einschränkungen, wie in [52] beschrieben.

Unter Betrachtung des Umstandes, dass jede neu abgefragte Stadt und jeder in dieser Stadt befindliche POI abgefragt werden muss, wenn ein Nutzer sich per *Travlyn* über eine Stadt informieren möchte, ist leicht zu erkennen, dass diese Beschränkung kaum eingehalten werden kann. Beispielsweise liefert ORS für die Stadt London ca. 670 POIs, die alle in einer für den Nutzer angemessenen Zeit abgefragt werden müssten. Dies kollidiert eindeutig mit der Minutenbeschränkung von DBpedia. Aus diesem Grund wurde für *Travlyn* ein Konzept entwickelt, was ein *lazy fetching* ermöglicht

und die nötigen Abfragen erst nach und nach absetzt und den Nutzer möglichst wenig einschränkt: Für eine neue (sich nicht im Cache befindliche) Stadt wird sowohl die Abfrage für die Stadt als auch für bis zu 99 POIs durchgeführt. Dies ist konform mit der Beschränkung von DBpedia und versucht den Nutzer möglichst wenig einzuschränken, indem die POIs bevorzugt abgefragt werden, für die sich der aktuelle Nutzer laut seiner Präferenzen interessiert. Es wird eine entsprechend reduzierte Antwort an den Client geschickt, mit dem Hinweis, dass einige POIs fehlen könnten. Nun hat der Nutzer die Möglichkeit, nach der Durchsicht der vorhandenen Sehenswürdigkeiten, weitere anzufordern und damit weitere Anfragen auf DBpedia auszulösen. Ebenso löst jede weitere Suche nach der vorliegenden Stadt weitere Requests aus. Dieses Verfahren operiert, bis alle POIs abgefragt wurden. Abbildung 6.2 zeigt vereinfacht den allgemeinen Ablauf des Verfahrens.

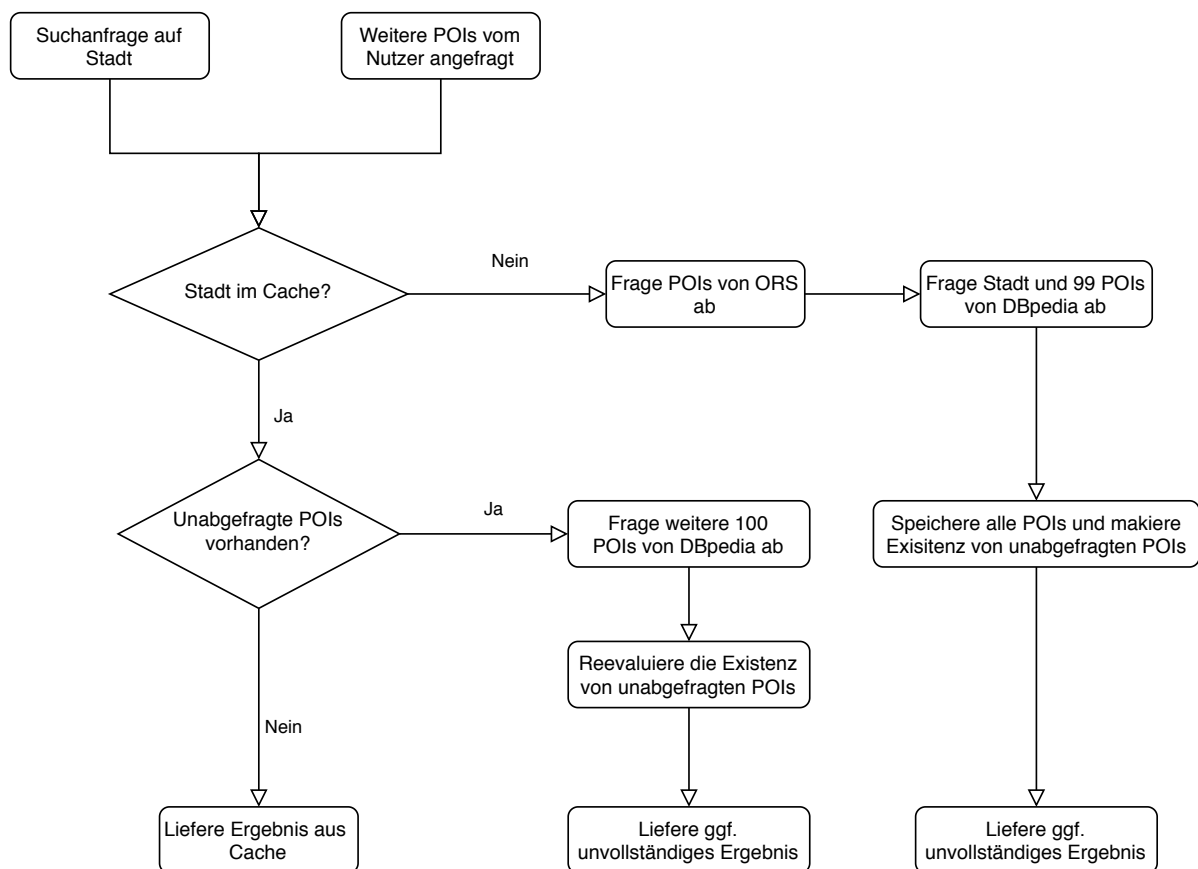


Abbildung 6.2: Flussdiagramm zum Ablauf des *lazy fetching* von DBpedia.

Über diesen Mechanismus können die großen Abfragenwellen abgefedert werden und die Beschränkung kann eingehalten werden.

Aus dem Vorangegangenen ergibt sich neben der Notwendigkeit eines *lazy fetchings* eindeutig, dass diese Beschränkungen für einen Produktivbetrieb mit vielen Nutzern nicht ohne Weiteres einhaltbar sind. Für diesen Fall sind zwei mögliche Lösungswege zu betrachten: Wenn für einen Produktivbetrieb mehr und stärkere Rechenressourcen in Form von Servern zur Verfügung stehen, als in diesem Studienprojekt, wäre es möglich, eine Instanz des DBpedia SPARQL Endpoints selber zu betreiben. Dies würde einen Zugangspunkt schaffen, der ohne Beschränkungen genutzt werden könnte. Ein zweiter Ansatz wäre ein *startup-script* zu entwerfen, welches innerhalb der vorgegeben Beschränkungen nach und nach dem Cache mit möglichen Nutzerzielen automatisch füllt, wovon die App freigeschaltet wird. Vorstellbar wäre, den Cache während dieser *Aufwärmphase* z.B. mit allen Hauptstädten zu füllen. Damit würden nur noch weniger Städte im Produktivbetrieb abgefragt und die Beschränkungen könnten, zusammen mit dem *lazy fetching*, eingehalten werden.

Da beide Lösungsansätze ausschließlich für den Produktivbetrieb und nicht für die grundsätzliche Funktionalität benötigt werden, wurden lediglich die Möglichkeiten für eine spätere Implementation geschaffen. Das hier beschriebene *lazy fetching* wurde komplett umgesetzt, da es systemkritisch ist und die Funktionalität sonst stark eingeschränkt wäre.

7 Evaluation

7.1 User Zufriedenheit

7.2 Aufbau

7.3 Ablauf

7.4 Ergebnis

8 Fazit

8.1 Ausblick

Literaturverzeichnis

- [1] Facebook Technologies, L. *Oculus Rift: VR-Headset für VR-fähige PCs* | Oculus. 4/23/2020. URL: https://www.oculus.com/rift/?locale=de_DE#oui-csl-rift-games=star-trek.
- [2] Dredge, D. u. a. *Digitalisation in Tourism: In-depth analysis of challenges and opportunities*. 2018.
- [3] Fielding, R. T. „Architectural Styles and the Design of Network-based Software Architectures“. In: (2000). URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.
- [4] OpenAPI Initiative. *Home - OpenAPI Initiative*. 3/23/2020. URL: <https://www.openapis.org/>.
- [5] SmartBear. *The Best APIs are Built with Swagger Tools | Swagger*. 2020. URL: <https://swagger.io/>.
- [6] Walls, C./ Carnell, J. *Spring Boot in action // Spring microservices in action*. Shelter Island: Manning and Manning Publications Co, 2016 // 2017. URL: https://www.nitinagrawal.com/uploads/2/1/3/6/21361954/spring_boot_in_action.pdf.
- [7] Johnson, R. *Expert one-on-one J2EE design and development*. Programmer to programmer. Indianapolis, Ind. und Great Britain: Wrox, 2003.
- [8] Fowler, M. *Inversion of Control Containers and the Dependency Injection pattern*. 23.01.2020. URL: <https://www.martinfowler.com/articles/injection.html#InversionOfControl>.
- [9] Gamma, E. *Design patterns: Elements of reusable object-oriented software / Erich Gamma ... [et al.]* Addison-Wesley professional computing series. Reading, Mass. und Wokingham: Addison-Wesley, 1995.
- [10] Wunderlich, L. *AOP: Aspektorientierte Programmierung in der Praxis*. S- & -S-pockets. Frankfurt [Main]: Entwickler.press, 2005.

- [11] Özsu, M. T./ Valduriez, P. *Principles of distributed database systems*. 3rd ed. New York: Springer Science+Business Media, 2011.
- [12] Oracle. *Java Transaction API (JTA)*. 3/23/2020. URL: <https://www.oracle.com/technetwork/java/javaee/jta/index.html>.
- [13] Hibernate. *Hibernate. Everything data. - Hibernate*. 3/17/2020. URL: <https://hibernate.org/>.
- [14] Chamberlin, D. D./ Boyce, R. F. „*SEQUEL: A structured English query language*“. In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 1974, S. 249–264.
- [15] Ireland, C. u. a. „*A Classification of Object-Relational Impedance Mismatch*“. In: *First International Conference on Advances in Databases, Knowledge, and Data Applications, 2009: DBKDA '09 ; Gosier, Guadeloupe, France, 1 - 6 March 2006*. Hrsg. von Chen, Q. Piscataway, NJ: IEEE, 2009, S. 36–43.
- [16] Bernd Müller/ Harald Wehr. *Java Persistence API 2*. CARL HANSER Verlag GMBH &, 2012.
- [17] Huber, T. C. *Einstieg in die Welt der Container: Docker-Grundlagen für .NET-Entwickler*. 2019. URL: <https://entwickler.de/online/windowsdeveloper/docker-grundlagen-dotnet-container-579859289.html>.
- [18] Turnbull, J. „*The Docker Book*“. In: (2014). URL: <http://opisboy.bandungbaratkab.go.id/books/DOCKER/James.Turnbull.The.Docker.Book.Containerization.is.the.new.virtualization.B00LRROT14.pdf>.
- [19] Docker Inc. *Docker Virtualisierung verglichen mit virtuellen Maschinen*. 2020.
- [20] Docker Inc. *Docker Hub*. 3/23/2020. URL: <https://hub.docker.com/>.
- [21] Google. *Android Developers*. 3/19/2020. URL: <https://developer.android.com/>.
- [22] Dossey, A. *Kotlin vs. Java: Which is the Better Option for Android App Development?* | Clearbridge Mobile. 2019. URL: <https://clearbridgemobile.com/java-vs-kotlin-which-is-the-better-option-for-android-app-development/>.
- [23] JetBrains. *Kotlin on Android. Now official*. 2017. URL: <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>.
- [24] Company, S. *Maven: The Definitive Guide*. Sebastopol: O'Reilly Media, Inc, 2009.

- [25] Google. *Android Gradle plugin release notes* | *Android Developers*. 2/28/2020. URL: <https://developer.android.com/studio/releases/gradle-plugin>.
- [26] Huizinga, D./ Kolawa, A. *Automated defect prevention: Best practices in software management* / Dorota Huizinga, Adam Kolawa. Piscataway, N.J.: Wiley-IEEE Computer Society and Chichester : John Wiley [distributor], 2007. URL: https://www.researchgate.net/profile/Michael_Karlesky/publication/228711578_Mocking_the_embedded_world_Test-driven_development_continuous_integration_and_design_patterns/links/5512b1c80cf20bfdad51c6b4.pdf.
- [27] Team, J. *JUnit 5 User Guide*. 3/1/2020. URL: <https://junit.org/junit5/docs/current/user-guide/>.
- [28] Fowler, M. *TestCoverage*. 2/29/2020. URL: <https://martinfowler.com/bliki/TestCoverage.html>.
- [29] Myers, G. J. u. a. *The art of software testing*. 2nd ed. Hoboken, N.J. und Great Britain: John Wiley & Sons, 2004.
- [30] A, S. S. *Code Quality and Security* | *Developers First* | *SonarSource*. 2/14/2020. URL: <https://www.sonarsource.com/>.
- [31] Fowler, M. *Continuous Integration*. 2/29/2020. URL: <https://martinfowler.com/articles/continuousIntegration.html>.
- [32] CloudBees. *Jenkins*. 3/2/2020. URL: <https://jenkins.io/>.
- [33] Humble, J./ Farley, D. *Continuous delivery*. Boston, Mass. und London: Addison-Wesley, 2011.
- [34] Balzert, H. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3. Auflage. Lehrbücher der Informatik. Heidelberg: Spektrum Akademischer Verlag, 2009. URL: <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10361884>.
- [35] Lawlor, K. B./ Hornyak, M. J. „SMART GOALS: HOW THE APPLICATION OF SMART GOALS CAN CONTRIBUTE TO ACHIEVEMENT OF STUDENT LEARNING OUTCOMES“. In: 39 (2012), S. 259–267.
- [36] ISO. *ISO/IEC 25010:2011*. 2011. URL: <https://www.iso.org/standard/35733.html>.

- [37] Braun, M. „Nicht-funktionale Anforderungen: Juristisches IT-Projektmanagement“. In: (2016). URL: <https://www.pst.ifi.lmu.de/Lehre/wise-15-16/jur-pm/braun-ausarbeitung.pdf>.
- [38] Mirkovic, J. *Internet denial of service: Attack and defense mechanisms*. The Radia Perlman series in computer networking and security. Upper Saddle River, N.J: Prentice Hall Professional Technical Reference, 2005. URL: <http://proquest.tech.safaribooksonline.de/0131475738>.
- [39] Google. *Geo-location APIs | Google Maps Platform | Google Cloud*. 01.02.2020. URL: <https://cloud.google.com/maps-platform/>.
- [40] Singhal, M./ Shukla, A. „Implementation of Location based Services in Android using GPS and Web Services“. In: 9.1 (2012), S. 237.
- [41] Google. *Google Maps APIs Terms of Service | Google Maps Platform*. 02.12.2019. URL: <https://developers.google.com/maps/terms-20180207?hl=en>.
- [42] The Heidelberg Institute for Geoinformation Technology. *Openrouteservice*. URL: <https://openrouteservice.org/>.
- [43] OpenStreetMap. *OpenStreetMap Deutschland: Die freie Wiki-Weltkarte*. URL: <https://www.openstreetmap.de/>.
- [44] Winkler, P. *Computer-Lexikon 2010: Extra: Tipps zum persönlichen Datenschutz*. Vollst. überarb. Neuaufl. München: Markt+Technik Verl., 2009.
- [45] Foursquare. *Foursquare Location Intelligence for Enterprise*. URL: <https://enterprise.foursquare.com/products/places>.
- [46] MediaWiki. *MediaWiki*. 24.01.2020. URL: <https://www.mediawiki.org/wiki/MediaWiki>.
- [47] DBpedia. *About | DBpedia*. 02.02.2020. URL: <https://wiki.dbpedia.org/about>.
- [48] osmdroid. *osmdroid/osmdroid: OpenStreetMap-Tools for Android*. 3/16/2020. URL: <https://github.com/osmdroid/osmdroid>.
- [49] GOLL, J. *ENTWURFSPRINZIPIEN UND KONSTRUKTIONSKONZEPTE DER SOFTWARETECHNIK: Strategien für schwach ... gekoppelte, korrekte und stabile software*. [Place of publication not identified]: MORGAN KAUFMANN, 2019.
- [50] Google. *Design - Material Design*. 3/18/2020. URL: <https://material.io/design/>.

- [51] Saadia Minhas. *Why You Need UI Guidelines? - UX Planet*. 3/23/2020. URL: <https://uxplanet.org/why-you-need-ui-guidelines-d380e407b759>.
- [52] DBpedia. *REST API | DBpedia*. 2020. URL: <https://wiki.dbpedia.org/rest-api>.