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 23. März 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

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
MVC	Model View Controller
OAI	OpenAPI
ORM	Object-relational Mapping
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 Objec Access Protocol
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

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

(Joshua)

In der heutige 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 einen 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 Geschäftsmodel 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

2 Voraussetzungen

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 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 und welche nicht zwischengespeichert werden, um zu verhindern, dass alte oder ungeeignete Daten versendet werden.

Einheitliche Schnittstelle Die REST API muss eine einheitliche Schnittstelle zur Verfügung stellen, welche den von [**Fielding.2000**] definierten Anforderungen entsprechen muss. Dies vereinfacht die Nutzung der API.

Mehrschichtige Systeme Die Struktur einer RESTful API soll mehrschichtig sein, sodass 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* 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. **[.2252020]**

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 Rückgabetyp 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 OpenAPI 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. [3]

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.

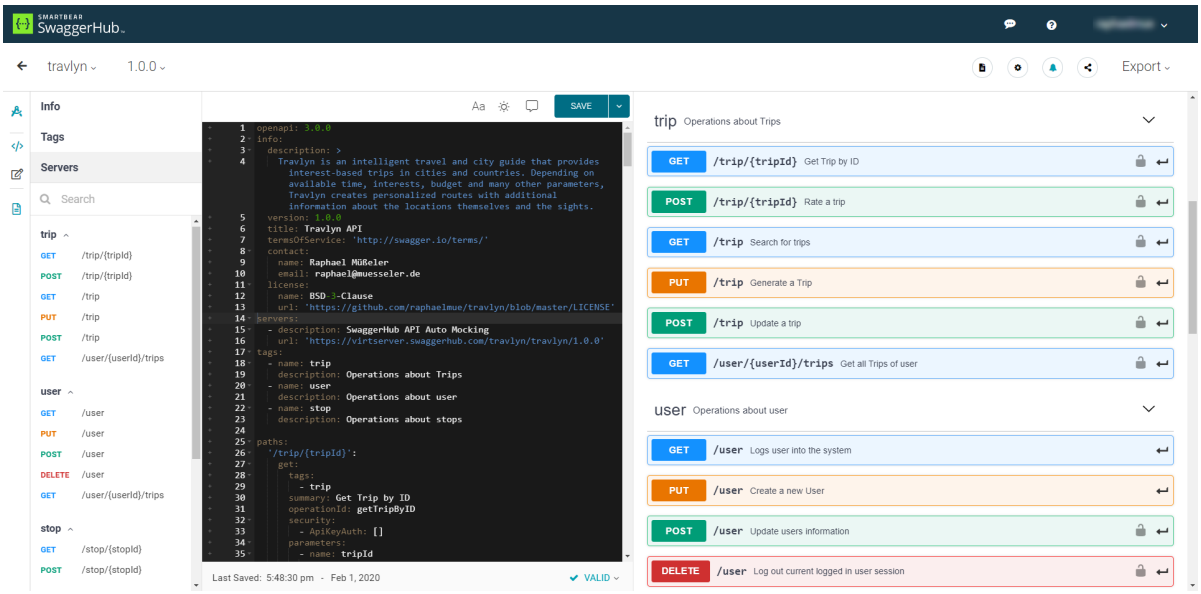

Abbildung 2.1: SwaggerHub Darstellung anhand des Projektes *Travlyn*

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. [4]

Bei der Entwicklung dieses Frameworks wurden die unter anderem in [5] 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 **[Fowler.23.01.2020]** 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).
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. [**Fowler.23.01.2020**]

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. [4]

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 Anwendungslogik und der Businesslogik (Cross-cutting Concern). [6]
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 Anwendungs- 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. [7]

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 lasst sich diese Abstraktion in fast allen Java Umgebungen einsetzen. Die von Java bereitgestellte Java Transaction API (JTA) 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. [8]

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 ent-

hält – 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.

2.2.4 Hibernate

Hibernate 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 eingesetzt Technik der Persistierung sind relationale und meist auch SQL 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. [9]

Java bietet mit der sog. Java Persistence API (JPA) 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 Attribute 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 Reflexion 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 erstellt, 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 nun sehr einfach ausgeliefert werden und sind darüber hinaus in der Lage, sehr schnell zu starten. [**Huber.2019**]

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. [10]

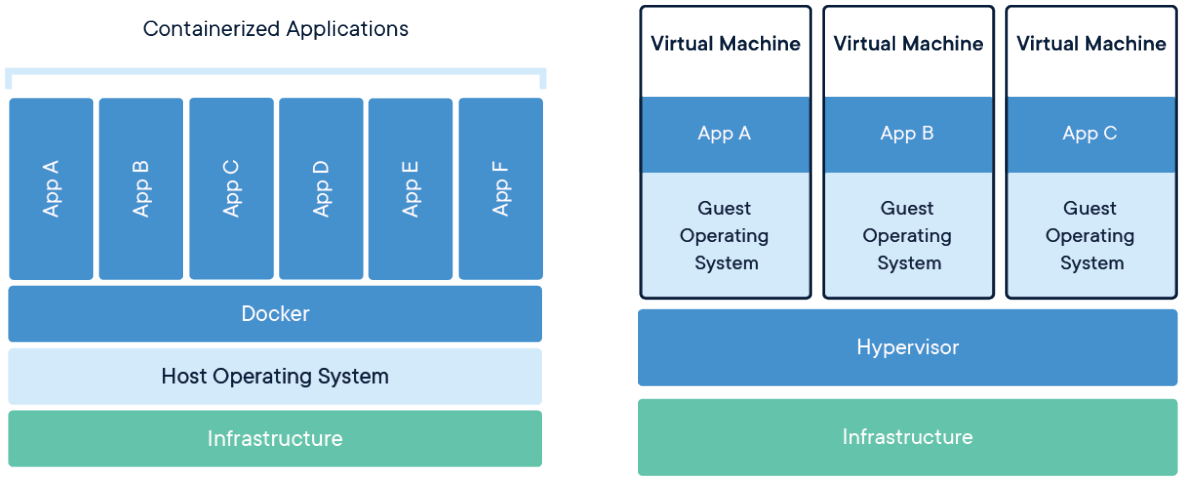

Abbildung 2.2: Docker Virtualisierung verglichen mit virtuellen Maschinen [11]

DockerHub

Eine Platform, die ebenfalls von Docker Inc. bereitgestellt wird, ist der sog. DockerHub. 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 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 derJVM basierende Kotlin, da die bereits genannte Frameworks ebenfalls in Java geschrieben sind und so Einheitlichkeit herrscht. Die in [**Dossey.2019**] 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 einen Bytecode übersetzt, der unter Anderem auf der JVM laufen kann. Seit 2017 wurde Kotlin offiziell von Google für die Entwicklung von Android Apps unterstützt [12].

Ein Vorteil von Kotlin gegenüber Java sing 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. **[Company.2009]**

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. [**Google.2282020**]

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. [**Huizinga.2007**] 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. [**Team.312020**]

Spring unterstützt die Implementierung von JUnit Tests durch die Bereitstellung eines Testsframeworks, 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 seien können, oder nicht unbedingt den Fehler hervorrufen. [**Fowler.2292020b**]

Man unterscheidet bei der Testabdeckung unter Anderem zwischen den in [**Myers.2004**] 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. [**A.2142020**]

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 Regelverstoße 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 [**Fowler.2292020**].

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. [**CloudBees.322020**]

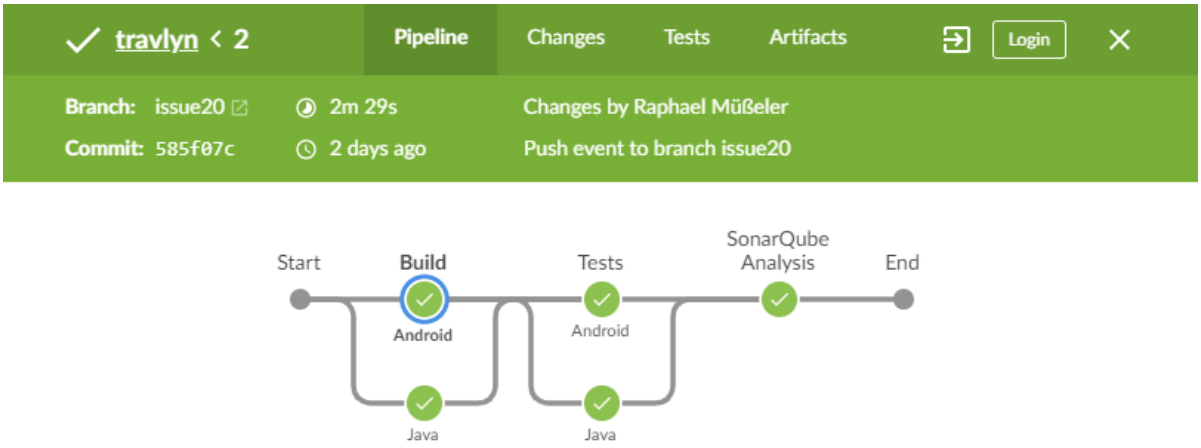

Abbildung 2.3: Darstellung der für das Projekt *Travlyn* erstellten Jenkins Pipeline

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 können aber ebenso gut Docker Images sein, mit deren Hilfe der Auslieferungsprozess simplifiziert wird. [**Humble.2011**]

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 [13]. In seinem Werk werden Anforderungen wie folgt definiert.

> **Anforderungen:** „Anforderungen (requirements) legen fest, was man von einem Softwaresystem als Eigenschaften erwartet“[13] (Seite 455). Mit der Annahme, dass „man“ alle Stakeholder (Personen, die ein Interesse an der Entwicklung und/oder der erstellten Software haben) beinhaltet. [13]

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 Vaiante gewählt [14]:
>
> - 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 festlegten 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 [13] (Seite 459).

3.2.1 Organisatorisches

Für *Travyln* 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 *Travyln* 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[10] läuft, welcher unabhängig von dem darunterliegendem 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 [13]. 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[13]. 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.

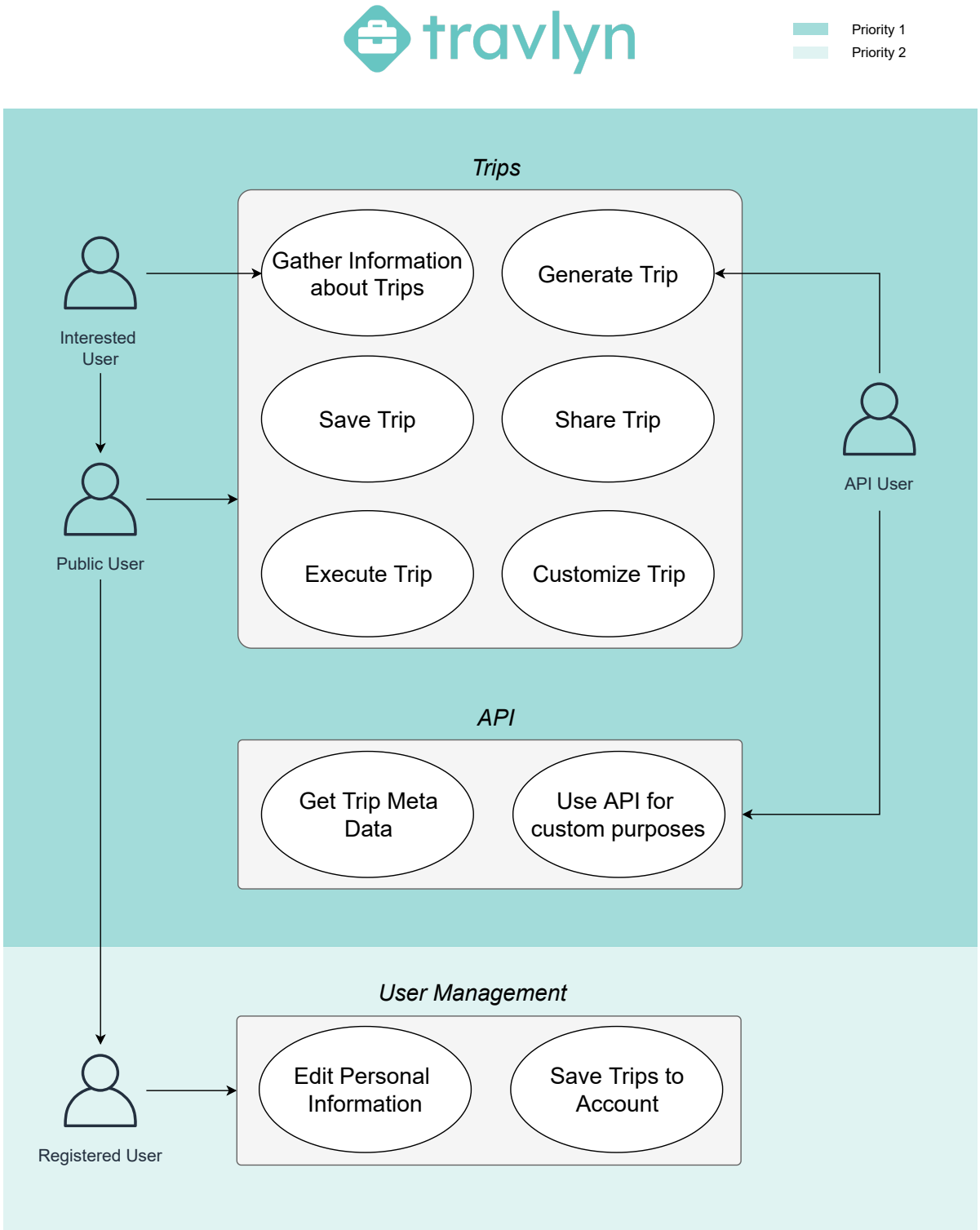

Abbildung 3.1: Darstellung aller geplanten Use Cases mit den assoziierten Nutzerprofilen.

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 *Travyln* 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 *Travyln* 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 [13].

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 [15] [16]:

- **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 Qualitiy 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 Datenmodel 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 *Travyln* 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 wir immer wichtiger und soll auch bei *Travyln* 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) [17] 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 *Travyln* 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 [18]. 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 [19]. Außerdem ist die Nutzung der erhaltenen Daten nur in Verbindung mit anderen von Google bereitgestellten Services erlaubt [20], somit wäre die ganze *Travlyn* Applikation an Google gebunden.

4.1.2 Openroute service

Openroute service [21] wird vom Heidelberger Institut für Geoinformationstechnik angeboten. Es handelt sich um eine Crowd Sourced API, d.h. sie wird durch Benutzer über OpenStreetMap (OSM) [22] 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 Crowdsouring 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 [23].

4.1.3 Foursquare

Die Firma Foursquare bietet ebenfalls eine API an [24], 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 werde, 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 [18]	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 [20]	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 einen 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 Inforationen 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 entscheiden, 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 beschreiben, die ausprobiert worden sind:

- **MediaWiki**: Hinter Wikipedia und vielen anderen Wiki-Seiten steht die selbe Software: MediaWiki [25]. 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 Protypes getestet wurde ist *DBpedia* [26]. DBpedia stellt die strukturierten Informationen aus Wikipedia in strukturierter Form zur Verfügung. Auch diese API folgt dem Crowd Souring 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.

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

Entiäten. 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 **??** 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.raphael-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.

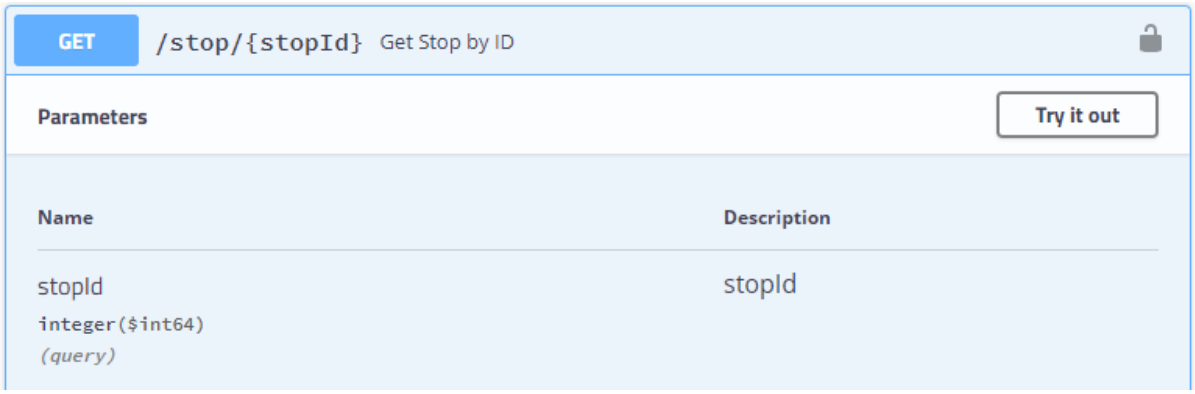

Abbildung 5.2: Beschreibung des *getStop* Requests im Swagger UI

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.

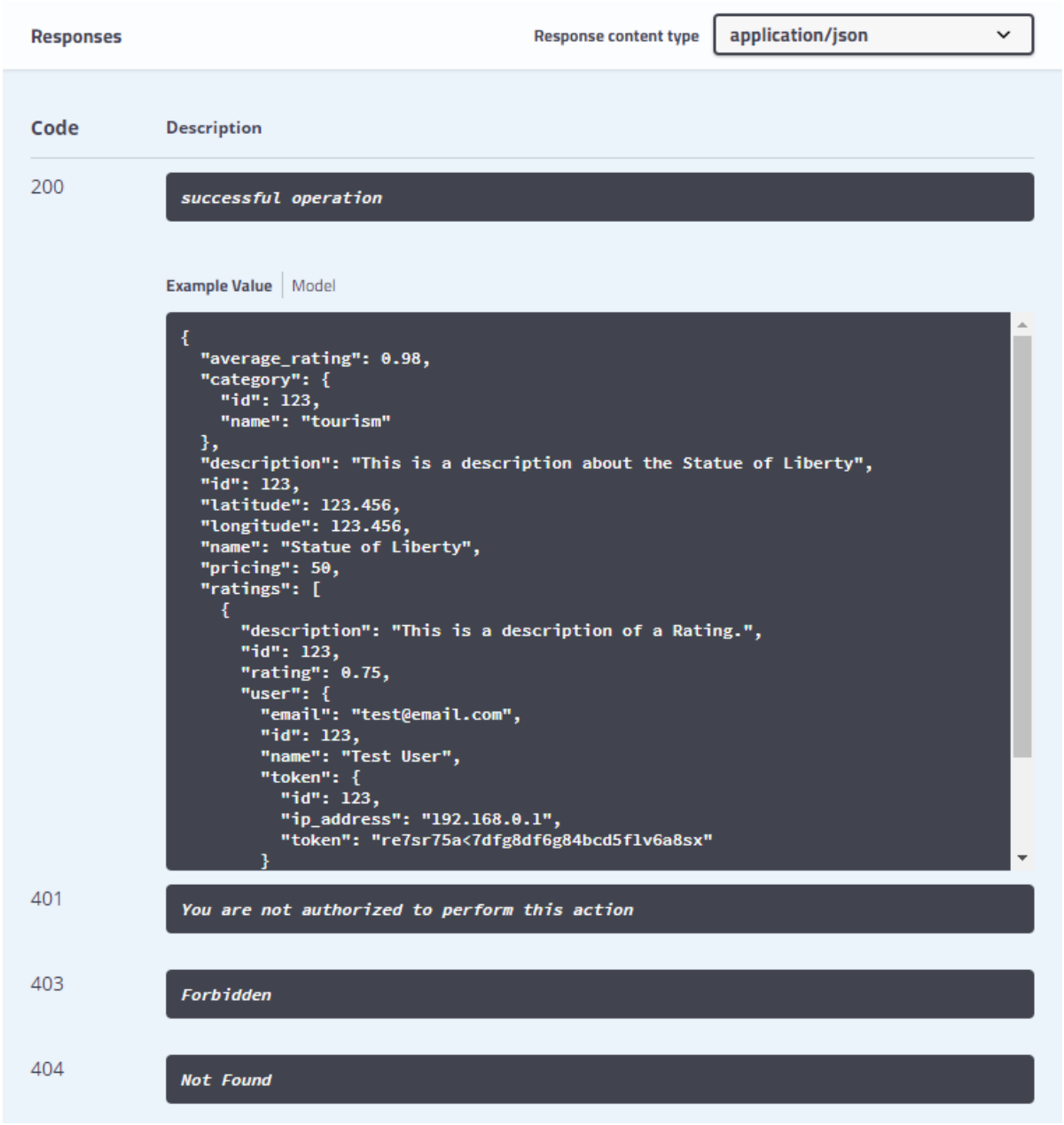

Abbildung 5.3: Beschreibung des *getStop* Requests im Swagger UI

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 Inforationen, 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 [27]. 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.

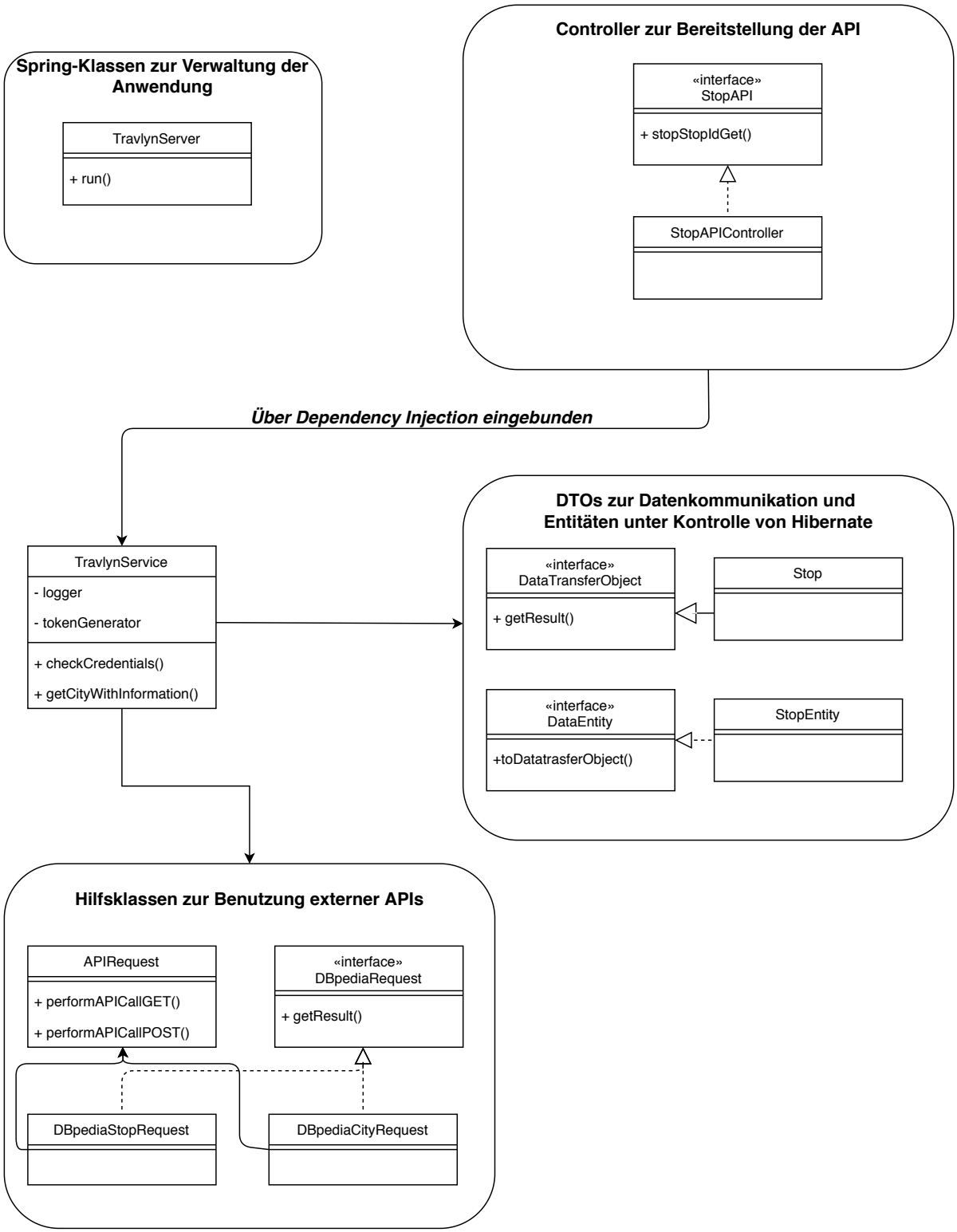

Abbildung 5.4: Struktur des *Travlyn* Servers als UML Diagramm. Zu Beachten ist, dass dieses Diagramm stark vereinfacht und nicht vollständig ist, um die Übersichtlichkeit zu wahren.

5.3 Client Architektur

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

6 Implementierung

6.1 Coding Conventions

6.2 Probleme

7 Evaluation

7.1 User Zufriedenheit

7.2 Aufbau

7.3 Ablauf

7.4 Ergebnis

8 Fazit

8.1 Ausblick

Literaturverzeichnis

[1] Oculus VR. *Oculus Rift S: VR Headset for VR Ready PCs | Oculus: Oculus Rift S: VR Headset for VR Ready PCs | Oculus*. 3/22/2020. URL: https://www.oculus.com/rift-s/.

[2] Dredge, D. u. a. „Digitalisation in Tourism: In-depth analysis of challenges and opportunities“. In: ().

[3] SmartBear. *The Best APIs are Built with Swagger Tools | Swagger*. 2020. URL: https://swagger.io/.

[4] 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.

[5] Johnson, R. *Expert one-on-one J2EE design and development*. Programmer to programmer. Indianapolis, Ind. und Great Britain: Wrox, 2003.

[6] Wunderlich, L. *AOP: Aspektorientierte Programmierung in der Praxis*. S- & -S-pockets. Frankfurt [Main]: Entwickler.press, 2005.

[7] Özsu, M. T./ Valduriez, P. *Principles of distributed database systems*. 3rd ed. New York: Springer Science+Business Media, 2011.

[8] 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.

[9] 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, Guadaloupe, France, 1 - 6 March 2006*. Hrsg. von Chen, Q. Piscataway, NJ: IEEE, 2009, S. 36–43.

[10] 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.B00LRROTI4.pdf.

[11] Docker Inc. *Docker Virtualisung verglichen mit virtuellen Maschinen*. 2020. URL: https://www.docker.com/sites/default/files/d8/2018-11/docker-containerized-and-vm-transparent-bg.png.

[12] JetBrains. *Kotlin on Android. Now official*. 2017. URL: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/.

[13] 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.

[14] 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.

[15] ISO. *ISO/IEC 25010:2011*. 2011. URL: https://www.iso.org/standard/35733.html.

[16] 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.

[17] 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.

[18] Google. *Geo-location APIs | Google Maps Platform | Google Cloud*. 01.02.2020. URL: https://cloud.google.com/maps-platform/.

[19] Singhal, M./ Shukla, A. „Implementation of Location based Services in Android using GPS and Web Services“. In: 9.1 (2012), S. 237.

[20] Google. *Google Maps APIs Terms of Service | Google Maps Platform*. 02.12.2019. URL: https://developers.google.com/maps/terms-20180207?hl=en.

[21] The Heidelberg Institute for Geoinformation Technology. *Openrouteservice*. URL: https://openrouteservice.org/.

[22] OpenStreetMap. *OpenStreetMap Deutschland: Die freie Wiki-Weltkarte*. URL: https://www.openstreetmap.de/.

[23] Winkler, P. *Computer-Lexikon 2010: Extra: Tipps zum persönlichen Datenschutz*. Vollst. überarb. Neuaufl. München: Markt+Technik Verl., 2009.

[24] Foursquare. *Foursquare Location Intelligence for Enterprise*. URL: https://enterprise.foursquare.com/products/places.

[25] MediaWiki. *MediaWiki*. 24.01.2020. URL: https://www.mediawiki.org/wiki/MediaWiki.

[26] DBpedia. *About | DBpedia*. 02.02.2020. URL: https://wiki.dbpedia.org/about.

[27] GOLL, J. *ENTWURFSPRINZIPIEN UND KONSTRUKTIONSKONZEPTE DER SOFTWARETECHNIK: Strategien fr schwach ... gekoppelte, korrekte und stabile software*. [Place of publication not identified]: MORGAN KAUFMANN, 2019.