

# Web-basierte Hochschul-App

## Modulare Web-Architektur

### **Praxisarbeit**

an der Hochschule für angewandte Wissenschaften Hof  
Fakultät Informatik  
Studiengang Informatik

#### **Vorgelegt bei**

Prof. Dr. Jürgen Heym  
Alfons-Goppel-Platz 1  
95028 Hof

#### **Vorgelegt von**

Dennis Brysiuk  
Richard-Wagner-Straße 64  
95030 Hof

Noah Christopher Lehmann  
Thüringer Str. 7  
95028 Hof

Hof, 29.01.2020

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Tabellenverzeichnis</b>	<b>V</b>
<b>Codeverzeichnis</b>	<b>VI</b>
<b>Abkürzungsverzeichnis</b>	<b>IX</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Beweggründe . . . . .	2
1.2 Zielsetzung . . . . .	3
1.3 Zielgruppe . . . . .	4
1.4 Vorgeschlagene Nebenlektüre . . . . .	5
<b>2 Vorgehensweise</b>	<b>6</b>
2.1 Grundkonzept . . . . .	6
2.2 Umsetzung . . . . .	6
<b>3 Technologien</b>	<b>11</b>
3.1 Programmiersprache . . . . .	11
3.2 Microservice Framework . . . . .	16
3.3 Build Management Tool . . . . .	21
3.4 Versionsverwaltung Tool . . . . .	22
3.5 API Gateway . . . . .	23
3.6 Service Registrierung und Discovery . . . . .	26
3.7 Benachrichtigung . . . . .	26
<b>4 Konfiguration</b>	<b>28</b>
4.1 Entwicklungsumgebung . . . . .	28
4.2 Spring Boot . . . . .	33
4.3 Spring Cloud Gateway . . . . .	47
4.4 Eureka . . . . .	49
4.5 API Filter . . . . .	51
4.6 Error Handling . . . . .	56
4.7 Firebase Cloud Message . . . . .	58

<b>5</b>	<b>Entwicklungsumgebungen</b>	<b>63</b>
5.1	Dev-Environment . . . . .	63
5.2	Prod-Environment . . . . .	68
<b>6</b>	<b>Microservice Dokumentation</b>	<b>71</b>
6.1	Konfiguration im Microservice . . . . .	71
6.2	Controller Beschreibung . . . . .	74
6.3	Dokumentation der Funktionen . . . . .	75
6.4	Export der Dokumentation . . . . .	76
6.5	Nutzen der grafischen Oberfläche . . . . .	76
<b>7</b>	<b>Testing</b>	<b>80</b>
7.1	Controller Tests . . . . .	81
7.2	Service Tests . . . . .	88
7.3	Persistenz Tests . . . . .	90
7.4	Funktionstests . . . . .	91
<b>8</b>	<b>Probleme</b>	<b>93</b>
8.1	Kontinuierliche Integration und Deployment . . . . .	93
8.2	Hypermedia . . . . .	95
8.3	Notification Service . . . . .	97
<b>9</b>	<b>Weiterentwicklung</b>	<b>98</b>
9.1	Pflege der Anwendung . . . . .	98
9.2	Programmcode Veröffentlichung . . . . .	101
9.3	Verteilung von Zugangsdaten . . . . .	103
9.4	Problemfindung und Beseitigung . . . . .	104
9.5	Neue Funktionen . . . . .	105
<b>10</b>	<b>Ausblick und Fazit</b>	<b>107</b>
10.1	Ausblick . . . . .	107
10.2	Fazit . . . . .	108
	<b>Anhang</b>	<b>110</b>
	Pflichtenheft . . . . .	110
	Autoren Referenz . . . . .	113
	<b>Literaturverzeichnis</b>	<b>115</b>

**Eidesstattliche Erklärung**

**118**

# Abbildungsverzeichnis

2.1	Agiles Trello Board . . . . .	7
2.2	Trello Board Backlog . . . . .	8
2.3	Trello Board Sprint Backlog . . . . .	8
2.4	Trello Board Entwickler Todo Listen . . . . .	9
2.5	Trello Board Liste der fertigen Aufgaben . . . . .	10
3.1	Hochschull-APP Architektur ohne Gateway . . . . .	24
3.2	Hochschul-APP Architektur mit Gateway . . . . .	25
4.1	Hot Deployment in <i>IntelliJ</i> . . . . .	29
4.2	Öffnen der Registry in <i>IntelliJ</i> . . . . .	30
4.3	Compiler Anweisung in <i>IntelliJ</i> . . . . .	30
4.4	Start des Services in <i>IntelliJ</i> . . . . .	30
4.5	Neustart des Services in <i>IntelliJ</i> . . . . .	31
4.6	Lombok Plugin in <i>IntelliJ</i> . . . . .	33
4.7	Projektwahl in <i>Spring Initializr</i> . . . . .	34
4.8	Programmiersprachenwahl in <i>Spring Initializr</i> . . . . .	34
4.9	Versionswahl in <i>Spring Initializr</i> . . . . .	34
4.10	Meta Daten in <i>Spring Initializr</i> . . . . .	35
4.11	Wahl der Abhängigkeiten in <i>Spring Initializr</i> . . . . .	36
4.12	Firebase Webkonfiguration . . . . .	58
4.13	Firebase Admin SDK Authentifizierung . . . . .	59
5.1	Login der <i>H2</i> -Console . . . . .	65
5.2	Start und Debuggen der Anwendung in der Integrated Development Environment (IDE) . . . . .	66
5.3	Einstellen der Test-Umgebung in der IDE . . . . .	67
6.1	Swagger UI am Beispiel des Timetable-Service . . . . .	77
6.2	Swagger UI Timetable Methoden . . . . .	77
6.3	Swagger UI Timetable Methoden Details . . . . .	78
6.4	Testen einer Anfrage in der Swagger UI . . . . .	79

# Tabellenverzeichnis

7.1	Validation Rules . . . . .	85
X.1	Pflichtenheft . . . . .	110
X.2	Pflichtenheft . . . . .	111
X.3	Pflichtenheft . . . . .	112
X.4	Autoren Referenz . . . . .	113
X.5	Autoren Referenz . . . . .	114

# Codeverzeichnis

4.1	Klassendefinition mit <i>Lombok</i> . . . . .	32
4.2	<i>Spring Web</i> Dependency . . . . .	36
4.3	Spring Data JPA Dependency . . . . .	36
4.4	H2 Dependency . . . . .	37
4.5	<i>MySQL</i> JDBC . . . . .	37
4.6	<i>Spring DevTool</i> Dependency . . . . .	37
4.7	Hypermedia As The Engine Of Application State (HATEOAS) De- pendency . . . . .	38
4.8	<i>Eureka</i> Service Discovery Client Dependency . . . . .	38
4.9	Lombok Dependency . . . . .	39
4.10	<i>Swagger</i> Dependency . . . . .	39
4.11	<i>Swagger-UI</i> Dependency . . . . .	39
4.12	Test Dependencies . . . . .	40
4.13	Eureka Server Dependencies . . . . .	40
4.14	Spring Cloud Gateway Dependencies . . . . .	41
4.15	FCM Dependencies . . . . .	41
4.16	Spring Application Klasse . . . . .	42
4.17	Spring Controller Klasse . . . . .	43
4.18	Spring Service Klasse . . . . .	43
4.19	Spring Entity Klasse . . . . .	44
4.20	Typische <i>application.properties</i> Datei . . . . .	46
4.21	Gateway <i>application.yml</i> Datei . . . . .	47
4.22	Gateway Routen Konfiguration Klasse . . . . .	48
4.23	CORS-Filter Konfigurationsklasse . . . . .	49
4.24	Spring Name in <i>application.properties</i> Datei . . . . .	50
4.25	Eureka Client Einstellungen aus <i>application.properties</i> Datei . . . . .	50
4.26	Service als Eureka Server kennzeichnen . . . . .	51
4.27	Typischer Anfragepfad mit Filtern . . . . .	52
4.28	Typischer Endpunkt mit @RequestParam . . . . .	52
4.29	Aufbau des Filter Objekts . . . . .	52
4.30	Filterparser . . . . .	53
4.31	Typischer Endpunkt mit eigener Lösung . . . . .	53

---

4.32	Ressourcenspezifisches Filter Objekt . . . . .	54
4.33	Serviceschicht . . . . .	55
4.34	Spezifikation der Filter für Datenbankanfrage . . . . .	55
4.35	Error Handling . . . . .	57
4.36	Firebase Service Account JSON . . . . .	60
4.37	Firebase Service Account application.properties Datei . . . . .	60
4.38	Firebase Service Account Variable . . . . .	60
4.39	Firebase Service Account Bean . . . . .	61
4.40	Firebase Service Account Bean . . . . .	61
4.41	Firebase Service Account Bean . . . . .	62
5.1	H2-Console . . . . .	64
5.2	Discovery und Gateway Deaktivierung . . . . .	66
5.3	Starten eines <i>jar</i> -Files . . . . .	68
5.4	Skript zum Verwalten der Microservices . . . . .	69
6.1	Einbinden der <i>Spring-Fox</i> -Abhängigkeiten . . . . .	71
6.2	Bean zur Swagger Konfiguration im Mensa-Service . . . . .	72
6.3	Swagger Konfiguration auf Controller Ebene am Beispiel des Time- table Controllers . . . . .	74
6.4	Swagger Konfiguration auf Methoden Ebene am Beispiel des Se- mester Controllers . . . . .	75
6.5	Swagger Export URL . . . . .	76
6.6	Swagger Export URL . . . . .	76
7.1	Basis Initialisierung für Controller Tests . . . . .	82
7.2	Gültige Anfrage Test . . . . .	83
7.3	Falscher Typ Test . . . . .	83
7.4	Fehlende Parameter Test . . . . .	84
7.5	Validierung Aktivieren . . . . .	86
7.6	Validierung einer PathVariable . . . . .	86
7.7	Validierung von RequestBody . . . . .	87
7.8	Parameter Regeln Test . . . . .	87
7.9	Verifizierung der Business Logik . . . . .	88
7.10	Basis Initialisierung für Service Tests . . . . .	89
7.11	Testen von richtigen Verhalten . . . . .	89
7.12	Testen von Fehlverhalten . . . . .	90
7.13	Testen von Fehlverhalten . . . . .	90



7.14	Testen von Fehlverhalten . . . . .	91
7.15	Testen von Fehlverhalten . . . . .	91
7.16	Testen von Fehlverhalten . . . . .	92
8.1	PluginRegistry Bean Konflikt . . . . .	96

# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>App</b>	Applikation
<b>CD</b>	Continuous Deployment
<b>CI</b>	Continuous Integration
<b>CORS</b>	Cross-Origin Resource Sharing
<b>DAO</b>	Data Access Object
<b>DBMS</b>	Database Management System
<b>DDoS</b>	Distributed Denial of Service
<b>DoS</b>	Denial of Service
<b>EE</b>	Enterprise Edition
<b>FCM</b>	Firebase Cloud Messaging
<b>HATEOAS</b>	Hypermedia As The Engine Of Application State
<b>HTML</b>	Hypertext Markup Language
<b>HTTPS</b>	Hypertext Transfer Protocol (Secure)
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IDE</b>	Integrated Development Environment
<b>JAX-RS</b>	Java Application Programming Interface (API) for Representational State Transfer (REST)ful Web Services
<b>JDBC</b>	Java Database Connectivity
<b>JPA</b>	Java Persistence API
<b>JS</b>	JavaScript
<b>JSON</b>	Javascript Object Notation

<b>JVM</b>	Java Virtual Machine
<b>kB</b>	Kilobyte
<b>LOC</b>	Line of Code
<b>OSS</b>	Open Source Software Center
<b>PC</b>	Personal Computer
<b>PHP</b>	Hypertext Pre-Processor
<b>POJO</b>	Plain Old Java Object
<b>PYPL</b>	Popularity of Programming Language Index
<b>REST</b>	Representational State Transfer
<b>SDK</b>	Software Development Kit
<b>SQL</b>	Structured Query Language
<b>URL</b>	Uniform Resource Locator
<b>XML</b>	Extensible Markup Language
<b>YAML</b>	YAML Ain't [sic]! Markup Language (ursprünglich <i>Yet Another Markup Language</i> )

# 1 Einleitung

Um ein Projekt sinnvoll durchführen zu können und um das nach der Vollendung entstandene Ergebnis auch weiter pflegen und entwickeln zu können, bedarf es einer strukturierten Dokumentation zur Vorgehensweise im Verlauf des Projekts. Hierbei müssen jedoch einige Dinge beachtet werden: sind die Projektplanungen bereits abgeschlossen, steht schon die Grundstruktur des Projekts und was genau muss eigentlich dokumentiert werden?

All das soll in diesem Kapitel kurz erläutert werden. Das Projekt, um das es sich handelt, ist der Prototyp der neuen web-basierten Hochschul-Applikation (App). Diese wird im Verlaufe dieser Arbeit parallel entwickelt, weshalb die Arbeit diesem Prototypen als Dokumentation und Rechtfertigung der genutzten Techniken dienen soll. Die folgenden Kapitel sollen nun darauf eingehen, weshalb solch eine Dokumentation von Nöten ist, was in dieser Dokumentation alles abgehandelt werden soll, welche Leser-Gruppen schlussendlich von ihr profitieren können und welche Grundkenntnisse und Nebenlektüre zum Lesen dieser Arbeit sinnvoll sind.

Im Anschluss soll kurz auf die Vorgehensweise bei der Erstellung des Prototypen eingegangen werden. Hierbei ist keine genaue Projektplanung von Nöten, stattdessen sollen nur kurz die Konzepte vorgestellt werden, nach denen gearbeitet wird. Sollten gewisse Techniken genannt werden, so werden diese auch später in der Arbeit genauer erläutert. Auch die hier nicht genannten Technologien sollen aber noch genauer erläutert werden, denn nur so kann ein volles Verständnis dafür entwickelt werden, weshalb gewissen Technologien ihren Alternativen vorgezogen wurden.

Nachdem ein allgemeines Verständnis der Technologien vermittelt wurde, soll nun genauer auf die Implementierung der neuen Hochschul-App eingegangen werden. Hierbei werden nun technische Details bereitgestellt, die vor allem neuen Entwicklern einen leichten Einstieg in die Implementierung bieten sollen. Um diesen Entwicklern noch weitere Werkzeuge an die Hand zu geben wird kurz auf die empfohlene Entwicklungsumgebung eingegangen. Hierbei soll vor allem beleuchtet wer-

den, welche nützlichen Tools eingerichtet wurden, um die Entwicklungsarbeit an dem Prototypen stark zu vereinfachen.

Anschließend kann nun die vorhandene Implementierung genauer betrachtet werden. Hierbei soll weniger auf den detaillierten Programmcode, sondern eher auf die Bereitstellung der Ressourcen des REST-Services geachtet werden. Es werden alle Microservices und ihre Feinheiten betrachtet, sodass am Ende ein klares Verständnis über den Aufbau der Datenschnittstelle vermittelt wurde. Neben der Dokumentation der Services fallen natürlich auch die Tests zu genau dieser an. Diese sind besonders wichtig und dringend zu dokumentieren, denn an den bereitgestellten Tests können sich spätere Entwickler richten, wenn sie neue Funktionen implementieren. So kann klargestellt werden, dass die Qualität der neuen Hochschul-App konstant bleibt, auch wenn sich das Entwicklerteam über längere Zeit verändert.

Zum Abschluss sollen noch die Probleme bei der Implementierung genauer betrachtet werden. Dabei soll es um nicht umgesetzte oder nicht umsetzbare Funktionen gehen. Das soll vor allem den späteren Verlauf der Entwicklung unterstützen, denn hier werden auch allgemeine Probleme festgehalten. Nachdem die Probleme festgehalten wurden soll noch die spätere Weiterentwicklung geplant werden. Hierbei werden Werkzeuge und Konzepte bereitgestellt, die eine Zusammenarbeit von vielen Entwicklern und auch interessierten Dritten ermöglicht. Abschließend daran soll das Gesamtprojekt noch einmal reflektiert werden.

## **1.1 Beweggründe**

Bei dem initialen Projekt zur Neuaufsetzung der Hochschul-App als web-basierte Anwendung ist viel Aufwand in die Analyse der Möglichkeiten und der Zusammenstellung der genutzten Techniken geflossen. Es wurden Umfragen gestartet, Anforderungen aus verschiedenen Quellen gesammelt und anschließend, aufbauend auf den gewonnenen Erkenntnissen, eine aufwändige Architektur gestaltet. Um diese Architektur schließlich sinnvoll umsetzen zu können wurden einige Techniken und Frameworks analysiert, welche einer Erklärung bedürfen. Die genannten Schritte bei der Analysearbeit wurden in der zu dieser Arbeit parallel erstellten Bachelorarbeit dokumentiert.<sup>1</sup>

---

<sup>1</sup>Brysiuk; Lehmann (2019a).

Jedoch reicht bei der Erstellung eines Prototypen die reine Analyse und die Sammlung der Anforderungen nicht aus. Am Ende muss auch ein lauffähiges Programm entstehen, welches dann ausgiebig getestet und anschließend weiterentwickelt werden kann. Auch aus den im Entwicklungsprozess einer Anwendung gewonnenen Erkenntnissen lassen sich im Nachhinein hilfreiche Schlüsse ziehen, weshalb bei der Durchführung eines Projektes immer eine Praxisarbeit zum Dokumentieren des Vorgehens angefertigt werden sollte. Genau dazu dient diese Arbeit. Sie soll den Entwicklungsprozess des Projektes der *web-basierten Hochschul-App* aufzeichnen und die verwendeten Techniken, Prinzipien, Frameworks aber auch Vorgehensweisen festhalten, damit spätere Entwicklerteams einen leichten Einblick in die Ideen und Lösungen der Entwickler des Prototypen der neuen Hochschul-App erhalten können.

## 1.2 Zielsetzung

Wie bereits erwähnt soll diese Arbeit als begleitendes Handbuch zur Implementierung des Prototypen der *web-basierten Hochschul-App* dienen. Das bedeutet nicht nur, dass hier die genutzten Techniken genauer erläutert werden, sondern auch die Herangehensweise an das Projekt, die Schwierigkeiten, die die Implementierung mit sich gebracht hat und die Vorkehrungen, die für spätere Entwicklerteams getroffen wurden.

Im Laufe dieser Arbeit soll die Vorgehensweise, mit der die Anwendung umgesetzt wurde, klar dargestellt werden. Das beinhaltet nicht nur die reine Vorgehensweise, mit der gearbeitet wurde, sondern auch die Organisation zwischen den Entwicklern untereinander. Des Weiteren sollen die genutzten Technologien genau erläutert werden. Hier soll vor allem auch darauf eingegangen werden, weshalb die Technologien ihren jeweiligen Alternativen vorgezogen wurden und wie die Technologien genau in den Prototypen der Hochschul-App eingeflossen sind. Sobald die Technologien dann wiederum ausreichend erklärt wurden soll die eigentliche Implementierung der web-basierten Hochschul-App genauer betrachtet werden. Mit dem gewonnenen Grundwissen aus den erklärten Technologien soll dem Leser nun die Konfiguration der einzelnen Komponenten deutlich gemacht werden. Dieser Teil der Arbeit ist eher Quellcode-lastig und deshalb vor allem für spätere Entwicklerteams interessant.

Ebenso für die Entwicklerteams sinnvoll zu lesen sind die darauf folgenden Kapitel. Sie handeln von den Umgebungen, die den Entwicklern beim Programmieren und Testen zur Verfügung stehen. Hier anzumerken ist, dass bei der Entwicklung des Prototypen viel Wert darauf gelegt wurde, dass später schnell neue Entwickler in das Projekt eingegliedert werden können. Auch die exakte Dokumentation der Endpunkte der einzelnen Services der Hochschul-App-Datenbereitstellung ist von besonderem Interesse für Entwickler. Denn nach dieser Dokumentation können sich vor allem auch Frontend Programmierer richten, die lediglich die Datenbereitstellung nutzen, um eine eigene Anwendung zu erstellen. Anschließend an die eigentliche Dokumentation wird kurz auf das Testverfahren für das Backend der Hochschul-App eingegangen.

Abschließend sollen die Leser dieser Arbeit noch erkennen, welche Vorkehrungen getroffen wurden, um die App nach Abschluss des Prototypen weiterentwickeln zu können. Dabei werden einige Punkte angesprochen, die die Einbindung von neuen Entwicklern unterstützen und vor allem auch die Verbesserung der Anwendung im Allgemeinen vorantreiben sollen. Dieser Teil soll dann auch als Abschluss der Arbeit dienen und in einem kurzen Fazit zur Durchführung des Projektes enden.

## **1.3 Zielgruppe**

Wie bereits mehrfach erwähnt wurde richtet sich diese Arbeit zu großen Teilen den Entwicklern, die später weiter an dem Projekt der neuen Hochschul-App arbeiten werden. Diese werden in dieser Dokumentation zum Prototypen einige Erklärungen zu verwendeten Techniken, Anleitungen zur Verwendung gewisser Technologien und Richtlinien zum Arbeiten am Programmcode finden. Wie bereits zu vermuten ist, sollten diese Entwickler bereits ein fundiertes Grundwissen in den Bereichen der Webtechnologien und in der Software Entwicklung im Allgemeinen vorweisen können.

Jedoch ist diese Praxisarbeit nicht nur an die Entwickler der neuen Hochschul-App gerichtet, sondern auch an alle Beteiligten, die ein Interesse am Aufbau und den benötigten Ressourcen dieser Anwendung haben. So wird der Einsatz gewisser Technologien gerechtfertigt, was spätere Entscheidungen bei Änderungen der Lizenzbedingungen dieser Technologien erleichtern soll. Des weiteren werden für solche Fälle stets Alternativen genannt. Die Teile der Arbeit, die für außenstehen-

de, nicht-Entwickler geeignet sind, sind auch stets in abgeschwächter Fachsprache verfasst, sodass jeder sie verstehen kann. Für diese Teile soll kein fundiertes Grundwissen aus der Informatik von Nöten sein.

## 1.4 Vorgeschlagene Nebenlektüre

Da es sich bei dieser Arbeit um eine Praxisarbeit handelt, die parallel zu Analyse und Implementierung des Prototypen der neuen, web-basierten Hochschul-App angefertigt wurde, bietet es sich zu aller erst an, die weiteren Arbeiten zu lesen, die im Zuge dieses Projektes erstellt wurden. Allen voran ist hierbei die Analyse zur Anforderungssammlung, der Architektur und der REST-Schnittstelle zu erwähnen, welche in der Bachelorarbeit *Web-basierte Hochschul-App - modulare Web-Architektur* ausführlich behandelt wurde.<sup>2</sup>

Da es sich bei dieser Arbeit in den technischen Abschnitten lediglich um die Dokumentation des Backends, also des Server-seitigen Codes handelt, ist es wichtig ebenfalls zu verstehen, wie die Studierenden der Hochschule Hof, also die primäre Nutzergruppe, von den Ergebnissen dieser Arbeit profitieren können. Das geschieht im Prototyp der Hochschul-App durch eine web-basierte Nutzeroberfläche, die mit Hilfe des Typescript-Frameworks *Angular* angefertigt wurde. Auch hierzu wurden umfangreiche Analysen zur Erstellung dieser und zur Kommunikation mit dem Backend erstellt, welche besonders auf die Sicherheitskonzepte dieses Prototypen eingehen. Die Analyse dieser Arbeit wurde in der Bachelorarbeit *Web-basierte Hochschul-App - Authentifizierung und Personalisierung* betrieben, die Dokumentation zur Implementierung in der gleichnamigen Praxisarbeit.<sup>3</sup>

---

<sup>2</sup>Brysiuk; Lehmann (2019a).

<sup>3</sup>Glaser (2019a); Glaser (2019b).



## 2 Vorgehensweise

Wie jedes Projekt benötigt auch die Erstellung des Prototypen der neuen Hochschul-App eine Vorgehensweise, nach der das Entwicklerteam geordnet arbeiten kann. Hierbei stellt sich vor allem die Frage, welches Vorgehen bei einer Teamgröße von zwei festen Entwicklern und einem Auftraggeber sinnvoll anzuwenden ist. Deshalb wurde im Rahmen dieses Projektes eine Abwandlung des bekannten *Scrum*-Vorgehensmodells angewendet. Die genaue Umsetzung dieser Abwandlung soll im folgenden nur kurz angedeutet werden, um einen Einblick zu verschaffen, wie die Ergebnisse aus dem Rest der Arbeit erstellt wurden.

### 2.1 Grundkonzept

Die Idee hinter *Scrum* liegt darin, eine flexible Zusammenarbeit zwischen Kunden und Entwicklern zu ermöglichen. Dabei muss am Anfang eines jeden Projektes eine Sammlung aus Anforderungen in technische Aufgaben umgewandelt werden. Diese Sammlung heißt dann *Backlog*. Danach werden Entwicklungsphasen in sogenannte *Sprints* aufgeteilt, die nicht länger als zwei bis drei Wochen andauern sollten. In diesen Sprints werden Aufgaben aus diesem *Backlog* übernommen. In der Regel sind die Aufgaben untereinander priorisiert, weshalb die ersten Sprints auch die wichtigsten Funktionen implementieren sollten. Während der Sprints können die Aufgaben im *Refinement* nochmals angepasst werden und Fragen an den Auftraggeber können hier nochmals adressiert werden. Am Ende eines jeden Sprints werden die Ergebnisse im sogenannten *Sprint Review* nochmals evaluiert und dann als abgeschlossen markiert, worauf im nächsten Sprint neue Aufgaben aufgenommen werden können.

### 2.2 Umsetzung

Die Anpassung dieses Konzepts liegt grundlegend darin, dass die Rolle des Auftraggebers hier nicht vom eigentlichen Auftraggeber gespielt wurde, sondern vom

Entwicklerteam als Ganzes. Immer wenn Entscheidungen getroffen werden mussten, mussten beide Entwickler zu einer gemeinsamen Antwort kommen. Die Sprint Refinements konnten deutlich vereinfacht werden, da lediglich die Kommunikation zwischen zwei Entwicklern nötig war und die Sprint Reviews so am Ende jeden Sprints als kurzes Treffen abgehalten werden konnte. Im folgenden sollen die wichtigsten Phasen und Aufgaben aus dem Vorgehen kurz erläutert werden. Für die Planung und Dokumentation der Sprints wurde die Online-Plattform *Trello* genutzt, die das Erstellen von agilen Sprint-Boards sehr einfach gestaltet.<sup>1</sup> Das *Scrum Board*, das dabei erstellt wurde, sieht wie folgt aus:

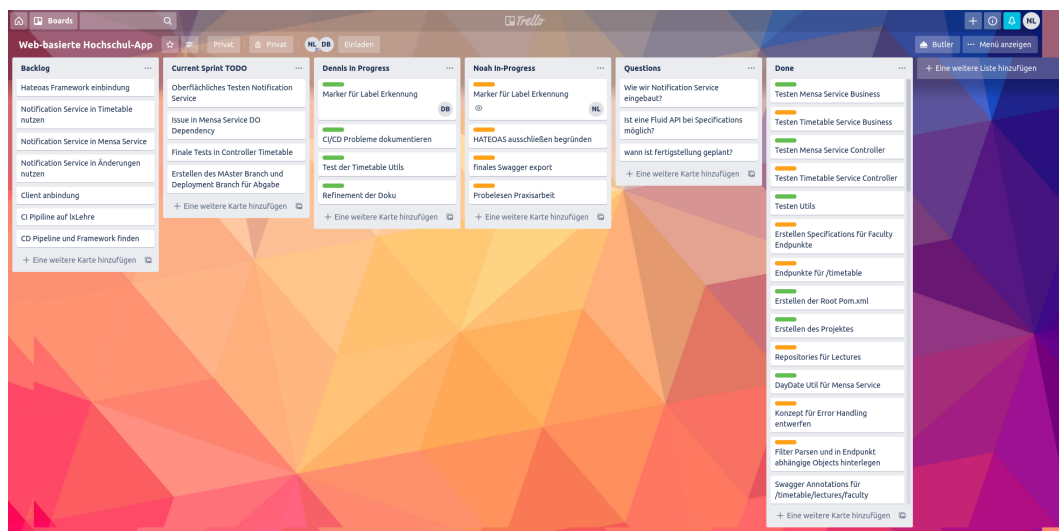


Abbildung 2.1: Agiles Trello Board

## Sammeln der Anforderungen

In der parallel zu dieser Arbeit erstellten Bachelorarbeit wurde bereits ein umfassendes Lastenheft erstellt, welches die funktionalen Anforderungen sammelt und kategorisiert. Anhand dieses Lastenhefts wurde ein Pflichtenheft<sup>2</sup> erstellt, aus dem dann die technischen Aufgaben abgeleitet wurden, welche dann in das Projekt-übergreifende Backlog übernommen werden konnten. Dieses sieht im genutzten *Trello*-Board folgendermaßen aus:

<sup>1</sup>Trello (2020).

<sup>2</sup>Siehe Anhang X.3

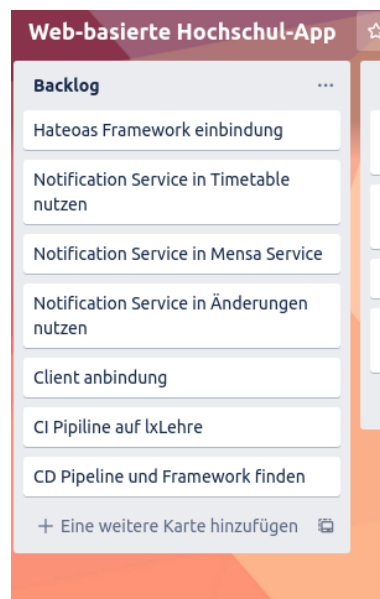


Abbildung 2.2: Trello Board Backlog

## Planung eines Sprints

Am Anfang eines jeden Sprints müssen die wichtigsten Aufgaben aus dem Backlog in das Sprint Backlog übernommen werden. Hier ist stets zu beachten, dass dabei möglichst nicht mehr Aufgaben übernommen werden, als im Sprint-Zeitraum erledigt werden können.

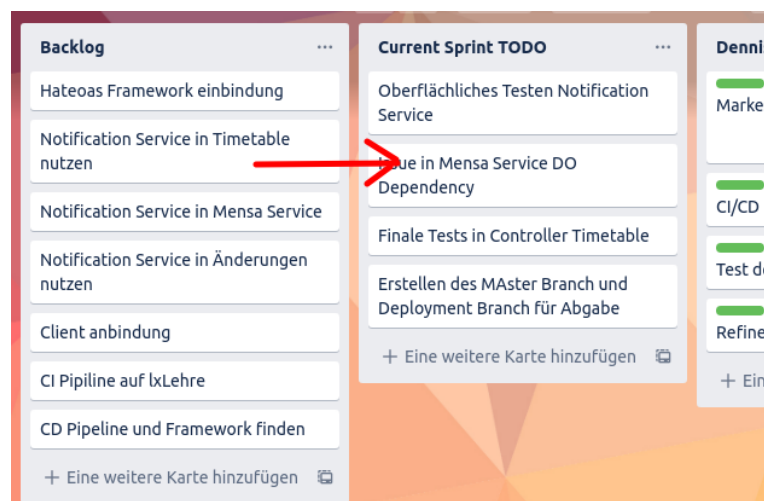


Abbildung 2.3: Trello Board Sprint Backlog

Innerhalb des Sprint Backlog sollte nun die Aufgaben nach ihrem Zeitaufwand bewertet werden. Die Entwickler entnehmen dem Sprint Backlog die Aufgaben ge-

ordnet nach der Priorisierung, die wichtigsten Aufgaben müssen dabei immer zuerst erledigt werden. Die Aufgaben, die am Ende eines Sprints noch nicht aus dem Sprint Backlog entnommen wurden müssen in den nächsten Sprint übernommen werden.

## Verteilen der Aufgaben

Wie bereits erwähnt entnehmen die Entwickler, sobald sie Zeit zur Verfügung haben, eine Aufgabe aus dem Sprint Backlog und hinterlegen sie in ihrer *Todo*-Liste. Von dort aus sollten die Aufgaben mit dem entsprechenden Label versehen werden, das anzeigt, wem die Aufgabe initial zugeordnet wurde. So ist am Ende auch zu erkennen, wer welche Aufgaben erledigt hat.

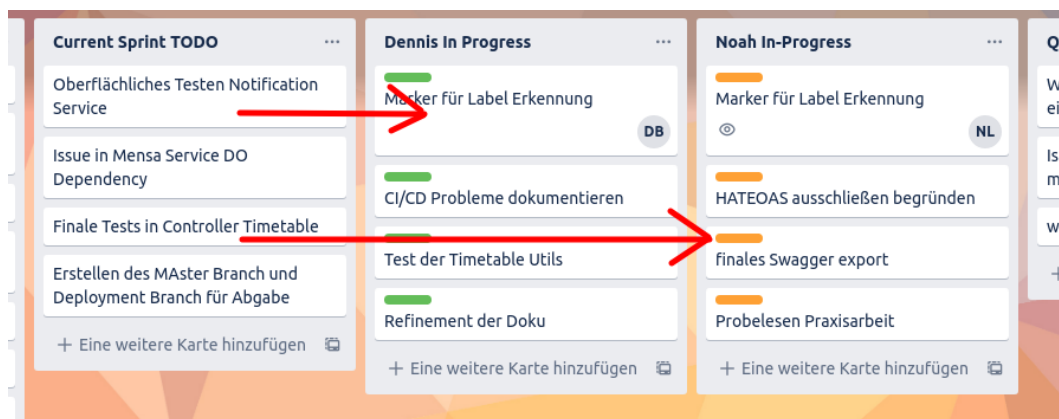


Abbildung 2.4: Trello Board Entwickler Todo Listen

Anders als im Sprint Backlog ist hierbei darauf zu achten, dass am Ende eines Sprints keine Aufgaben mehr in den Todo-Listen der Entwickler übrig sind. Diese führen sonst auf Dauer nur dazu, dass der Überblick über das Board verloren geht, was das Trello Board im Allgemeinen überflüssig machen würde. Es können allerdings jederzeit mehrere Aufgaben übernommen werden, insofern es sinnvoll ist, diese parallel zu bearbeiten.

## Abschluss von Aufgaben

Sobald eine der Aufgaben erledigt und getestet ist, können die Entwickler sie in die Spalte *Done* übernehmen, in welcher die Aufgaben als endgültig erledigt markiert werden.

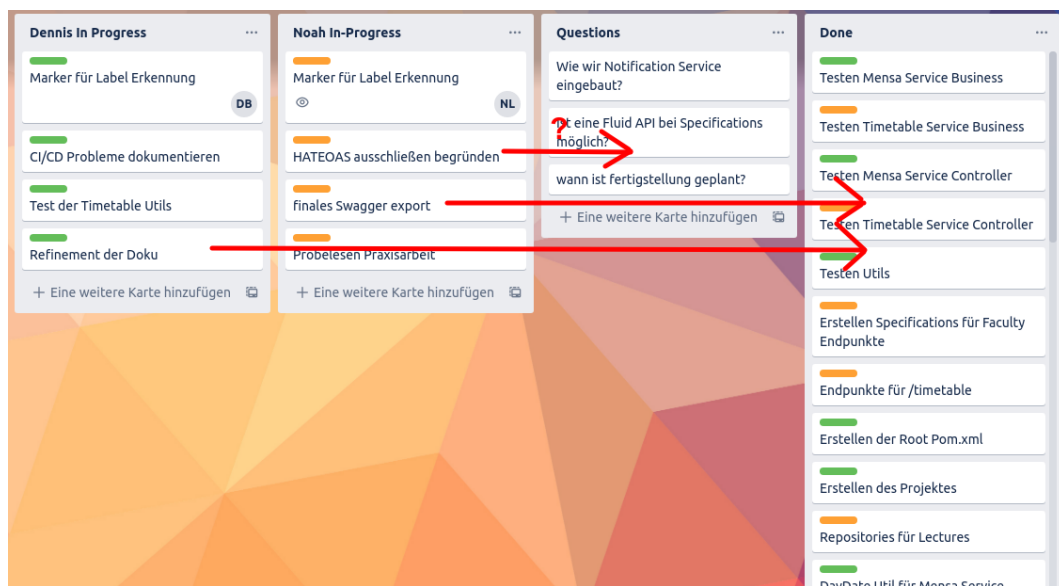


Abbildung 2.5: Trello Board Liste der fertigen Aufgaben

Sollten während der Bearbeitung jedoch noch Fragen auftreten, welche die Weiterarbeit oder den Abschluss der Aufgabe verhindern, so können die Fragen als einzelne Karte in die Spalte *Questions* erstellt werden. Wenn die Weiterarbeit an einer Aufgabe bis zur Beantwortung der Frage nicht möglich ist, so kann man die Aufgabe auch als Ganzes in die Spalte *Questions* verschieben.

Abschließend ist zur Organisation des Projektes zu sagen, dass viele Ansprüche und Konzepte, die *Scrum* mit sich bringt, in einem Team von zwei Entwicklern nicht sinnvoll sind. Dennoch sollte die Grundidee aufgenommen werden, um eine Struktur in der Entwicklung beizubehalten und um wichtige Aufgaben nicht zu vergessen. Dies ist durch die abgewandelte Nutzung des *Scrum*-Konzepts gelungen.

## 3 Technologien

Es gibt verschiedene Möglichkeiten Microservices zu implementieren. Möchte man diese erschließen, sollte man mindestens zwischen vier Ebenen in der Softwarehierarchie unterscheiden. Diese sind die genutzte Programmiersprache, die verwendeten Frameworks, der Aufbau der Architektur und die genutzte Protokolle in der Kommunikation mit dem Client. Da das Service-Orientated-Design, die Schichtenarchitektur und die Nutzung von REST in Verbindung mit Hypertext Transfer Protocol (HTTP) bereits in der parallel zu dieser Arbeit geschriebenen Bachelorarbeit festgelegt wurden, wird in dieser Arbeit nur noch auf die Programmiersprache und die genutzten Frameworks eingegangen.<sup>1</sup> Zusätzlich zu der verwendeten Programmiersprache können auch unterschiedliche Bibliotheks- und Quellcodeverwaltungstools zur Unterstützung genutzt werden. Die bei der Implementierung des Prototypen der neuen Hochschul-App verwendeten Programmiersprachen, Frameworks und Hilfstools werden im folgenden genauer betrachtet.

### 3.1 Programmiersprache

Bei der Wahl der Programmiersprache gilt es im Allgemeinen auf folgende Punkte zu achten:

- Allgemeine Beliebtheit
- Nutzbarkeit der Sprache
- Zuverlässigkeit der Sprache
- Eignung für Anwendung
- Programmiererfahrung in der Sprache

Da der Prototyp der Hochschul-App in der Zukunft von Studierenden gepflegt und weiterentwickelt wird, werden für die Auswahl der Programmiersprache nur drei

---

<sup>1</sup>Siehe Brysiuk; Lehmann (2019a).

Möglichkeiten in Betracht gezogen, die in den Vorlesungen der Fakultät Informatik an der Hochschule auch gelehrt werden. Diese sind JavaScript (JS), Hypertext Pre-Processor (PHP) und Java.

## Kandidaten

Für ein besseres Verständnis der folgenden Bewertungen werden die drei Programmiersprachen kurz vorgestellt.

### JavaScript

Der Autor Christian Wenz beschreibt die anfängliche Intention der Programmiersprache JavaScript wie folgt:

Mit JavaScript kann man die eher beschränkten Möglichkeiten von HTML erweitern. Es handelt sich bei JavaScript um eine clientseitige Programmiersprache. Das heißt, alles läuft im Browser ab, und man muss keine besonderen Server-Voraussetzungen erfüllen.<sup>2</sup>

Ursprünglich wurde JS 1995 von Netscape für deren Internet Browser in der Version 2.0 unter dem Namen LiveScript entwickelt. LiveScript war eine Programmiersprache, die dafür gedacht war, auf Hypertext Markup Language (HTML)-Seiten Inhalte dynamisch zu manipulieren. Da die Syntax stark an die der Programmiersprache Java angelehnt war, wurde LiveScript später aus Marketinggründen zu JavaScript umbenannt, jedoch hat die Sprache bis auf die syntaktischen Ähnlichkeiten keine weiteren Gemeinsamkeiten mit Java<sup>3</sup>.

Heute kann man mit JS dank Frameworks wie *Node.js* oder *Angular* weitaus komplexere Anwendungen schaffen. So kann man nicht nur dynamisch HTML-Inhalte manipulieren, sondern auch Contentmanagement betreiben oder ganze Serveranwendungen implementieren.

### Hypertext Preprocessor

Ursprünglich als Hobby-Projekt des Entwicklers Rasmus Lerdorf unter dem Namen *Personal Homepage Tools* entstanden, entwickelte sich PHP zu einer Programmiersprache mit der man einfach dynamische Webanwendungen erstellen kann. Anfangs

---

<sup>2</sup>Wenz (2014), S. 18.

<sup>3</sup>Vgl. a.a.O S. 19

bestand PHP nur aus einer Sammlung von *Perl*-Scripten die den Zugriff auf Webseiten protokollierten. Jedoch wurde der Funktionsumfang stetig erweitert, weshalb PHP später aus Performanz gründen komplett in der Programmiersprache *C* verfasst wurde.<sup>4</sup>

Heute zeichnet sich PHP vor allem durch seine umfangreichen Funktionsbibliotheken und durch seine hervorragenden Anknüpfungsmöglichkeiten an Datenbanksysteme aus. PHP-Scripte liegen ständig auf dem Webserver und werden nie mit dem statischen Content an den aufrufenden Client ausgeliefert. Sie werden bei einer Anfrage des Clients auf dem Server ausgeführt und liefern danach eine Antwort zurück. Somit eignen sich die PHP-Scripte ebenfalls als eine Implementierungsmöglichkeit für Server.

## Java

Java entstand in den 1970er Jahren als Wunsch des Softwareentwicklers Bill Joy, die Vorteile einiger bereits etablierten Programmiersprachen wie *C* und *Mesa* in einer neuen Sprache zu vereinen. Über einige Umwege entstand letztendlich die Programmiersprache *Oak* die schlussendlich 1994 den Namen Java bekam.<sup>5</sup>

Java ist eine objektorientierte Programmiersprache, die sich durch einige zentrale Eigenschaften auszeichnet. Diese machen sie universell einsetzbar und für die Industrie als robuste Programmiersprache interessant. Da Java objektorientiertes Programmieren ermöglicht, können Entwickler damit relativ einfach moderne und wiederverwertbare Softwarekomponenten entwickeln<sup>6</sup>.

In den folgenden Kapitel werden die Programmiersprachen untereinander verglichen und auf ihre Verwendbarkeit für den Prototypen der neuen Hochschul-App analysiert.

## Allgemeine Beliebtheit

Bei der Wahl der Programmiersprache sollte unter anderem auch auf die allgemeine Beliebtheit der Sprache geachtet werden, da diese die Weiterentwicklung der Anwendung maßgeblich beeinflussen kann. Sollte eine Sprache verwendet werden, die

---

<sup>4</sup>Vgl. Christian Wenz (2019), S. 31 ff.

<sup>5</sup>Vgl. Ullenboom (2019), S. 49 f.

<sup>6</sup>a.a.O S. 51



im Trend immer schlechter bewertet wird und kaum noch genutzt wird, so könnten die Studierenden in der Zukunft das Interesse an der Weiterentwicklung der Anwendung verlieren.

Hierbei sticht vor allem die Sprache Java deutlich heraus. Laut dem Popularity of Programming Language Index (PYPL) liegt Java im Gesamtvergleich aller Programmiersprachen auf Platz 2 mit etwa 19% des Anteils der Suchen von Programmiersprachen auf der Suchplattform *Google*. Doch auch JS (Platz 3, 8%) und PHP (Platz 5, 6%) können sich laut PYPL durchaus als beliebt bezeichnen.<sup>7</sup> Anhand der Werte ist zu erkennen, dass alle drei Sprachen grundsätzlich geeignet sind, Java hier prozentual jedoch klar besser abschneidet als JS und PHP.

Anzumerken ist hierbei, dass PYPL lediglich die Suchanfragen der Stichwörter zu den Programmiersprachen auf der Suchmaschine *Google* untersucht und daraus die Beliebtheit der Sprachen ableitet. PYPL nimmt dabei an, dass eine Programmiersprache umso beliebter ist, wenn sie oft in Verbindung mit Suchanfragen im Internet zu finden ist<sup>8</sup>. Als Messwert für die Beliebtheit einer Sprache für die Auswahl der Programmiersprache in dieser Arbeit soll dieser Wert aber genügen.

## Nutzbarkeit der Sprache

Ähnlich zur Beliebtheit einer Programmiersprache bietet die Nutzbarkeit der Sprachen einen weiteren wichtigen Punkt zur Entscheidungsfindung. Unter Nutzbarkeit ist in diesem Kontext der Schreibfluss, die Einfachheit des Syntax und die vorhandene Dokumentation einer Sprache zu verstehen. Des weiteren sollten auch die nötigen Rahmenbedingungen und Vorbereitungen für die Nutzung der Sprache sowie die benötigten Lines of Code (LOC) für das Erfüllen einer bestimmten Aufgabe betrachtet werden. Auch die Lizenzierung der Sprachen spielt bei der Nutzung in einer Hochschulumgebung eine wichtige Rolle.

Der Schreibfluss bei den Sprachen Java, JS und PHP unterscheidet sich nur minimal. JS lehnt seine Syntax sogar an der von Java an. Auch PHP orientiert sich an einer ähnlichen Syntax, störend wirkt hier lediglich das `$`-Zeichen vor den Variablen, was je nach Tastatur Layout schwieriger zu tippen ist. Den größten Schreibfluss hat man wohl bei JS, da dort Typisierung und störende Sonderzeichen komplett wegfallen.

---

<sup>7</sup>PYPL PopularitY of Programming Language (2020).

<sup>8</sup>ebd.

Jedoch ist das eine rein subjektive Auffassung und kann von anderen durchaus anders eingeschätzt werden. Die Dokumentation ist bei allen Sprachen hervorragend, wobei sich hier auch Java durch seine starke Community auszeichnet.

Bei der Nutzung der Sprachen und den nötigen Rahmenbedingungen stellt sich die Sprache JS als hervorragend dar, da für die Nutzung dieser Sprache lediglich ein Web-Browser von Nöten ist. Für Java wird hingegen eine Laufzeit- beziehungsweise eine Entwicklungsumgebung benötigt, bei PHP ein Server mit den benötigten Installationspaketen. Das schnelle Lösen von Aufgaben mit möglichst wenigen LOC ist die Stärke von PHP, da die Sprache eine breite Auswahl an Bibliotheken und Funktionen bereits im Sprachumfang enthalten hat.

Bei der Lizenzierung gibt es für die Sprachen PHP und JS keine Probleme, für Java kann die Open-Source Implementierung OpenJDK statt der originalen Version, die unter Oracle lizenziert ist, verwendet werden. Somit fällt bei der Nutzbarkeit der Sprachen kein Kandidat aus der Auswahl heraus. Anzumerken ist, dass vor allem JS leicht zu verwenden ist, jedoch Java hier die Grundlage für die Syntax bietet. Auch die statische Typisierung der Sprache Java macht es Entwicklern leichter, fehlerfreie Software zu schreiben.

## **Zuverlässigkeit der Sprache**

Bei der Zuverlässigkeit einer Sprache geht es vor allem um die Fehleranfälligkeit der Software, die in dieser Sprache geschrieben wurde. Hierbei ist vorweg zu sagen, dass die meisten unerwarteten Fehler durch eine fehlende Typisierung oder durch schwer lesbare Code-Abschnitte verursacht werden. Da bei der Programmiersprache Java die statische Typisierung gegeben ist und die Syntax eindeutig ist, liegt die Sprache in dieser Kategorie klar vorne. JS und PHP haben hier das Problem, dass in der ursprünglichen Fassung der Sprachen eine Typisierung nicht vorgesehen war und erst später durch Zusätze versucht wurde, diese einzufügen oder zumindest die Fehleranfälligkeit dieser Sprachen durch nicht-typisierte Variablen zu minimieren. Zudem bieten beide Sprachen mehrere Möglichkeiten einfache Probleme durch unterschiedlichen Syntax zu lösen, was bei der Programmierung eventuell zu Problemen führen kann. Auch die Einschätzungen dieses Absatzes sind jedoch subjektiv und müssen nicht von jedermann akzeptiert werden.

## Eignung für Anwendung

Für die Eignung der Anwendung qualifiziert sich klar die Programmiersprache Java. Es ist zwar durchaus möglich, in JS und PHP Web-Server zu schreiben, der Ursprung beider Sprachen liegt aber in der Manipulation von HTML-Content. Java hingegen bietet durch Java *EE* und andere Spracherweiterungen nativ die Grundlagen, Web-Server zu implementieren. Durch die Laufzeitumgebung von Java ist es auch einfach, auf Servern mit hoher Rechenkraft aufwändige Anfragen zu bearbeiten. Des weiteren bieten Drittanbieter einige Frameworks für Java, die genau auf die Bedürfnisse von Web-Anwendungen ausgelegt sind.

## Programmiererfahrung in der Sprache

Als letzter Punkt soll die allgemeine Programmiererfahrung der Studierenden der Hochschule Hof betrachtet werden, denn diese ist ausschlaggebend dafür, welche Sprachen in Zukunft für die Pflege und Weiterentwicklung der Anwendung in Frage kommen. Auch hier stellt sich Java in den Vordergrund, da Java in der Fakultät Informatik als Einstieg in die Programmierung genutzt wird. Weiterführende Vorlesungen bauen dann oft auf den in der Programmiersprache Java gelernten Kenntnissen auf. Sowohl JS und PHP werden je nach lehrenden Dozenten zwar auch behandelt, jedoch nur selten als Hauptsprache einer Vorlesung, weshalb die Kenntnisse hier innerhalb der Studierenden stark schwankt.

## Fazit

Trotz der Attraktivität der Sprachen JS und PHP im Bereich der Web-Programmierung und der Nutzung im Allgemeinen fällt die Wahl der Programmiersprache für den Prototypen der neuen Hochschul-App klar auf Java. Die Kenntnisse der Studierenden der Hochschule sowie die hohe Zuverlässigkeit der Sprache und die Eignung für den Anwendungsfall des Web-Servers überwiegen hier klar.

## 3.2 Microservice Framework

Im Bereich der Server-seitigen Programmierung in Java haben sich im Laufe der letzten Jahre einige Frameworks zur Unterstützung typischer Webanwendungen etabliert. Dabei fokussieren sich diese Frameworks im allgemeinen auf verschiedene Use-Cases, zu denen zum Beispiel leichtgewichtige Anwendungen, große Server-

implementierungen oder auch große Anwendungen, die aber wenig Daten im Hintergrund halten, gehören. Um etwas genauer auf die Wahl des Frameworks einzugehen, werden kurz einige davon aufgelistet und deren Use-Cases dargestellt.

- **Spring Boot**

*Spring Boot* ist ein Framework, das darauf ausgerichtet ist, komplexe Strukturen in einer Web Applikation einfach erstellen zu können und die Konfiguration der Ressourcen so einfach wie möglich zu halten. Wegen der breiten Einbindung anderer Frameworks für Dinge wie Datenbankbindung, Discovery Services und andere Hilfsmittel bietet *Spring Boot* eine gute Grundlage für die schnelle Entwicklung einer großen Anwendung. *Spring Boot* ist vor allem bekannt durch seine Einsatzmöglichkeiten im Microservice Bereich in Verbindung mit REST-APIs.<sup>9</sup>

- **Struts**

*Struts* ist ein open-source Framework, welches für moderne Web Anwendungen entwickelt wurde. Durch eine Plug-In Architektur soll *Struts* die Einbindung anderer Bibliotheken, beziehungsweise die Entwicklung neuer Schnittstelle,n deutlich vereinfachen. *Struts* ist nicht zwingend für Microservices ausgelegt und kann weit mehr als nur REST-Anwendungen unterstützen.<sup>10</sup>

- **Kumuluz Enterprise Edition (EE)**

*Kumuluz EE* ist ein open-source Microservice Framework. Im Gegensatz zu *Spring Boot* legt *Kumuluz EE* seinen Fokus auf auf die Entwicklung leichtgewichtiger Web Anwendungen, wobei auch hier Schnittstellen für leichtgewichtige Erweiterungen geboten werden.<sup>11</sup>

- **Java API for RESTful Web Services (JAX-RS)/ Jersey**

*Jersey* ist eine Implementierung von *JAX-RS*, eine Bibliothek, welche darauf abzielt, einen Standard für die Entwicklung von Java REST Services mit Java EE zu bereitzustellen. Da dieser Standard so viele Funktionen wie möglich unterstützen soll, ist er sehr mächtig, im gleichen Zug allerdings auch deutlich aufwändiger in der Nutzung. Des weiteren wurde *JAX-RS* auch dafür entwickelt, sowohl den Server als auch die benötigten Clients in Java zu implementieren.<sup>12</sup>

---

<sup>9</sup>Spring Boot (2020).

<sup>10</sup>Apache Struts (2020).

<sup>11</sup>Kumuluz EE (2020).

<sup>12</sup>Building RESTful Web Services with JAX-RS (2020); Jersey (2020).

## Kriterien bei der Framework Wahl

Die oben genannten Frameworks sind nur eine Auswahl der bekanntesten und der meist genutzten Frameworks für Web Anwendungen. Da hier eine große Auswahl für die Entwickler zur Verfügung steht, bedarf es eine Sammlung von Funktionen, die das endgültig gewählte Framework auch beherrschen muss. Zudem gibt es auch noch einige Kriterien, die die Frameworks erfüllen sollten. Diese Funktionen und Kriterien werden hier kurz aufgelistet und erläutert.

- **Startzeiten**

Jeder Webserver muss an mehreren Punkten in seinem Lebenszyklus auch gestoppt und gestartet werden. Ist ein Server gestoppt, so kann er seine Funktionen nicht erfüllen. Diesen Zeitraum, in der ein Server keine Anfragen bearbeiten kann, nennt man *Downtime*. Im Falle des Prototypen der neuen Hochschul-App sollte sich diese *Downtime* möglichst auf die Nacht beschränken. Sollte der Server dennoch aus unerwarteten Gründen in den Hauptnutzungszeiten gestoppt werden, so ist es durchaus wichtig, wie schnell er neu gestartet werden kann. Da sich die voraussichtliche Nutzung der neuen Hochschul-App auf wenige Tausend Studierende beschränken wird ist dieses Kriterium lediglich für die Weiterentwicklung relevant, da kurze Ausfallzeiten von ein paar Minuten durchaus vertretbar sind.

- **Erste Anfrage**

Jeder der oben genannten Frameworks unterstützt Caching Mechanismen, die es erlauben, Daten zwischenspeichern, sobald sie einmal von einem Client abgefragt wurden. Das ist vor allem bei Daten sinnvoll, die sich selten ändern, da so nicht die komplette Schichtenarchitektur bis zur Datenbank des Servers durchlaufen werden muss, um die benötigten Daten abzufragen. Jedoch werden Daten in der Regel erst im Cache abgelegt, wenn sie das erste mal abgefragt wurden. So ist die Bearbeitungszeit so einer Anfrage durchaus von Relevanz. Im Falle der Hochschul-App ist dieses Kriterium durchaus relevant, aber nicht ausschlaggebend für die Wahl des Frameworks. Die meisten Daten können gecached werden, jedoch hält sich die allgemeine Komplexität der Datenabfrage in Grenzen, was die Bearbeitungszeit von Anfragen gering hält.

- **Antwortzeiten bei kleinen Datenmengen**

Wie im Punkt *Erste Anfrage* bereits erwähnt wurde, liegt bei Web Server Implementierungen der Fokus auf Antwortzeiten. Dabei muss man oft die Art

der Anfrage unterscheiden. Neben der ersten Anfrage einer Ressource und der Anfrage einer Ressource aus dem Cache muss man noch zwischen den angefragten Datenmengen unterscheiden. Dabei ist es irrelevant, ob die Daten gecached sind oder nicht. Kleinere Datenmengen sollten mit minimalem Zeitversatz beantwortet werden. Das ist vor allem auch bei der Hochschul-App wichtig, denn dort werden in der Regel nur kleinere Datenmengen abgefragt, die dann aber auch möglichst schnell auf den Endgeräten der Studierenden angezeigt werden sollen

- **Antwortzeiten bei großen Datenmengen**

Im Gegensatz zu den Antwortzeiten bei kleineren Datenmengen ist diese bei größeren Datenmengen im Falle der neuen Hochschul-App weniger relevant. Es kann zwar durchaus vorkommen, dass große Datensätze angefragt werden, das sollte aber nur bei der Einrichtung der Nutzerpräferenzen und der personalisierten Daten vorkommen. Da das pro Nutzer in der Regel nur einmal passiert, sind die Antwortzeiten bei Anfragen mit großen Datenmengen weniger relevant.

- **Verhalten bei Hochlast**

Jede Web Anwendung hat sehr wahrscheinlich einen Zeitraum, in der überdurchschnittlich viele Nutzer auf sie zugreifen. In dieser Zeit ist es kritisch, die Anfragen trotz der hohen Last schnell beantworten zu können, da man sonst die Nutzer abschreckt und sich zusätzlich verwundbar für Denial of Service (DoS) und Distributed Denial of Service (DDoS) Angriffe macht. Bei der neuen Hochschul-App ist davon auszugehen, dass solche Stoßzeiten früh am Tag vor der ersten planmäßigen Vorlesung und durchaus auch später am Tag vor der Mittagspause entstehen können. Trotz der geringen Gesamtnutzerzahl sollte nun darauf geachtet werden, dass die Webserver der Anwendung auch unter Last nur geringe Ressourcen verbrauchen, da noch nicht klar ist, wie mächtig die darunterliegende Hardware ist.

- **Einfachheit der Entwicklung**

Der wohl ausschlaggebende Punkt bei der Wahl des passenden Frameworks für den Prototypen der neuen Hochschul-App ist die Einfachheit der Nutzung dieses Frameworks. Möchte man eine Anwendung schaffen, die im Umfeld von Studierenden, die ihr sich ihr professionelles Wissen erst aneignen, weiterentwickelt werden soll, so sollte man sich auf einfache und doch mächtige Techniken beschränken.

## Entscheidungsfindung

Grundsätzlich erfüllen alle der oben genannten Frameworks die Kriterien aus Kapitel 3.2, besonders bei der erwarteten geringen Nutzung der Anwendung aufgrund der Größe der Hochschule Hof. Dennoch erweist sich das *Spring Boot* Framework besonders für die Implementierung der neuen Hochschul-App. Die Gründe hierfür werden im folgenden kurz erläutert.

## Hervorragende Dokumentation

*Spring Boot* verfügt über eine hervorragende Dokumentation. Dies zeigt sich hauptsächlich durch zwei Gründe. Der erste ist die sehr simple und umfangreiche Dokumentation, die *Spring Boot* von sich aus zur Verfügung stellt. Der zweite ist die große Community, die durch die Popularität des Frameworks stetig am wachsen ist. Durch diese Community sind einige Foren entstanden, in denen viele häufig auftretende Probleme bei der Entwicklung diskutiert werden.

## Kompatibilität

*Spring Boot* liefert von Haus aus eine große Liste von unterstützten Frameworks mit sich. Dazu gehören unter anderem Hibernate (eine Java Persistence API (JPA) Implementierung), Zuul (API-Gateway) und JUnit5 (Testframework). Des Weiteren ist die Einbindung anderer Frameworks durch die Nutzung von Maven<sup>13</sup> sehr einfach gestaltet.

## Abstrahierung

Web Anwendungen sind oft voller komplexer Konzepte, speziell in Verbindung mit Java EE. Hierbei wird oft auf komplizierte Annotations, Dependency Injection und Beans gesetzt. Die Nutzung solcher Konzepte ist grundsätzlich unterschiedlich zu der normalen Verwendung der Programmiersprache Java. Des Weiteren werden diese Konzepte nur geringfügig in den Vorlesungen der Fakultät Informatik der Hochschule Hof gelehrt. *Spring Boot* vereinfacht die Nutzung dieser Konzepte durch die Einführung der Auto-Konfiguration und vereinfachten Annotations. Dies macht Spring sehr mächtig und dennoch leicht nutzbar für Entwickler mit beschränkter Programmiererfahrung.

---

<sup>13</sup>Siehe 3.3

### Einfache Nutzung

*Spring Boot* bietet ein Web-Service an, der es Nutzern erlaubt, Projekte, die mit *Spring Boot* implementiert werden sollen, über eine Webseite zu erstellen und sie dann vorkonfiguriert herunterzuladen. Dieser Service heißt *Spring Initializr* [sic!].<sup>14</sup> Nach dem Herunterladen des voreingestellten Projektes können Feineinstellungen dann über sogenannte *Property Files* vorgenommen werden.

Aufgrund der oben genannten Vorteile des *Spring* Frameworks wird dieses zur Implementierung des Prototypen der neuen Hochschul-App genutzt. Anzumerken ist hier auch, dass die Entwickler des Prototypen bereits Erfahrung bei der Nutzung von *Spring Boot* haben, was auch in die Entscheidungsfindung eingeflossen ist. Des weiteren ist festzuhalten, dass die Anwendung prinzipiell mit allen oben genannten Frameworks gut implementiert werden kann, dennoch muss das Niveau der Programmierung von Studierenden mit in Betracht gezogen werden, weshalb *Spring Boot* durch seine Einfachheit und der trotzdem enthaltenen Mächtigkeit die beste Wahl ist. Genauer zur Konfiguration und Details zum Framework *Spring Boot* wird in Kapitel 4 beschrieben.

## 3.3 Build Management Tool

Zur einfachen Einbindung von neuen Bibliotheken und zur Konfiguration des Build Prozesses benötigt der Prototyp der neuen Hochschul-App ein sogenanntes Build Management Tool. Die bekanntesten Vertreter dieser Tools sind aktuell Gradle, Maven und Ant. Diese Vertreter sollen nun kurz betrachtet werden, worauf dann die Entscheidung für eines der drei Tools begründet wird.

- **Ant**

*Ant* ist ein Build Management Tool das von der Apache Software Foundation entwickelt wurde. Es erlaubt seinen Nutzern, den Build Prozess einer Anwendung zu automatisieren. Zudem ist anzumerken, dass *Ant* speziell für Java Anwendungen entwickelt wurde. Der große Vorteil von *Ant* ist seine Erweiterbarkeit, denn man kann mit selbst entwickelten Java Tasks die Automation immer weiter verfeinern.<sup>15</sup>

---

<sup>14</sup>Siehe Spring Initializr (2020).

<sup>15</sup>Vgl. Richard Hightower (2004), S. 37.



- **Gradle**

Im Gegensatz zu *Ant* ist *Gradle* ein Build Management Tool, welches dafür entwickelt wurde, jede Art von Software zu unterstützen. *Gradle* ist ein open-source Projekt, das in der Java Virtual Machine (JVM) Umgebung läuft. Das soll den Kennern der Sprache Java ermöglichen, Erweiterungen für *Gradle* zu entwickeln.<sup>16</sup>

- **Maven**

*Maven* ist aus dem Wunsch heraus geboren, einfach neue Funktionalitäten in einen Build-Prozess während einer Entwicklung einer Anwendung hinzuzufügen. Solche Funktionalitäten können beispielsweise Unit-Tests oder das Erfassen von Codemetriken sein. Die Anfänge von *Maven* bauen auf *Ant* auf. Jedoch nutzt *Maven* mittlerweile *Jelly* als primären Scripting-Engine. Die Stärke von *Maven* ist es vor allem, mächtige *Ant* Skripte mit in den Build Prozess aufzunehmen.<sup>17</sup> Des weiteren bietet Apache ein mächtiges Verzeichnis an, das es ermöglicht, Bibliotheken in verschiedensten Versionen per *Maven Dependency* automatisiert in das Projekt mit einzubinden.

## Entscheidungsfindung

Für den Prototypen der neuen Hochschul-App wird das Build Management Tool *Maven* genutzt. Ein Vorteil dessen ist, das *Spring Boot* in seinem *Spring Initializr* das Erstellen von sowohl *Maven*- als auch *Gradle*-Projekten unterstützt. *Maven* bietet sich aus diesen zwei Möglichkeiten deshalb an, da es unter anderem sehr leicht und intuitiv zu nutzen ist. Außerdem kann man mit *Maven* sehr einfach *Ant* Skripte einbinden.

## 3.4 Versionsverwaltung Tool

Jedes Projekt, ob nun kollaborativ erstellt oder in Eigenarbeit geschaffen, benötigt ein Versionsverwaltung-Tool, idealerweise mit der Möglichkeit den Source Code auf externen Servern zu speichern, um Verluste durch einen Ausfall des Entwickler-Computers zu vermeiden. In der heutigen Zeit kommen dabei in der Regel nur noch

---

<sup>16</sup>What is Gradle? (2020).

<sup>17</sup>Vgl. Richard Hightower (2004), S. 457.

zwei Kandidaten in Betracht, *git* und *Mercurial*. Für dieses Projekt gilt *git* jedoch als einzig sinnvolle Option. Das liegt vor allem daran, dass die Hochschule einen eigenen GitLab Server besitzt, welcher es ermöglicht, Projekte, die über *git* verwaltet werden, zu synchronisieren und auf dem Hochschul Server zu speichern. Des weiteren ist *git* das weltweit wohl verbreitetste Versionsverwaltung-Tool. Auch der Lehrinhalt der Vorlesungen der meisten Dozenten der Hochschule Hof beinhaltet lediglich das Tool *git*.

### 3.5 API Gateway

Wie in der parallel verfassten Bachelorarbeit bereits ausführlich erläutert wurde, besteht die Architektur der neuen Hochschul-App aus mehreren Microservices, die die implementierten Funktionalitäten jeweils kapseln.<sup>18</sup> Um dennoch eine einheitliche Anwendung zu schaffen, die nach außen nur eine Schnittstelle hat, wird ein sogenanntes API-Gateway verwendet, das als Fassade zu den einzelnen Services dienen soll. Diese Microservices sind in folgende Funktionalitäten gekapselt:

- Stundenplan
- Planänderung
- Speiseplan
- Benachrichtigungen
- Sicherheit
- Anwenderverwaltung
- Sprachzentrum
- Termine

Jedoch ist enthält der Umfang dieser Arbeit lediglich die Stundenplaninformationen, Stundenplanänderungen Mensadaten und den Benachrichtigungsservice. Alle anderen Services werden in einer anderen Abschlussarbeit separat entwickelt. Um die Anbindung der restlichen Services zu ermöglichen, ohne dass sich die Schnittstelle nach außen ändert, dient ebenfalls das API-Gateway. Das Gateway ermöglicht es ebenfalls, zusätzlich zur Fassaden-Funktion, ein Load-Balancing der einzelnen

---

<sup>18</sup>Brysiuk; Lehmann (2019a).

Services zu konfigurieren, wenn diese redundant auf mehreren physischen Instanzen laufen. Die Auswirkung, die bei der redundanten Haltung der Service-Instanzen in Kombination mit der Microservice-Architektur anfallen würden, sind in der folgenden Abbildung bildlich dargestellt.

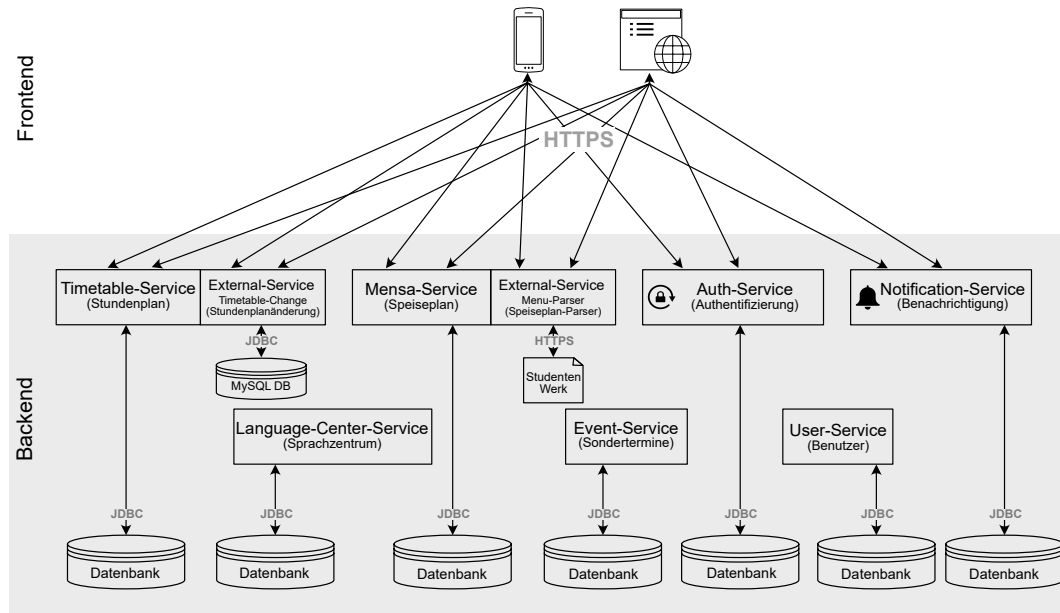


Abbildung 3.1: Hochschull-APP Architektur ohne Gateway

Eine weitere Funktion des Gateways ist die Versionsverwaltung der Services, da dieses Projekt lediglich den Prototypen implementiert und in Zukunft noch weitere Releases erscheinen werden. Deshalb und wegen der bereits genannten Vorteile wird im Prototyp ein API-Gateway eingebunden.

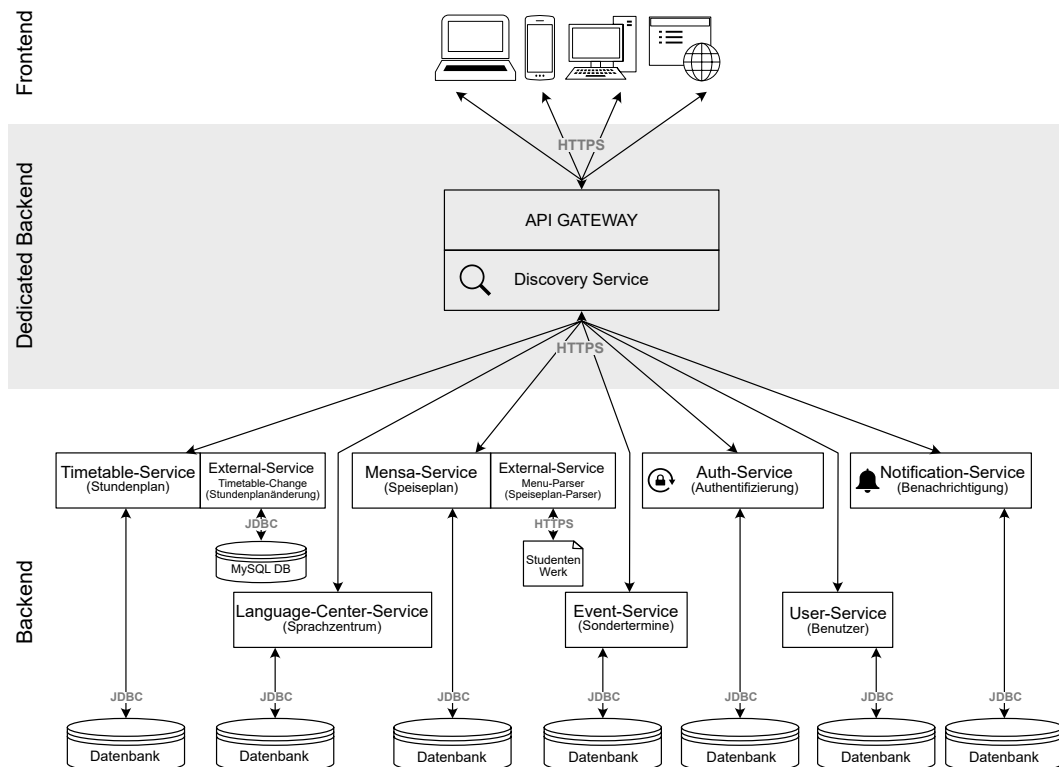


Abbildung 3.2: Hochschul-APP Architektur mit Gateway

Für die Implementierung des Gateways bieten sich ähnlich wie beim Microservice Framework zahlreiche Alternativen. Darunter sind unter anderen *NGINX*, *Spring Cloud Gateway*, *Zuul* und *KrakenD*. Da die Wahl des Microservice Frameworks jedoch bereits auf *Spring Boot* gefallen ist, wurde die Auswahl der API-Gateway Frameworks auf lediglich zwei Kandidaten reduziert, welche bereits in Verbindung mit *Spring* getestet wurden und mit dem Framework harmonieren. Diese Frameworks sind *Zuul* und *Spring Cloud Gateway*.

Auch zwischen diesen beiden Frameworks bedarf es jedoch kaum eines aufwändigen Vergleichs, da die von *Spring* entwickelte Gateway Implementierung auf der Basis von *Zuul* implementiert wurde und dieses in Zukunft auch endgültig in Verbindung mit *Spring Boot* ablösen soll. Zudem wurde der Support für *Zuul* in der Version 1.0 eingestellt, was potentielle Sicherheitslücken nicht ausschließen lässt. Neuere Versionen des Frameworks werden von *Spring* ohnehin nicht mehr unterstützt.

## 3.6 Service Registrierung und Discovery

Durch die Nutzung des API-Gateways wurde die Auslagerung der Funktionalitäten in dezentrale Einheiten ermöglicht. Das bringt, wie bereits erwähnt, einige Vorteile mit sich, erfordert jedoch auch, dass das Gateway selbst die Adressen der einzelnen Services kennt. Diese müssen sich dadurch mit ihren physischen Adressen beim Gateway oder einer anderen zentralen Instanz registrieren. Hierfür wurde beim Prototypen der Hochschul-App ein Discovery Server eingebaut. Dieser kümmert sich um das Load-Balancing, verteilt die Anfragen zwischen den Instanzen und leitet die Anfragen an eine ausgefallene Instanz an den passenden Backup Service weiter.

Für die konkrete Entwicklung dieser Discovery Funktion gibt es eine Reihe bewährter open-source Lösungen. Zu diesen gehören Frameworks, die auf die Konsistenz der Services ausgelegt sind, wie *Apache Zookeeper*, *Doozer* und *Etcd*, jedoch auch Frameworks, die auf dedizierte Lösungen setzen, um sich auf Load-Balancing und die eigentliche Service Discovery zu konzentrieren, wie *netflix Eureka*, *Spotify DNS*, *SkyDNS*, *NSQ* oder *SmartStack*.<sup>19</sup>

Alle Frameworks bringen bestimmte Vor- und Nachteile mit sich, wobei für die eigentliche Implementierung der Hochschul-App prinzipiell jedes der genannten Framework verwendet werden kann. Deshalb wird bei der Wahl des Discovery Frameworks das Hauptaugenmerk auf die Einbindung in das *Spring* Framework gelegt. Speziell in der *Spring Cloud* existieren bereits Lösungen für dieses Konzept, von denen eines das oben genannte *Netflix Eureka* ist. Dieses baut auf Services aus dem *Netflix Open Source Software Center (OSS)* auf, aus dem auch einige Konzepte in *Spring Cloud* genutzt werden, weshalb sich *Netflix Eureka* perfekt für die Nutzung im Prototypen der neuen Hochschul-App eignet.

## 3.7 Benachrichtigung

Da es technisch nicht möglich ist, über eine reine Datenschnittstelle Benachrichtigungen an die aufrufenden Clients zu senden, weil nie klar ist, welche Clients den Dienst nutzen, muss ein Benachrichtigungsservice genutzt werden, bei dem sich die Clients registrieren können, um über mögliche Änderungen informiert zu werden. Hierbei bieten sich in der Praxis zwei Alternativen, eine Push-Benachrichtigungs-

---

<sup>19</sup>Open-Source Service Discovery (2020).

API selbst entwickeln oder das Nutzen von Drittanbieter Software mit Cloud-Lösungen. Da die Implementierung und Testen solcher Anwendungen sehr aufwändig ist, fällt die eigene Entwicklung dabei aus den möglichen Umsetzungen heraus.<sup>20</sup>

Deswegen wird auf eine Drittanbieter Lösung bei der Prototyp Entwicklung zurückgegriffen. Diese kümmern sich selbstständig um die Unterstützung unterschiedlicher Anwendungen und der Verschlüsselung der Benachrichtigungen. Eine weit verbreitete und mittlerweile gut getestete Lösung ist das von Google entwickelte Firebase Cloud Messaging (FCM). FCM ermöglicht plattformübergreifende Benachrichtigungen mit der zusätzlichen Unterstützung von mobilen Endgeräten, was im Fokus der Entwicklung der neuen Hochschul-App liegt.<sup>21</sup> Die kostenlose Lizenz für FCM ist für die Hochschull-App mehr als ausreichend. Es können täglich bis zu 20.000 Benachrichtigungen mit einem Nachrichteninhalt von 4Kilobyte (kB) an Clients versendet werden. Das entspricht um die 4096 Buchstaben pro Nachricht. Ein weiteres Argument für die Verwendung von FCM ist die einfache Einbindung des *Firebase Admin Software Development Kit (SDK)* über Maven. Dieses erlaubt die Konfiguration der Authentifizierung, Registrierung und des Versenden von Nachrichten über einen Service und über die Programmiersprache Java.

---

<sup>20</sup>Sending Web Push Notifications with Java (2020).

<sup>21</sup>Firebase console (2020).

## 4 Konfiguration

Um den Aufbau und die Einstellungen der grundlegenden Anwendung der Hochschul-App besser verstehen zu können, bietet es sich an, die Schritte, die bei der Erstellung dieser Anwendung durchgeführt worden sind, zu dokumentieren und zu erläutern. Im folgenden werden diese Schritte deshalb aufgeführt. Es soll jedoch nicht nur um die Einrichtung der Software im Allgemeinen gehen, sondern auch um die Einrichtung der Entwicklertools, die für die Arbeit an der Hochschul-App bereitgestellt wurden.

### 4.1 Entwicklungsumgebung

Für die Arbeit am Prototypen der neuen Hochschul-App wird das Nutzen einer IDE empfohlen. Der Vorteil hierbei liegt in der Verwaltung der Projektstruktur, der Abhängigkeiten und der automatisierten Kompilierung des Quellcodes. Dabei ist es grundsätzlich egal, welche IDE verwendet wird, da die meisten dieser Entwicklungsumgebungen die wichtigsten Funktionen bereits beinhalten. Dennoch wird hier explizit die Java IDE *IntelliJ IDEA* der Firma *JetBrains* empfohlen.<sup>1</sup> *IntelliJ IDEA* wird in zwei Formaten angeboten, einer umfassenden *Ultimate* Version, welche alle Funktionen der Anwendung beinhaltet, und einer vereinfachten *Community* Edition, welche kostenfrei ist, jedoch nicht alle Features unterstützt. Studierende können sich kostenlos eine Lizenz für *IntelliJ IDEA Ultimate* besorgen.

Im folgenden wird kurz die Einstellung der IDE *IntelliJ IDEA Ultimate* erläutert, welche nötig ist, um die Vorteile der geschaffenen Rahmenbedingungen vollständig zu nutzen. Die genutzte Version bei der Demonstration ist *INTELLIJ IDEA 2019.3.1 (ULTIMATE EDITION)*. Es wird hier noch einmal darauf hingewiesen, dass die Entwicklung unabhängig von der genutzten IDE möglich ist, die Nutzung von *IntelliJ IDEA* hier jedoch empfohlen wird. Außerdem soll hierbei auch noch einmal auf die Lizenzbedingungen der *Ultimate* Version verwiesen werden.<sup>2</sup>

---

<sup>1</sup>Siehe IntelliJ IDEA (2020).

<sup>2</sup>Siehe JetBrains (2020).

## Hot Deploy

*Hot Deploy* ist eine Funktion, welche es ermöglicht, den Programmcode einer Software automatisiert bei der Erkennung von Änderungen oder beim manuellen Anstoßen einer Speicherung zu kompilieren und dann auf den laufenden Server zu deployen. Das erspart dem Entwickler die Wartezeit, die beim manuellen Deployment einer Serversoftware entsteht und ist bei der Entwicklerarbeit von großem Vorteil.

## Einstellung in der IDE

Die Schritte zur Einrichtung sind folgende:

1. Öffnen der IDE *IntelliJ IDEA* und des Projektes, das die Microservices enthält
2. Öffnen des Menüpunktes *File -> Settings* (Alternativ STRG+ALT+S)
3. Auswählen der Option *Build, Execution, Deployment -> Compiler*

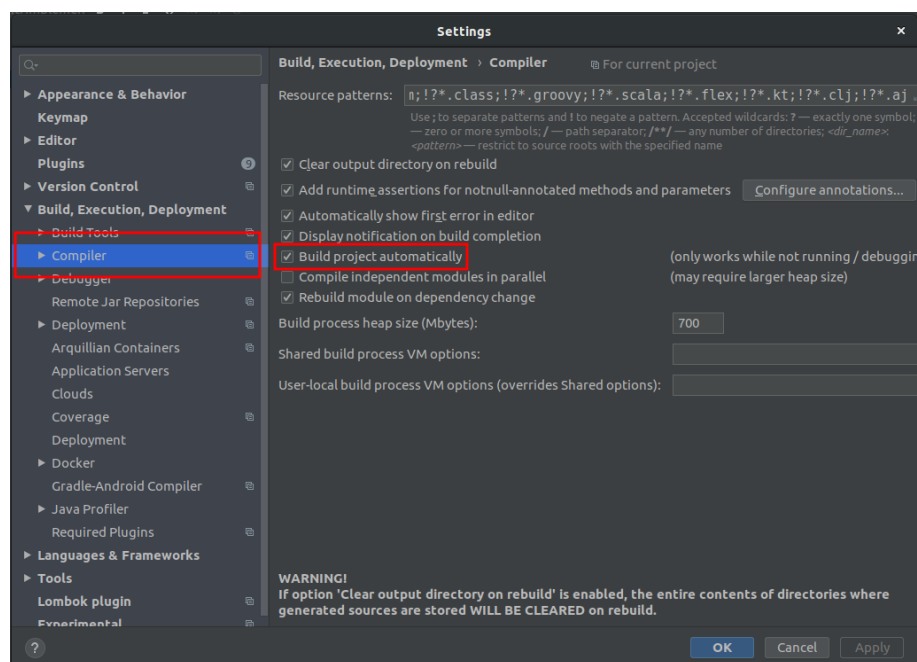


Abbildung 4.1: Hot Deployment in *IntelliJ*

4. Bestätigen der Checkbox *Build project automatically*
5. Öffnen der *Registry*
  - a) Tastenkombination STRG+N
  - b) Erweitern des Suchradius auf *All*



c) Suchen nach Registry

d) Auswählen des Punktes *Actions* -> *Registry...*

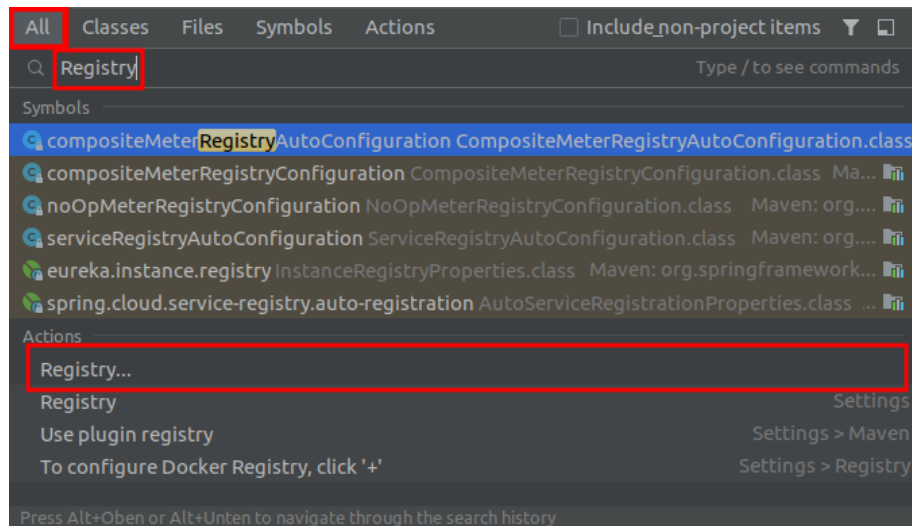


Abbildung 4.2: Öffnen der Registry in *IntelliJ*

6. Bestätigen der Checkbox *compiler.automake.allow.when.app.running*

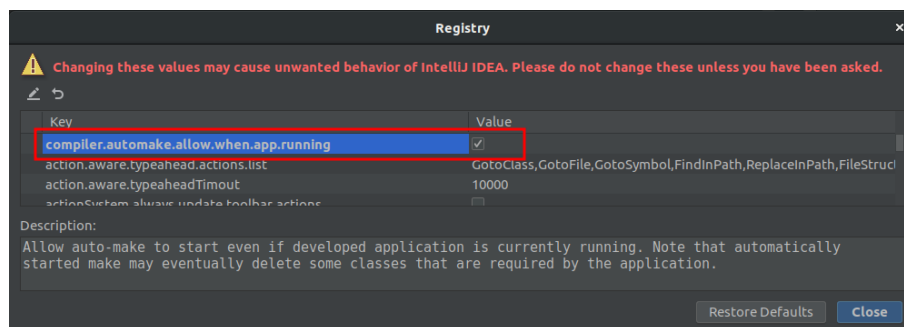


Abbildung 4.3: Compiler Anweisung in *IntelliJ*

7. Neustart der IDE

Der Start der Services erfolgt nach wie vor manuell. Sobald der Service läuft, werden Änderungen automatisch in den laufenden Server übernommen.



Abbildung 4.4: Start des Services in *IntelliJ*

Hierbei startet der *Play*-Button den Service normal, während der Button mit dem Käfer als Icon den Service im *Debug*-Modus startet.

## Manuelles Deployment

Das *Hot Deployment* wird in der Regel bei der Erkennung von Änderungen und in der IDE *IntelliJ IDEA* beim Drücken der Tastenkombination STRG+S angestoßen. Das kann auch der Fall sein, wenn gerade erst am Code geschrieben wird und die Software noch nicht kompilierbar ist. Außerdem verbraucht diese Funktion deutlich mehr Ressourcen auf dem Personal Computer (PC) des Entwicklers, was es zu einer Funktion macht, die eventuell nicht gewünscht ist. In diesem Fall kann ein Service der Hochschul-App folgendermaßen manuell deployed werden:

1. Start des Services wie in automatischem Deployment
2. Neustart des Services über Toolbar

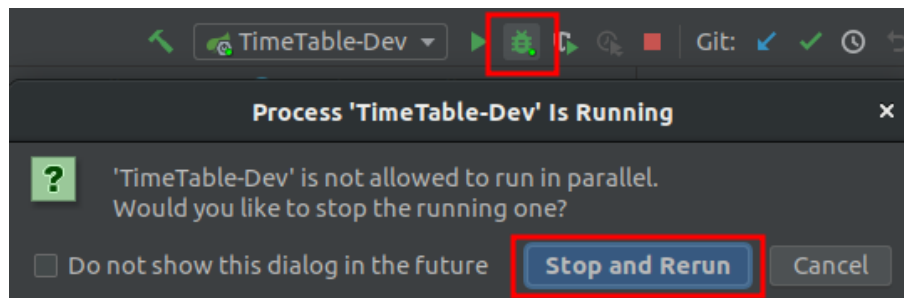


Abbildung 4.5: Neustart des Services in *IntelliJ*

## Lombok

*Lombok* ist ein Framework, welches es ermöglicht, einfachen Code, der sich in jedem Plain Old Java Object (POJO) befindet, nicht immer wieder neu tippen zu müssen. Durch einfache Annotations können so den POJOs *Getter*, *Setter*, *Konstruktor*en und andere häufig verwendeten Codeabschnitte zur Kompilierzeit hinzugefügt werden. Das macht das manuelle Erstellen dieser Codeabschnitte überflüssig und macht das Projekt allgemein lesbarer, da durch die Annotations die eigentlichen Methoden-Implementierungen entfallen.<sup>3</sup> Ein solches POJO könnte nach der Nutzung von *Lombok* folgendermaßen aussehen:

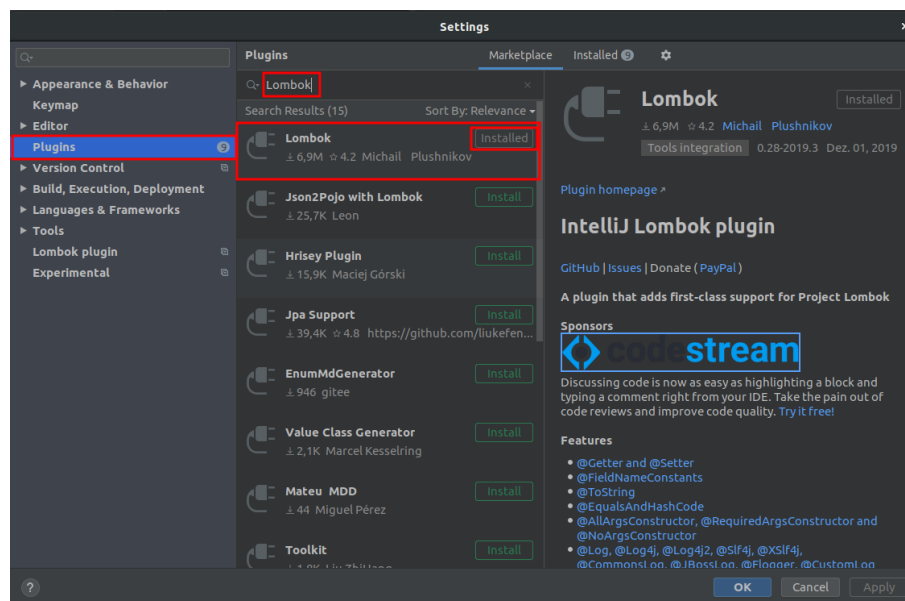
<sup>3</sup>Project Lombok (2020).

Code 4.1: Klassendefinition mit *Lombok*

```
1 package test.lombok.pojo
2
3 import lombok.AllArgsConstructor;
4 import lombok.Getter;
5 import lombok.Setter;
6
7 @Getter
8 @Setter
9 @AllArgsConstructor
10 class POJO {
11     private int attr1;
12     private int attr2;
13 }
```

*Lombok* wird im Sourcecode des Prototypen der neuen Hochschul-App bereits verwendet und sollte so auch in der IDE der Entwickler konfiguriert werden, da diese sonst möglicherweise den verwendeten Sourcecode nicht kompiliert, da sie nicht erkennt, dass verwendete Methodenreferenzen von *Lombok* erst zur Kompilierzeit hinzugefügt werden. Die Konfiguration in *IntelliJ IDEA* erfolgt folgendermaßen:

1. Öffnen der IDE *IntelliJ IDEA* und des Projektes, das die Microservices enthält
2. Öffnen des Menüpunktes *File -> Settings* (Alternativ STRG+ALT+S)
3. Auswählen der Option *Plugins*
4. Suche nach dem Plugin *Lombok*

Abbildung 4.6: Lombok Plugin in *IntelliJ*

## 5. Installation des Plugins und Neustart der IDE

### 4.2 Spring Boot

Wie bereits in Kapitel 3.2 erwähnt wurde, wird *Spring Boot* als Microservice Framework für den Prototypen der neuen Hochschul-App verwendet. *Spring* ist ein Framework, das durch die vereinfachte Einbindung anderer Bibliotheken die Entwicklung mit Java deutlich vereinfacht.<sup>4</sup> *Spring Boot* zeichnet sich vor allem dadurch aus, dass es Elemente aus dem Java EE Umfeld sowie open-source Bibliotheken zur Server-Entwicklung mit Fokus auf Microservices vereint. Im folgenden soll nun auf die Einbindung des *Spring Boot* Frameworks in das Gesamtprojekt eingegangen werden.

### Spring Initializr

*Spring Initializr* ist ein Dienst der Entwickler des *Spring* Frameworks, mit dem man voreingestellte Projekte erstellen und aus dem Dienst herunterladen kann. Dabei gibt man lediglich an, welche Abhängigkeiten das neue Projekt benötigt und mit welchen Versionen der Abhängigkeiten und Programmiersprachen das Projekt

<sup>4</sup>Vgl. Wolff (2007), S. 1.

laufen soll. Durch diesen Dienst wurde ebenfalls der Prototyp der neuen Hochschul-App erstellt, wobei hier zu beachten ist, dass alle Microservices als einzelnes Projekt betrachtet werden müssen und deshalb jeder Service seine eigenen Abhängigkeiten besitzt. Im folgenden wird kurz die Vorgehensweise zum Erstellen des Projektes erläutert, danach werden dann die einzelnen Abhängigkeiten aufgelistet.

### Erstellen des Projekts

Um ein Projekt mit *Spring Initializr* zu erstellen, muss man die Website <https://start.spring.io/> aufrufen. Danach durchläuft man folgende Schritte. Alle Abbildungen in der folgenden Abbildung stammen von der Website des Dienstes.<sup>5</sup>

#### 1. Auswahl des Projekt Build Tools und Dependency Managements

Für dieses Projekt wurde *Maven* ausgewählt. Mehr Informationen dazu sind in Kapitel 3.3 zu finden.

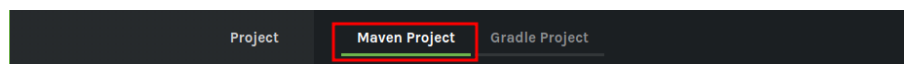


Abbildung 4.7: Projektwahl in *Spring Initializr*

#### 2. Wahl der Programmiersprache

*Spring Boot* unterstützt aktuell Java, Kotlin und Groovy. Für dieses Projekt wurde Java gewählt, zu diesem Thema wurde eine ausführliche Analyse in Kapitel 3.1 angefertigt.

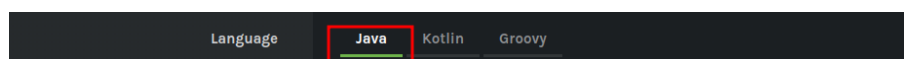


Abbildung 4.8: Programmiersprachenwahl in *Spring Initializr*

#### 3. Wahl der *Spring Boot* Version

Bei der Wahl der Version des Frameworks sollte man grundsätzlich eine Version wählen, die bereits aus der Snapshot Phase heraus ist. So umgeht man unvorhergesehene Fehler.

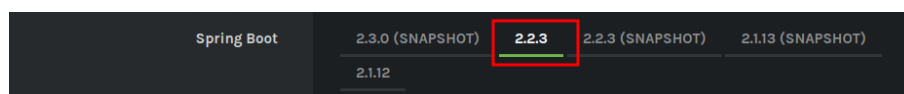


Abbildung 4.9: Versionswahl in *Spring Initializr*

<sup>5</sup>Spring Initializr (2020).

#### 4. Einstellung der Projekt Metadaten

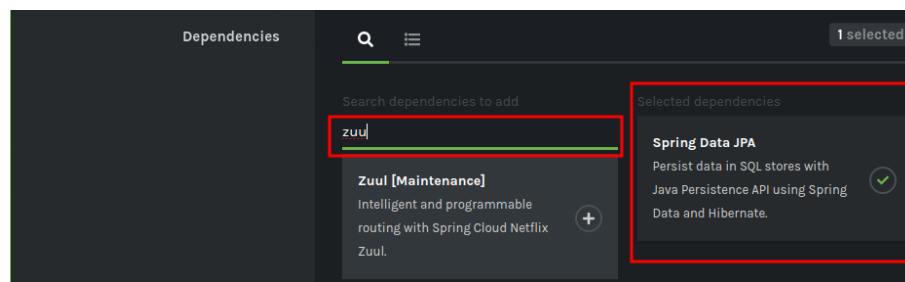
*Maven* Projekte erfordern der Eingabe einiger Projekt-Metadaten. Darunter fallen die *Group*, welche die Organisationseinheit beschreiben soll, welche das Projekt implementiert, diverse Bezeichner für das Projekt an sich und die Version der Programmiersprache, mit der das Projekt implementiert werden soll.

The screenshot shows the 'Project Metadata' configuration interface. It has a dark theme. The 'Group' field is set to 'com.example'. The 'Artifact' field is set to 'demo'. Under the 'Options' section, the 'Name' field is 'demo', the 'Description' is 'Demo project for Spring Boot', and the 'Package name' is 'com.example.demo'. The 'Packaging' section shows 'Jar' selected with a green bar, and 'War' is unselected. At the bottom, the 'Java' version is set to '11', with '13' and '8' also visible as options.

Abbildung 4.10: Meta Daten in *Spring Initializr*

#### 5. Hinzufügen der benötigten Dependencies

*Spring Boot* unterstützt standardmäßig eine breite Auswahl an anderen Bibliotheken, welche über den *Spring Initializr* einfach hinzugefügt werden können. Tut man dies über den Dient und nicht manuell mit *Maven*, so vermeidet man größtenteils Fehler und Konflikte in den Abhängigkeiten. Die Liste der Abhängigkeiten wird in Kapitel 4.2 aufgeführt. Möchte man eine Dependency hinzufügen, so sucht man einfach im Textfeld danach und wählt diese aus. Auf der rechten Seite der Anwendung erscheint dann eine Liste aller gewählten Abhängigkeiten.

Abbildung 4.11: Wahl der Abhängigkeiten in *Spring Initializr*

## Abhängigkeiten

Da der Prototyp der neuen Hochschul-App bereits eine breite Auswahl an Funktionen unterstützen soll und dabei auch ständig Erweiterbar und leicht zu pflegen bleiben muss, wurden in das Projekt eine Auswahl an Abhängigkeiten eingefügt. Diese werden hier nun kurz aufgelistet.

### Spring Web

Die Sammlung der vom *Spring*-Framework benötigten Abhängigkeiten ist im Projekt in einer Dependency zusammengefasst. Die benötigte *Maven* Dependency sieht wie folgt aus:

#### Code 4.2: *Spring Web* Dependency

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-web</artifactId>
4 </dependency>
```

### JPA

Für die Einbindung der Datenbanken wird in allen Services JPA verwendet. *Spring Boot* verwendet hierzu die JPA Implementierung *Hibernate*. Die benötigte *Maven* Dependency sieht wie folgt aus:

#### Code 4.3: *Spring Data JPA* Dependency

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-data-jpa</artifactId>
4 </dependency>
```

## H2

*H2* ist eine *In.Memory* Datenbank, welche im Prototypen der Hochschul-App als Testdatenbank genutzt wird, um nicht eine dauerhafte Verbindung zum Datenbank Server der Hochschule aufbauen zu müssen. Die benötigte *Maven* Dependency sieht wie folgt aus:

Code 4.4: H2 Dependency

```
1 <dependency>
2     <groupId>com.h2database</groupId>
3     <artifactId>h2</artifactId>
4     <scope>runtime</scope>
5 </dependency>
```

## MySQL Java Database Connectivity (JDBC) Dependency

Um auf den Datenbank Server der Hochschule zuzugreifen benötigt das Projekt einen JDBC. Da auf dem Server eine *MySQL* Datenbank installiert ist wurde ein *MySQL-JDBC* eingebunden. Die benötigte *Maven* Dependency für die Version des *MySQL*-Servers sieht wie folgt aus:

Code 4.5: *MySQL* JDBC

```
1 <dependency>
2     <groupId>mysql</groupId>
3     <artifactId>mysql-connector-java</artifactId>
4     <version>5.1.6</version>
5 </dependency>
```

## DevTools

Das in Kapitel 4.1 erwähnte Entwicklertool *Hot Deployment* benötigt ebenfalls eine Abhängigkeit, um in *Spring Boot* zu funktionieren. Die benötigte *Maven* Dependency sieht wie folgt aus:

Code 4.6: *Spring* DevTool Dependency

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-devtools</artifactId>
4     <scope>runtime</scope>
5     <optional>true</optional>
6 </dependency>
```



## Hypermedia

Die HATEOAS Funktion wurde bereits ausführlich in der parallel zu dieser Arbeit entwickelten Bachelorarbeit erläutert.<sup>6</sup> Leider konnte diese Funktion nicht in den Umfang der fertiggestellten Funktionen aufgenommen werden. Genauer dazu wird in Kapitel 8 angesprochen. Die Dependency wird eingefügt, jedoch auskommentiert, da das Projekt ohne die Nutzung der Abhängigkeit nicht korrekt funktioniert. Die benötigte *Maven* Dependency sieht wie folgt aus:

Code 4.7: HATEOAS Dependency

```
1 <!-- UNCOMMENT ONLY IF THIS TECHNOLOGY IS IMPLEMENTED ,  
   OTHERWISE NO BUILD -->  
2 <!--dependency>  
3     <groupId>org.springframework.boot</groupId>  
4     <artifactId>spring-boot-starter-hateoas</artifactId>  
5 </dependency-->
```

## Eureka Client

Die Service Discovery des Gesamtprojektes baut auf der von Netflix entwickelten *Eureka*-Bibliothek auf. Um sich auf einem *Eureka* Server registrieren zu können muss ein Service ebenfalls ein *Eureka* Client sein. Die benötigte *Maven* Dependency sieht wie folgt aus:

Code 4.8: *Eureka* Service Discovery Client Dependency

```
1 <dependency>  
2     <groupId>org.springframework.cloud</groupId>  
3     <artifactId>  
4         spring-cloud-starter-netflix-eureka-client  
5     </artifactId>  
6 </dependency>
```

---

<sup>6</sup>Brysiuk; Lehmann (2019a).

## Lombok

Die Verwendung von *Lombok* wurde in Kapitel 4.1 bereits ausführlich erläutert. Die benötigte *Maven* Dependency sieht wie folgt aus:

Code 4.9: Lombok Dependency

```
1 <dependency>
2   <groupId>org.projectlombok</groupId>
3   <artifactId>lombok</artifactId>
4   <optional>true</optional>
5 </dependency>
```

## Swagger

Zur automatischen Generierung der Endpunkt-Dokumentation wird *Spring-Fox Swagger* genutzt. Durch *Swagger* kann man RESTful Services einheitlich dokumentieren. Durch eine einfache, Menschen-lesbare Sprache, die auf *yaml* beruht, können sowohl Entwickler, als auch Maschinen die Definition verstehen.

Code 4.10: *Swagger* Dependency

```
1 <dependency>
2   <groupId>io.springfox</groupId>
3   <artifactId>springfox-swagger2</artifactId>
4   <version>2.9.2</version>
5 </dependency>
```

Aus der generierten Dokumentation lässt sich ebenfalls eine automatisierte Nutzeroberfläche generieren, welche zum Testen der Endpunkte genutzt werden kann. Diese Website wird automatisch in allen Services deployed und hilft den Nutzern, die Endpunkt-Dokumentation durch eine leicht verständliche Visualisierung besser zu verstehen.

Code 4.11: *Swagger-UI* Dependency

```
1 <dependency>
2   <groupId>io.springfox</groupId>
3   <artifactId>springfox-swagger-ui</artifactId>
4   <version>2.9.2</version>
5 </dependency>
```

## Test dependencies

Zum testen der einzelnen Services werden mehrere Test-Frameworks benötigt. Dazu gehören unter anderem JUnit, aber auch die von *Spring* eigens entwickelten Fra-

meworks zum mocken verschiedener Abhängigkeiten. Zum Testen der Anwendung wird in Kapitel 7 mehr geschrieben. Die benötigten *Maven* Dependencies sehen wie folgt aus:

Code 4.12: Test Dependencies

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-test</artifactId>
4   <scope>test</scope>
5   <exclusions>
6     <exclusion>
7       <groupId>org.junit.vintage</groupId>
8       <artifactId>junit-vintage-engine</artifactId>
9     </exclusion>
10  </exclusions>
11 </dependency>
12
13 <dependency>
14   <groupId>org.assertj</groupId>
15   <artifactId>assertj-core</artifactId>
16   <scope>test</scope>
17 </dependency>
```

### Eureka Server

Damit der Discovery Service die Registrierungen der Services dynamisch konfigurieren und verwalten kann, muss der Discovery Service als Eureka Server fungieren. Die dafür benötigte Maven Dependency sieht wie folgt aus:

Code 4.13: Eureka Server Dependencies

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>
4     spring-cloud-starter-netflix-eureka-server
5   </artifactId>
6 </dependency>
```

### Spring Cloud Gateway

Um die Requests der Benutzer an die registrierten Services weiterleiten zu können, benötigt der Proxy Service die Adressen der registrierten Services. Dies wird durch

die Konfiguration des Spring Cloud Gateways durchgeführt. Die dafür benötigte Maven Dependency für den Proxy Service sieht wie folgt aus:

**Code 4.14: Spring Cloud Gateway Dependencies**

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-gateway</artifactId>
4 </dependency>
```

**Firestore Cloud Messaging**

Um die Push Benachrichtigungen über den Notification Service versenden zu können wird die offizielle *Firebase Admin SDK* eingebunden. Dies wurde bereits in Kapitel 3.7 erwähnt. Die dafür benötigte Maven Dependency für den Notification Service sieht wie folgt aus:

**Code 4.15: FCM Dependencies**

```
1 <dependency>
2     <groupId>com.google.firebase</groupId>
3     <artifactId>firebase-admin</artifactId>
4     <version>6.11.0</version>
5 </dependency>
```

**Wichtige Annotations**

Eine der Eigenschaften, die *Spring Boot* auszeichnet, ist seine Fähigkeit, komplexe Zusammenhänge und Techniken, die ihren Ursprung im Java EE Umfeld haben, zu abstrahieren und dem Nutzer eine gut verständliche, leicht lesbare Lösung zu bieten. Das Konzept, auf das *Spring Boot* dabei verstärkt setzt sind aussagekräftige Annotations. Bereits bei der Initialisierung eines Services müssen einige Voreinstellungen durch Annotations durchgeführt werden. Diese werden im Zusammenhang mit ihrer Positionierung im folgenden kurz erläutert.

**Anwendungseinstieg**

Die Wurzel eines jeden *Spring Boot* Programms ist die Klasse, welche die MAIN-Methode als Einstieg definiert. Diese Klasse wird in allen *Spring* Anwendungen mit der Annotation `@SPRINGBOOTAPPLICATION` gekennzeichnet. Zusätzlich wird diese Klasse in den funktionalen Microservices des Prototypen der neuen Hochschul-App mit der Annotation `@ENABLEEUREKACLIENT` gekennzeichnet. Das bewirkt,

dass diese Services sich bei dem in den Konfigurationen definierten *Eureka*-Server als Clients registrieren.

Einige Services benötigen ebenfalls Jobs, welche in bestimmten Zeitintervallen ihre Funktionalitäten ausführen. Diese Jobs nennt man *Scheduled Jobs*. Beispiele hierfür sind der *Timetable-Service*, welcher ständig prüft, ob in seiner Datenquelle neue Kurse oder Änderungen eingetragen wurden und der *Mensa-Service*, welcher in festen Abständen die neuen Speisepläne aus seiner Datenquelle zieht.

Die Einstiegsklasse sieht damit in der Regel folgendermaßen aus:

Code 4.16: Spring Application Klasse

```

1  @EnableEurekaClient
2  @SpringBootApplication
3  @EnableScheduling
4  public class TestServiceApplication{
5      public static void main(String[] args) {
6          SpringApplication.run(TestServiceApplication.class,
7                               args);
8      }
9  }
```

## Controller

Die Klassen, die die Endpunkte verwalten und die Anfragen an die Services bearbeiten heißen in *Spring* CONTROLLER. Um der Autokonfiguration von *Spring Boot* zu symbolisieren, dass eine Klasse ein Controller ist, benötigt sie die Annotation `@RestController`. Damit der Service den Uniform Resource Locator (URL)-Pfad einer Anfrage zu einer solchen Klasse zuordnen kann, wird die Annotation `@RequestMapping([PFAD])` benötigt. Diese Klassen benötigen in der Regel noch Property Objekte, an die sie die eigentliche Bearbeitung der Anfrage delegieren können. Diese Klassen befinden sich in der Service-Schicht, enthalten die Business Logik und heißen dementsprechend Service Klassen. Damit *Spring Boot* diese Objekte als Java-Bean zur per Dependency Injection zur Verfügung stellen kann, müssen die privaten Instanzen mit der Annotation `@Autowired` versehen werden. Eine entsprechende Controller Klasse sieht dann folgendermaßen aus:

Code 4.17: Spring Controller Klasse

```
1 @RestController
2 @RequestMapping("/test")
3 public class TestController {
4
5     private final Service service;
6
7     @Autowired
8     public TestController(Service service){
9         this.service = service;
10    }
11 }
```

## Service

Die Business Logik der *Spring*-Anwendung liegt in der Regel in sogenannten *Service*-Klassen. Das ist auch in sämtlichen Microservices des Prototypen der neuen Hochschul-App der Fall. Damit das *Spring*-Framework diese Klassen erkennt und als Beans für die Controller zur Verfügung stellen kann müssen sie mit der Annotation `@SERVICE` markiert werden. Des weiteren benötigen diese Klassen, ähnlich wie die Controller Klasse, eine Instanz der Klasse aus der unter der Service-Schicht liegenden Persistenz-Klasse. Im Falle der Hochschul-App wird JPA verwendet, weshalb lediglich ein Verweis auf die benötigten Data Access Objects (DAOs) der Entity Klassen injiziert werden muss. Das geschieht wiederum mit der Annotation `@AUTOWIRED`. Die Service Klassen sehen in den Hochschul-App Microservices in der Regel wie folgt aus:

Code 4.18: Spring Service Klasse

```
1 @Service
2 public class TestServiceImpl implements TestService{
3
4     private final TestDAO dao;
5
6     @Autowired
7     public TestServiceImpl(TestDAO dao){
8         this.dao = dao;
9     }
10 }
```

## JPA

Die Tabellen, die in den Datenbanken der neuen Hochschul-Anwendung benötigt werden, werden bei der Verwendung von JPA als POJO-Klassen definiert, die man auch als *Business Entity* oder *DataObject* bezeichnet. Passend dazu müssen solche Klassen, um von JPA als Tabelle kannt zu werden, mit der Annotation `@ENTITY` markiert werden. Die Annotation `@TABLE(NAME=[TABLE_NAME])` legt dabei den Namen der Tabelle fest, die JPA in der Datenbank anlegen - oder falls schon vorhanden - verwenden soll. Jede Klasse, die als *Entity* markiert wurde, muss ein Klassenattribut besitzen, welches mit `@ID` annotiert wird. Alle Attribute der Klasse können zudem mit `@COLUMN(NAME=[COLUMN_NAME])` markiert werden, was den Spaltennamen des Attributs in der Datenbanktabelle festlegt. Des weiteren können komplexere Beziehungen mit anderen Annotations festgelegt werden. Diese können in der Dokumentation der JPA Schnittstelle nachgelesen werden.<sup>7</sup> Solche Entity Klassen können wie folgt aussehen:

Code 4.19: Spring Entity Klasse

```

1  @Entity
2  @Table(name="T_TEST_ENTITY")
3  @Getter
4  @Setter
5  @NoArgsConstructor
6  @AllArgsConstructor
7  public class TestEntity{
8
9      @Id
10     @Column(name="TEST_ID")
11     private Long id;
12
13     @Column(name="VALUE")
14     private String value;
15
16 }
```

## Sonstige

Wie bereits erwähnt zeichnet sich das *Spring*-Framework besonders durch seine starke Abstraktion komplexer Zusammenhänge aus. Dabei kann man zwar alle Zusammenhänge klassisch über Konfigurationsdateien - in der Regel Extensible Markup Language (XML) oder Property-Files - definieren, jedoch liegt die Stärke des

<sup>7</sup>Entities (2020).

Frameworks besonders in der annotationsbasierten Konfiguration. Zusätzliche Features, wie das *Swagger*-Framework, können in der Applikationsklasse per Annotation aktiviert werden. Ein Beispiel hierfür ist die erwähnte *Swagger* Annotation `@ENABLESWAGGER2`. Weitere funktionstragende Klassen, welche als Java Bean in die Applikation eingebunden werden sollen, werden ebenfalls annotiert. Beispiele hierfür sind `@COMPONENT` und `@CONFIGURATION`.

## Ressourcen

In *Spring Boot* gibt es, wie bereits erwähnt, mehrere Möglichkeiten, die Anwendung zu konfigurieren. Dabei wird im Prototypen der neuen Hochschul-App großenteils die annotationsbasierte Konfiguration verwendet. Dennoch gibt es einige Einstellungen, welche sich am einfachsten durch sogenannte Property Files einrichten lassen. Diese müssen für jeden Microservice einzeln angelegt werden und liegen in den Ressourcen der Anwendung. Sie werden standardmäßig *application.properties* benannt und enthalten in den Microservices der Hochschul-Anwendung mindestens folgende Einstellungen:

- **Name**

Unter dieser Eigenschaft wird der Name des Services festgelegt. Dieser ist vor allem bei der Registrierung beim Eureka Server und beim API-Gateway wichtig. Der vollständige Name der Property lautet `SPRING.APPLICATION.NAME`.

- **Port**

Diese Eigenschaft legt den Port fest, auf dem der embedded *Tomcat*-Server, der den Microservice hostet, hören soll. Der vollständige Name der Property lautet `SERVER.PORT`.

- **Eureka-Konfiguration**

Für die Registrierung des Eureka Clients beim Server müssen einige Einstellungen definiert werden. Die wichtigsten werden im Beispiel 4.20 gezeigt.

- **Log**

Bei einer Anwendung kann es oft sinnvoll sein, wichtige Ereignisse zu loggen. Damit diese im Nachhinein abgespeichert und ausgewertet werden können, gibt es dazu eine Sammlung von Properties. Die wichtigsten werden im Beispiel 4.20 gezeigt.



- **Datenbankverbindung**

Um eine Verbindung zu den Datenbank Servern aufbauen zu können, benötigt *Spring Boot* eine Reihe von Informationen. Diese beschreiben in der Regel die logische Adresse des Servers, den Nutzernamen, den die Anwendung als Datenbanknutzer zugewiesen bekommen hat und das passende Passwort für diese Verbindung. Des weiteren benötigt die Anwendung dann noch Details zum Namen der genauen Datenbank, mit der sie sich verbinden soll, die Art der Installation des Servers und eventuell auch Informationen dazu, wie sich JPA bei einer Verbindung mit dem Datenbank Server verhalten soll. Da es hierbei zahlreiche Einstellungsmöglichkeiten gibt und in einer Properties-Datei mehrere Datenbanken eingebunden werden können, wird im Beispiel 4.20 exemplarisch die Datenbank des *Lx-Lehre* Servers eingebunden. Hierbei wurden sensible Daten jedoch anonymisiert.

Das folgende Beispiel orientiert sich an den Einstellungen in den *application.properties* des *Timetable*-Microservices. Hierbei wurden aus Gründen des Leseflusses allerdings nicht alle Properties aufgenommen. Die sensiblen und sicherheitstechnisch relevanten Attribute wurden anonymisiert.

Code 4.20: Typische *application.properties* Datei

```

1  spring.application.name=timetable-service
2  server.port=8080
3
4  eureka.client.register-with-eureka=true
5  eureka.client.fetchRegistry=true
6  eureka.client.serviceUrl.defaultZone=
7      http://localhost:8081/eureka/
8
9  logging.level.org.springframework.web=WARN
10 logging.level.org.hibernate=ERROR
11 logging.file.name = ./logs/timetable-service.log
12
13 spring.datasource.lx-lehre.jdbc-url=
14     jdbc:mysql://localhost:60012/default_database
15 spring.datasource.lx-lehre.username=default_user
16 spring.datasource.lx-lehre.password=5AF3_PA55WORD
17 spring.datasource.lx-lehre.driver-class-name=
18     com.mysql.jdbc.Driver
19 spring.datasource.lx-lehre.jpa.hibernate.hbm2ddl.auto=none
20 spring.datasource.lx-lehre.jpa.show-sql=true

```

## 4.3 Spring Cloud Gateway

Das Framework *Spring Cloud Gateway* basiert auf den Frameworks *Spring 5*, *Project Reactor* und *Spring Boot 2.0* und bietet eine breite Auswahl an Einstellungsmöglichkeiten. Es können Filter und Routen für spezielle Anfragen definiert werden, Discovery Clients eingebunden werden, Verschlüsselungen konfiguriert und Cross-Origin Resource Sharing (CORS)-Einstellungen definiert werden. Von all diesen Funktionen ist für den Prototypen der neuen Hochschul-App jedoch nur die Konfiguration von CORS und Routen zu den Microservices relevant.

### Routen Konfiguration

Um die Anfragen der Clients an die richtigen Microservices weiterleiten zu können, müssen für jeden Client Routen definiert werden. Hierfür bietet das *Spring Cloud Gateway*-Framework zwei Möglichkeiten.

**Möglichkeit 1:** Statische Routen Definition über Property- oder YAML Ain't [sic]! Markup Language (ursprünglich *Yet Another Markup Language*) (YAML)-Files.

Code 4.21: Gateway *application.yml* Datei

```

1  spring:
2    cloud:
3      gateway:
4        routes:
5          - id: mensa-service
6            uri: lb://MENSA-SERVICE
7            - Path=/api/mensa-service/v1/**
8              filters:
9                - RewritePath=/api/mensa-service/v1/(?<resource>.*),
                  /${resource}

```

Die se Möglichkeit ist vor allem sinnvoll, wenn nur wenige Routen definiert werden müssen, die sich nicht ändern. Für komplexere Konfigurationen eignet sie sich jedoch nicht. Deshalb wurde im Prototypen der Hochschul-App die sogenannte *Fluent-API* verwendet.

**Möglichkeit 2:** Dynamische Routen-Definition (Fluent-API) über die Einbindung eines Beans vom Typ `ROUTELOCATOR`. Die Einbindung dieser Bean kann in der *Spring Application* Klasse erfolgen, da es sich aber um eine Konfiguration handelt

wurde der *Best Practice* Vorsatz einer @CONFIGURATION-Klasse genutzt.

Code 4.22: Gateway Routen Konfiguration Klasse

```

1  @Configuration
2  public class RouteConfig {
3
4      @Bean
5      public RouteLocator proxyRoutes(RouteLocatorBuilder
        routeLocatorBuilder){
6          return routeLocatorBuilder.routes()
7              .route(route ->
8                  route.path("/api/mensa-service/v1/**")
9                      .filters(filter -> filter
10                          .rewritePath(
11                              "/api/mensa-service/v1/(?<resource>.*)",
12                              "/${resource}")
13                          .uri("lb://MENSA-SERVICE")
14                          .id("mensa-service"))
15                      .build());

```

Hierbei zu beachten ist die eingebundene URL in den Codebeispielen 4.21 und 4.22. Hier werden die Vorteile des Eureka Servers voll ausgenutzt. Hierfür wird in der *application-properties* lediglich der Pfad zum Eureka Server eingetragen, worauf das *Spring Cloud Gateway*-Framework die benötigten Servicepfade selbstständig beziehen kann.

## Cross-Origin-Filter Konfiguration

Da die REST-Schnittstelle und die Clients dieser API meist auf unterschiedlichen Quellen laufen und viele Clients ihre Anfragen über einen Browser durch CORS realisieren, scheitern jegliche Anfragen in der Regel an mangelnden Einstellungen zur Sicherheit bei der Nutzung mit CORS. Da diese anfällig für Angriffe sind, blockieren die gängigen Browser solche Anfragen in der Regel. Deshalb muss das CORS im API-Gateway uzerst konfiguriert werden, dies erfolgt wie auch bei den Routen entweder über die Poperties- oder YAML-Files oder über die Bereitstellung von Configuration-Beans. Die bereitgestellte BEan muss zusätzlich zur @CONFIGURATION Annotation noch mit @ENABLEWEBFLUX markiert werden. *WebFlux* ist ein Framework von *Spring* und bietet eine reaktive Programmierunterstützung für Webanwendungen.

Code 4.23: CORS-Filter Konfigurationsklasse

```

1  @Configuration
2  @EnableWebFlux
3  public class CORSFilterConfig implements WebFluxConfigurer
4  {
5
6      @Bean
7      public CorsWebFilter corsWebFilter() {
8          CorsConfiguration corsConfiguration = new
9              CorsConfiguration();
10         corsConfiguration.addAllowedMethod("*");
11         corsConfiguration.addAllowedOrigin("*");
12         corsConfiguration.addAllowedHeader("*");
13         corsConfiguration.setAllowCredentials(true);
14
15         UrlBasedCorsConfigurationSource
16             urlBasedCorsConfigurationSource = new
17                 UrlBasedCorsConfigurationSource();
18         urlBasedCorsConfigurationSource
19             .registerCorsConfiguration("/**",
20                 corsConfiguration);
21
22         return new
23             CorsWebFilter(urlBasedCorsConfigurationSource);
24     }
25
26     @Override
27     public void addCorsMappings(CorsRegistry corsRegistry){
28         corsRegistry.addMapping("/")
29             .allowedMethods("*")
30             .allowedOrigins("*")
31             .allowedHeaders("*")
32             .allowCredentials(true);
33     }
34 }

```

## 4.4 Eureka

Um die bereits erwähnten Vorteile der Registry und des Discovery Services aus Kapitel 3.6 nutzen zu können, müssen die Eureka Clients und die Eureka Server zunächst konfiguriert werden. Die Integration der Abhängigkeiten wurde bereits in Kapitel 4.2 beschrieben und die grundlegenden Konfigurationen des Eureka Clients

in Kapitel 4.2 und 4.2. Dennoch sollen diese Einstellungen nochmals kurz beleuchtet werden.

## Eureka Client

Im folgenden werden die wichtigsten Einstellungen für die Registrierung beim Eureka-Server in der *application.properties*-Datei genauer erläutert. Der Name des Services spielt eine wichtige Rolle, dieser wird beim Eureka Server als Identifikation verwendet.

Code 4.24: Spring Name in *application.properties* Datei

```
1 spring.application.name=eureka-client-service-id
```

Wird ein weiterer Eureka Client mit dem selben Namen definiert, so erkennt der Eureka Server, dass es sich bei den beiden Services um eine Duplizierung handelt und übernimmt das Load Balancing zwischen diesen. Die weiteren Properties sollen nochmals genauer erläutert werden.

Code 4.25: Eureka Client Einstellungen aus *application.properties* Datei

```
1 eureka.client.register-with-eureka=true
2 eureka.client.fetchRegistry=true
3 eureka.client.serviceUrl.defaultZone=
4     http://localhost:8081/eureka/
```

Das Flag *register-with-eureka* aktiviert die Registrierung des Clients mit Server. Durch *fetchRegistry* werden die Registrations-Informationen vom Eureka Server geladen und lokal im Cache des Client abgespeichert. Diese Informationen werden in einem festgelegten Zeitraum aktualisiert. Dieser kann durch die Einstellung *registryFetchIntervalSeconds* manuell geändert werden. Das lokale cachen der Informationen bietet den Vorteil, dass die Clients weiterhin miteinander kommunizieren können, auch wenn keine Instanz eines Eureka Servers mehr verfügbar ist. Die *defaultZone* beschreibt die physische Adresse des Eureka Servers.

## Eureka Server

Die Funktionen des Eureka Servers wurden bereits mehrmals erläutert. Deshalb sollen hier nur kurz die nötigen Konfigurationen dargestellt werden. Zuerst muss die `@SPRINGBOOTAPPLICATION`-Klasse im Hochschul-App Discovery-Service als Eureka-Server annotiert werden. Dies kann mithilfe der `@ENABLEEUREKA-SERVER` Annotation durchgeführt werden.

Code 4.26: Service als Eureka Server kennzeichnen

```

1  @EnableEurekaServer
2  @SpringBootApplication
3  public class DiscoveryServiceApplication {
4
5      public static void main(String[] args) {
6          SpringApplication
7              .run(DiscoveryServiceApplication.class, args);
8      }
9
10 }
```

Des weiteren muss die *application.properties* Datei Konfiguriert werden. Hierfür wird auf das Kapitel 4.2 verwiesen.

Es müssen wieder die gleichen Properties wie bei einem Eureka Client eingestellt werden, mit dem Unterschied, dass es sich hierbei um einen Eureka Server handelt und somit die Registrierung bei einem Server und das Laden von Informationen der registrierten Clients deaktiviert werden. Dies ist in diesem konkreten Anwendungsfall von Nöten, da es nur einen Eureka Server in der gesamten Anwendung gibt. Bei einer gewünschten Redundanz dessen ist eine andere Konfiguration nötig. Die EUREKA.SERVER Einstellungen sind für die Minimierung der Startzeit des Servers nötig.

## 4.5 API Filter

Da bei REST-Services üblicherweise Daten angeboten werden, die man in Form von Listen abrufen kann, bietet es sich oft an, einer Anfrage Filterkriterien anzufügen, nach denen die Liste schon auf dem Server gefiltert wird. So spart sich ein leichtgewichtiger Client die wertvollen Ressourcen, die bei solch einer Aktion nötig sind. Dies ist auch beim Prototypen der neuen Hochschul-App der Fall. Daten wie Vorlesungen, Speisen und Stundenplanänderungen können im Ganzen vom Server abgerufen werden. Zusätzlich sind sie noch in Kategorien unterteilt, über die die Suche anhand des Ressourcenpfades verfeinert werden kann. Dennoch gibt es Attribute in diesen Daten, die nicht in eine solche Kategorie fallen. Typischerweise sind das Zeitangaben oder sehr spezielle Eigenschaften einer Ressource. Um dennoch nach diesen Attributen filtern zu können, werden solche Filter als sogenannte *Queryparamter* übergeben. Diese werden der Endpunkt-URL angefügt. Typischer-

weise sieht eine Anfrage mit solchen *Queryparametern* wie folgt aus:

Code 4.27: Typischer Anfragepfad mit Filtern

```
1 https://server.com/pfad?filter=wert&datumfilter=20-12-2018
```

In *Spring Boot* können solche Parameter in der Endpunkt-Methode über die Annotation `@RequestParam` abgerufen werden. Die obere Methode würde dann folgendermaßen aussehen:

Code 4.28: Typischer Endpunkt mit `@RequestParam`

```
1 @GetMapping("/pfad")
2 public Response testMethod(
3     @RequestParam(name="filter", required=false)
4     String filter,
5     @RequestParam(name="date", required=false) Date
6     date){
7     return Response
8         .ok(service.getData(filter, date))
9         .build();
10 }
```

Das hat den Vorteil, dass man Typ-sichere Parameter erhält, ohne sie selbst aus der Anfrage URL auslesen zu müssen. Dennoch hat das den Nachteil, dass man Filterparameter so für jede Anfrage einzeln definieren muss. Zudem ist diese Herangehensweise sehr statisch und sehr aufwändig bei einer Änderung. Deshalb wurde für die Microservices der Hochschulanwendung eine andere Lösung implementiert.

Für die implementierte Lösung muss die Klasse `FILTERTO.CLASS` erstellt werden. Sie sieht folgendermaßen aus:

Code 4.29: Aufbau des Filter Objekts

```
1 @Setter
2 @AllArgsConstructor
3 public class FilterTO {
4
5     protected Map<String, String> filters;
6
7     public Map<String, String> getFilters() {
8         return Collections.unmodifiableMap(filters);
9     }
10 }
```

Zu Beachten ist hier die verwendete MAP, welche dafür da ist, die Namen der Filterparameter auf ihren Wert zu mappen. Die `FILTERTO` Instanz wird bei jeder Anfrage

in der folgenden Klasse erstellt:

Code 4.30: Filterparser

```

1  @ControllerAdvice
2  public class FilterParser {
3
4      private final HttpServletRequest request;
5
6      @Autowired
7      public TimetableFilterParser(HttpServletRequest
          request) {
8          this.request = request;
9      }
10
11     @ModelAttribute
12     public void addAttributes(Model model) {
13         model.addAttribute("filter", getFilterTO());
14     }
15
16     private FilterTO getFilterTO(){
17         Map<String, String> filters = new HashMap<>();
18         Collections.list(request.getParameterNames())
19             .forEach(name->{
20             String value =
21                 request.getParameter(name);
22             value = value
23                 .replaceAll("\\\"", "\"");
24             filters.put(name, value);
25             });
26         return new FilterTO(filters);
27     }
28 }

```

Die Annotation `@CONTROLLERADVICE` bewirkt, dass die Controller-Klassen, die darin explizit referenziert werden, die Möglichkeit haben, das Model der `ADDATTRIBUTES()`-Methode in ihren Anfragen zu injizieren. Dieses Model bekommt ein `FILTERTO` Objekt, welches in seiner Map die Query Parameter enthält. Diese sind jedoch noch nicht Typ-sicher, sondern lediglich als String hinterlegt. Zudem sind in dieser Map alle Parameter, auch die, die die Controller eventuell nicht akzeptieren. Die Controller benötigen in ihren Endpunkt-Methoden folgende Definition, um auf den Filter zugreifen zu können:



Code 4.31: Typischer Endpunkt mit eigener Lösung

```

1 @GetMapping("/pfad")
2 public Response testMethod(@ModelAttribute("filter")
   FilterT0 filter){
3     return Response
4         .ok(service.getData(new
           DataFilterT0(filter)))
5         .build();
6 }

```

Wie zu erkennen ist, ist nirgends definiert, welche Filter der Endpunkt akzeptiert. Jedoch wird an den ausführenden Service nicht das eigentliche FILTERTO weitergegeben, sondern die Klasse DATAFILTERTO.CLASS. Diese ist wie folgt aufgebaut:

Code 4.32: Ressourcenspezifisches Filter Objekt

```

1 @Getter
2 public class DataFilterT0 extends FilterT0{
3
4     private final I18nResourceBundle I18N = new
       I18nResourceBundle();
5
6     private String filter;
7     private Date date;
8
9     public LectureFilterT0(FilterT0 filterT0){
10         super(filterT0.filters);
11         this.setFilter(this.filters.get("filter"));
12         this.setDate(this.filters.get("date"));
13     }
14
15     public void setFilter(String filter){
16         if(I18N.validateContent(filter))
17             this.filter = filter;
18     }
19
20     public void setDate(String date){
21         SimpleDateFormat format = new
           SimpleDateFormat(Constants.DATE_FORMAT);
22         try{
23             this.date = format.parse(date);
24         }catch(ParseException pE){
25             this.date = null;
26         }
27     }

```

In dieser Klasse werden die rohen Filterdaten gesammelt und die für die angefragte Ressource erlaubten Filter in ihrem korrekten Typ extrahiert. Dies geschieht in den zu den Klassenattributen gehörenden Settern. Die Klasse `I18NRESOURCEBUNDLE.CLASS` regelt in diesem Fall exemplarisch den Test, ob der übergebene Filter in einer der von der Anwendung unterstützten Sprachen korrekt ist. Das Datum wird ebenfalls nur dann gesetzt, wenn es im richtigen Format übergeben wurde. `CONSTANTS.CLASS` ist hierbei eine globale Sammlung von Konstanten. Nach der Erstellung dieser Klasse sind nur noch Filter relevant, die die Klasse `DATAFILTERTO.CLASS` auch akzeptiert. Der Service, der vom Endpunkt aufgerufen wurde, verarbeitet die Filter dann folgendermaßen:

Code 4.33: Serviceschicht

```

1 public DataT0 getData(DataFilterT0 filter){
2     return testDao.findAll(new FilterSpecification(filter))
3         .stream()
4         .map(DataD0::createT0)
5         .collect(Collectors.toList());
6 }

```

Die Methode `FINDALL()` ist eine Methode des *Spring Data*-Frameworks, welche Instanzen der Klasse `SPECIFICATION.CLASS` zum Verfeinern der automatisch generierten Datenbankabfrage akzeptiert. Auch hier ist anzumerken, dass nach wie vor nicht kontrolliert wurde, welche Filter für diese Abfrage gesetzt wurden. Das geschieht in der folgenden Klasse:

Code 4.34: Spezifikation der Filter für Datenbankabfrage

```

1 public class FilterSpecification implements
    Specification<DataD0> {
2
3     private final DataFilterT0 FILTER;
4     private final Set<Specification<DataD0>>
        SPECIFICATIONS;
5
6     public FilterSpecification(DataFilterT0
        filter) {
7         this.FILTER = filter;
8         this.SPECIFICATIONS = new HashSet<>();
9     }
10
11     @Override
12     public Predicate toPredicate(Root<LectureD0> root,
        CriteriaQuery<?> criteriaQuery, CriteriaBuilder

```

```

        criteriaBuilder) {
13         Specification<DataDO> spec = ...; /*Hier ist die
           Instanziierung abhaengig vom Service*/
14         if(FILTER.getFilter() != null)
15             spec = spec.and(this.getFilterSpecification());
16         if(FILTER.getDate() != null)
17             spec = spec.and(this.getDateSpecification());
18         return spec.toPredicate(root, criteriaQuery,
           criteriaBuilder);
19     }
20
21     /*Implementierung der get...Specification()-Methoden
           unterschiedlich*/
22 }

```

Das Interface `TESTDAO` des Typs `JPARepository<DataDO, Long>`, `JPA SpecificationExecutor<DataDO>` liefert dann nur noch die Daten zurück, die den Filtern, die in die Datenbankabfrage durch die hinzugefügten Spezifikationen angefügt wurden, entsprechen. Diese Lösung ist zwar anfangs deutlich aufwändiger zu implementieren, als die Nutzung von `@RequestParam`, jedoch ist sie genau auf die Bedürfnisse der Schichtenarchitektur in Verbindung mit der Nutzung von JPA angepasst und ist nach der initialen Implementierung deutlich flexibler. Neue Filter können also jederzeit hinzugefügt werden und werden so bis in die Datenbankschicht übernommen.

## 4.6 Error Handling

Wie bei vielen weiteren Funktionen bietet *Spring Boot* ebenfalls Lösungsansätze für das Error-Handling, speziell beim Abfangen von Exceptions im Code und der Rückgabe der richtigen Status-Informationen an den aufrufenden Client. Die standardmäßig verwendeten Funktionen von *Spring* sind hier jedoch nicht ausreichend, sie bieten dem Client zu viele Informationen darüber, was den Fehler ausgelöst hat. Um dies abzufangen werden im Prototypen der Hochschul-App die Annotations `@ControllerAdvice` und `@ExceptionHandler` verwendet. Dies sieht wie folgt aus:

Code 4.35: Error Handling

```

1  @ControllerAdvice
2  public class ExceptionHandler {
3
4      @ExceptionHandler(Throwable.class)
5      public ResponseEntity<HttpReturnErrorPattern>
6          handleUncaughtExceptions(Throwable e){
7          LOGGER.warn(e.getMessage(), e);
8          return handleServerException(new
9              ServerException(e.getMessage()));
10     }
11
12     @ExceptionHandler(BaseException.class)
13     public ResponseEntity<HttpReturnErrorPattern>
14         handleServerException(BaseException e){
15         LOGGER.warn(e.getMessage(), e);
16         return e.toResponse();
17     }
18 }

```

Zu erkennen ist, dass in der Klasse EXCEPTIONHANDLER lediglich zwei Methoden vorhanden sind. Diese Methoden sind jeweils mit der Annotation @EXCEPTIONHANDLER markiert. Diese Annotation benötigt als Parameter die Klasse des *throwables*, die abgefangen werden soll. Konkret wird in der Methode HANDLESERVEREXCEPTION eine eigens entwickelte Exception abgefangen, welche sich um Informationsfluss nach außen, Internationalisierung der Antworten und Formattierung der Fehler kümmert. Die Methode HANDLEUNCAUGHTEXCEPTIONS ist lediglich eine Methode, die alle Exceptions abfängt, die im Code unerwartet geworfen werden. Diese Exceptions werden anonymisiert und die Bearbeitung wird an die HANDLESERVEREXCEPTION Methode delegiert.

Diese Konfiguration ermöglicht es, alle Exceptions abzufangen und intern zu verarbeiten. Alle benötigten Infos können geloggt und entsprechend verarbeitet werden. Der Client sieht immer nur die Informationen, die er sehen darf, was mögliche Sicherheitslücken vermindert. Zudem wird nie eine Exception geworfen, die den Server zum Absturz bringt.


## 4.7 Firebase Cloud Message

FCM ist ein Cloud Messaging Service, der von *Google* entwickelt wurde. Er stellt die Grundlage für den Notification Service dar und wird deshalb hier explizit nochmals genauer betrachtet. Für die Nutzung des FCM-Services wird zunächst ein Google Account benötigt. Für den Prototypen der neuen Hochschul-App wurde die E-Mail-Adresse *hochschul-app-fh@gmx.de* erstellt und für den benötigten und kostenlosen Google Account verwendet. Mit dieser Mail-Adresse kann nun ein Firebase-Projekt erstellt werden. Dafür muss die Website des Services aufgerufen werden.<sup>8</sup> Nun muss für die Konfiguration der Kommunikation zwischen dem Notification Server der Hochschul-App und dem Firebase-Cloud Server noch ein sogenanntes *Cloud Messaging*-Schlüsselpaar erstellt werden. Dieses wird danach für die *Web-Push*-Zertifikate genutzt und soll den Notifikations-Client - in diesem Fall den Notification Service der Hochschul-App - eindeutig identifizieren.

Webkonfiguration

Web-Push-Zertifikate

Web-Push-Zertifikate

Firebase Cloud Messaging kann mithilfe von Anwendungsidentitäts-Schlüsselpaaren eine Verbindung zu externen Push-Diensten herstellen. [Mehr erfahren](#) 

Schlüsselpaar	Hinzugefügt am
BE-D1mABEU4RQKAUm9E-AShcDLAUNh1KOID35zdUWbNrSjlrNel6GYXTT_SjLka1ul0PJ7OMtN7iJljq3BkiKyc	29.12.2019

Abbildung 4.12: Firebase Webkonfiguration

Für die Nutzung der FCM-Funktionalitäten im Notification-Service der Hochschul-App muss unter den Dienstknoten zuerst ein neuer privater Schlüssel erstellt werden. Dies sieht folgendermaßen aus:

<sup>8</sup>Firebase console (2020).

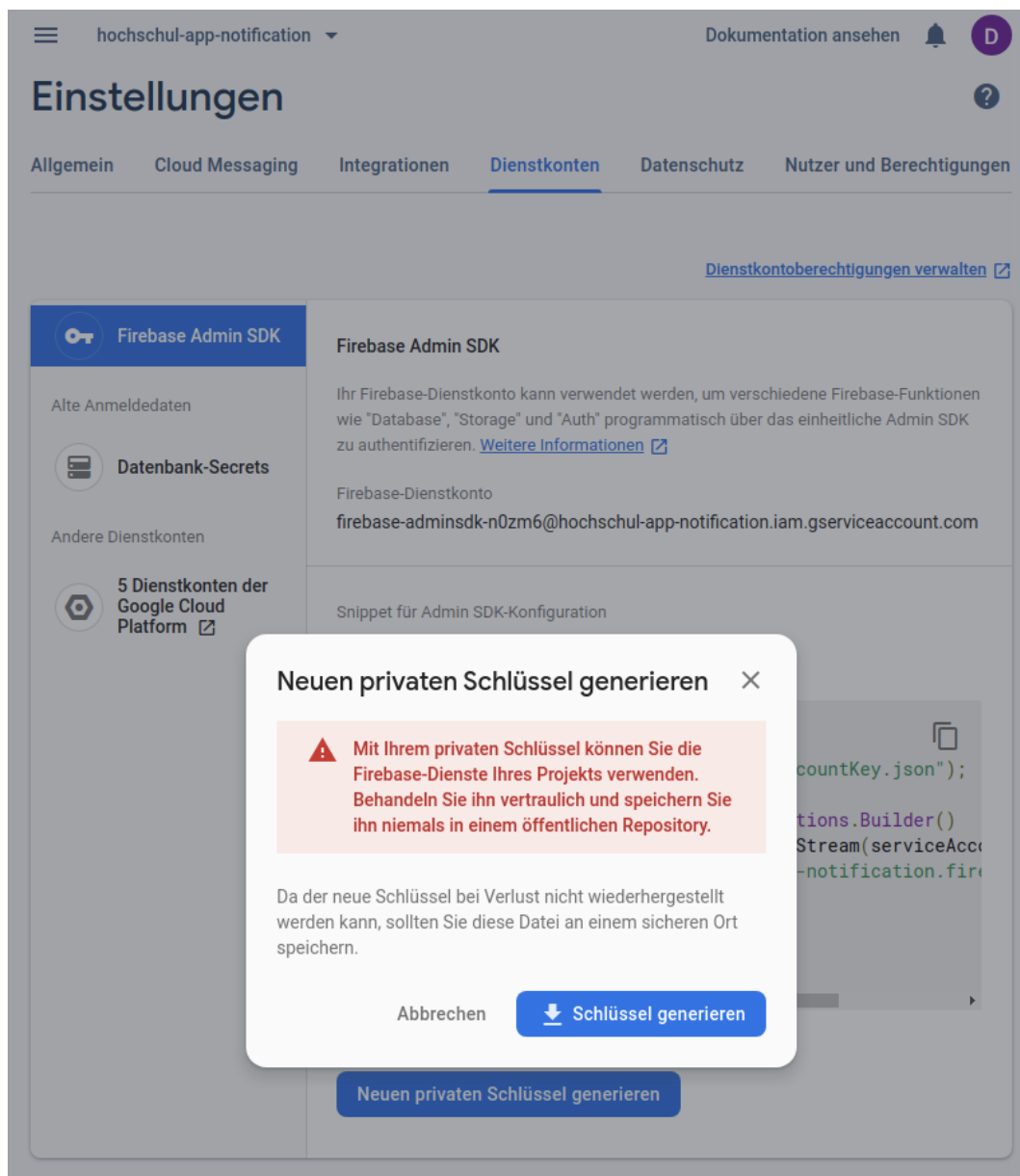


Abbildung 4.13: Firebase Admin SDK Authentifizierung

Nach diesem Schritt wird vom Firebase-Service ein Javascript Object Notation (JSON) generiert, welches alle relevanten Firebase-Service-Account Informationen wie Projekt-ID, Private-Key, Client-ID, Authentifikations-URL, Tokens und die URLs der Zertifikate enthält. Das komplette JSON-File ist folgendermaßen aufgebaut:

Code 4.36: Firebase Service Account JSON

```

1  {
2      "type": "service_account",
3      "project_id": "hochschul-app",
4      "private_key_id": "5ECUR3",
5      "private_key": "-BEGIN PRIVATE KEY--END PRIVATE KEY-",
6      "client_email": "firebase-adminsdk@hochschul-app.com",
7      "client_id": "12345678",
8      "auth_uri": "https://accounts.google.com/o/oauth2/auth",
9      "token_uri": "https://oauth2.googleapis.com/token",
10     "auth_provider_x509_cert_url":
11         "https://www.google.com/certs",
12     "client_x509_cert_url":
13         "https://www.google.com/hochschul-app.com"
14 }

```

Dieses File muss dann im Notification-Service eingebunden werden. Dafür wird im *application.properties*-File der Pfad des abgelegten JSON definiert. Es wird dabei folgende Zeile angefügt:

Code 4.37: Firebase Service Account application.properties Datei

```

1  fcm.firebase-config-file=${PFAD}/firebase-adminsdk.json
2  }

```

Im Package CONFIG des Source Codes des Notification-Service wird eine Klasse erstellt, die mit der Annotation @CONFIGURATIONPROPERTIES markiert wurde und die Inhalte der FCM-Properties in eine Variable schreibt.

Code 4.38: Firebase Service Account Variable

```

1  @Setter
2  @Getter
3  @ConfigurationProperties(prefix = "fcm")
4  public class FcmProperties {
5
6      private String firebaseConfigFile;
7
8  }

```

Zudem wird ein Bean der Konfigurationsklasse definiert und in den Notification Service eingebunden.

Code 4.39: Firebase Service Account Bean

```

1  @Configuration
2  @EnableConfigurationProperties(FcmProperties.class)
3  public class FcmPropertiesConfig {
4
5      @Bean
6      public FcmProperties fcmConfig () {
7          return new FcmProperties();
8      }
9
10 }
```

Mithilfe des Beans kann die Verbindung zur FCM-Cloud dann folgendermaßen aufgebaut werden:

Code 4.40: Firebase Service Account Bean

```

1  Path path = Paths.get(fcmPropertiesConfig.fcmConfig()
2      .getFirebaseConfigFile());
3  InputStream inputStream = Files.newInputStream(path);
4  FirebaseOptions options = new FirebaseOptions
5      .Builder()
6      .setCredentials(
7          GoogleCredentials
8              .fromStream(inputStream)
9      ).build();
```

Nach dem Verbindungsaufbau können im Notification Service Clients registriert sowie informiert werden. FCM bietet zusätzlich die Möglichkeit der Erstellung von *Topics*. Diese sind organisatorische Einheiten, in die Benachrichtigungen eingeteilt werden können. Somit muss sich ein Client nicht für alle Benachrichtigungen registrieren, sondern kann diese nach der Relevanz für seinen Anwendungsfall filtern.



Code 4.41: Firebase Service Account Bean

```
1 //build message with topic
2 Message message = Message.builder()
3     .putAllData(messageT0.getData())
4     .setTopic(topic)
5     .setWebpushConfig(WebpushConfig.builder().putHeader("ttl",
6         "300")
7         .setNotification(new
8             WebpushNotification(messageT0.getTitle(),
9                 messageT0.getBody(), icon))
10        .build())
11    .build();
12
13 //subscribe to topic
14 TopicManagementResponse response =
15     FirebaseMessaging.getInstance()
16         .subscribeToTopicAsync(
17             Collections.singletonList(registrationToken),
18             topic
19         ).get();
20
21 //unsubscribe from topic
22 TopicManagementResponse response =
23     FirebaseMessaging.getInstance()
24         .unsubscribeFromTopicAsync(
25             Collections.singletonList(registrationToken),
26             topic
27         ).get();
28
29 //send message
30 FirebaseMessaging.getInstance().sendAsync(message).get();
```

## 5 Entwicklungsumgebungen

Wie in allen größeren Software Projekten ist es in der Regel üblich, zwei verschiedene Einrichtungen für die Anwendung in den verschiedenen Phasen ihrer Existenz zu verwenden. Dabei unterscheidet man in der Regel von der Entwicklungsumgebung in der Entwicklungsphase und der Umgebung in der Produktionsphase. Passend zu ihren Einsatzzwecken werden diese Umgebungen jeweils *dev* und *prod* genannt. Auch der Prototyp der Hochschul Anwendung verfügt über diese beiden Konfigurationen. Warum diese benötigt werden und wie sie im Detail aufgebaut sind, beziehungsweise was sie genau unterscheidet, wird im folgenden genauer betrachtet.

### 5.1 Dev-Environment

Die Umgebung, die den Entwicklern beim Arbeiten an der Anwendung zur Verfügung steht, zeichnet sich vor allem in drei Punkten aus: der Datenbankanbindung, der Kommunikation innerhalb der Services und des Starts der Anwendung. Diese drei Punkte wurden einerseits so angepasst, dass der Entwickler einfacher Änderungen vornehmen kann, aber auch so, dass den Anbindungen der nur im Produktionsmodus der Anwendung benötigten Ressourcen nicht durch versehentliche Fehler geschädigt werden. Die genauen Eigenschaften werden nun kurz erläutert.

#### Datenbankanbindung

Die Datenanbindung der Hochschul-App benötigt Zugriffe auf verschiedene Datenbanken. Einerseits werden Datenbanken angelegt, die für den internen Gebrauch der Services benötigt werden. Dazu gehören die Datenbanken für die Mensa Daten, Stundenplaninformationen aber auch die User-spezifischen Daten. Andererseits werden ebenfalls die Daten des *LxLehre*-Servers der Hochschule benötigt. Dieser enthält die aktuellen Stundenplan-Daten und ist aktuell noch als Proxy-Datenbank zwischen die Services und die Datenbank der Hochschule Hof geschaltet. Beide Arten von Datenbanken benötigen eine gültige *MySQL* Installation, auf der sie laufen können. Würde dies auch im Entwicklungszeitraum benötigt werden, würden sich

zwei Alternativen anbieten. Entweder, jeder Entwickler installiert MySQL auf seinem PC und richtet die Datenbanken als perfekte Abbilder zu ihren Originalen auf den Hochschul-Servern ein oder es werden die echten Datenbanken eingebunden. Die erste Lösung bietet keine flexiblen Änderungen an der Datenbank und erfordert, dass die Datenbanken auf jedem System einzeln installiert werden. Die zweite Möglichkeit sollte gar nicht erst in Betracht gezogen werden, da jeder Fehler der Entwickler sich sofort auf die originalen Daten auswirkt.

Deshalb wurde in der *dev*-Entwicklungsumgebung die Datenbank *H2* eingebunden. Diese Datenbank ist eine reine In-Memory Datenbank, welche ihre Konfiguration aus einer statischen Datei mit Testdaten sammelt. Diese sind eine Spiegelung der originalen Daten. Beim Start der Anwendung wird hierbei jedes Mal die *H2* Datenbank neu initialisiert, wobei Fehler aus alten Starts der Anwendungen nicht beachtet werden. Diese *H2* Datenbank enthält alle Tabellen, die vom System benötigt werden, weshalb die Einbindung mehrerer Datenquellen nicht mehr nötig ist. Des Weiteren muss der Entwickler keine zusätzliche Software auf seinem System installieren, da die *H2* Datenbank über eine Java-Bibliothek läuft. Wie man es aus anderen Database Management Systems (DBMS) kennt, gibt es für die *H2* Datenbank eine grafische Benutzeroberfläche, über welche die Daten eingesehen und mit Standard-Structured Query Language (SQL) abgeändert werden können. Diese ist mit jedem Browser unter folgender URL erreichbar:

Code 5.1: H2-Console

```
1 https://localhost:[port-des-services]/h2-console
```

Über diese URL gelangt man - insofern der Service im *dev*-Modus gestartet wurde - auf folgende Oberfläche:

The screenshot shows the H2-Console Login window. At the top, there is a language dropdown set to 'English' and menu items for 'Preferences', 'Tools', and 'Help'. The main area is titled 'Login' and contains several input fields and buttons. The 'Saved Settings' dropdown is set to 'Generic H2 (Embedded)'. Below it, the 'Setting Name' field also contains 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons to its right. A horizontal separator line follows. The 'Driver Class' field contains 'org.h2.Driver'. The 'JDBC URL' field contains 'jdbc:h2:mem:testdb'. The 'User Name' field contains 'sa'. The 'Password' field is masked with dots. At the bottom, there are 'Connect' and 'Test Connection' buttons.

Abbildung 5.1: Login der H2-Console

Der Login erfolgt über den Nutzernamen und das Passwort, die in der *application-dev.properties*-Datei hinterlegt wurden. Nach dem Login wird dann ein Textfeld angezeigt, in dem SQL-Satements auf alle in der linken Seite der Oberfläche angezeigten Tabellen ausgeführt werden können.

### Microservice Kommunikation

Anders als im *prod*-Modus einer Anwendung, können alle Teile der Anwendungen, darunter auch die Microservices, individuell gestartet werden. Dabei sind keine weiteren Services nötig, alle Abhängigkeiten werden in der Test Umgebung simuliert. Somit müssen bei Abhängigkeiten zwischen den Microservices keine anderen Services gestartet werden. Da dies in einer Microservice Architektur im allgemeinen nicht vorgesehen ist, sind hierbei vor allem das API-Gateway und der *Eureka*-Server zu betrachten. In der Entwicklungsumgebung muss zuerst der *Eureka*-Server gestartet werden, worauf das API-Gateway sich dann bei seinem Start beim *Eureka*-Server registrieren kann. Danach können alle anderen Services gestartet werden. Im *dev*-Modus ist das nicht der Fall, hier sind Gateway- und Service-Discovery-Funktionen deaktiviert. Dies geschieht über Konfigurationen in der *application-dev.properties*-Datei, die benötigten Eigenschaften sehen wie folgt aus:

## Code 5.2: Discovery und Gateway Deaktivierung

```
1 eureka.client.enabled=false
2 eureka.client.register-with-eureka=false
```

**Start der Anwendung**

Der wohl gravierendste Unterschied zwischen der produktiven und der Entwicklungsumgebung liegt im Start der Anwendung. Möchte man einen Teil der Anwendung starten, um ihn in der *dev*-Umgebung zu testen, so benötigt man dazu, wie bereits erwähnt, keine Instanzen des *Eureka*-Servers oder des API-Gateways. Man stößt lediglich die Kompilierung und den Build-Prozess des Services an und startet die danach generierte *\*.jar*-Anwendung mit dem Konsolenbefehl `JAVA -JAR [DATEINAME].JAR`.

Um dies Prozesse jedoch nicht manuell anstoßen zu müssen sind die im Kapitel 4.1 beschriebenen Tools hilfreich. Durch die IDE kann der Programmcode automatisch kompiliert, gebuildet und gestartet werden. Das alles sogar im sogenannten *Debug*-Modus der IDE, welcher es ermöglicht, Breakpoints im Code zu setzen, bei deren Durchlaufen das Programm gestoppt wird, um die Variablen während der Laufzeit auslesen zu können. In der in dieser Praxisarbeit empfohlenen IDE sind die Buttons zum Start und zum Debuggen der Anwendung rechts oben in der Oberfläche zu finden. Diese sehen wie folgt aus:



Abbildung 5.2: Start und Debuggen der Anwendung in der IDE

Der Linke der markierten Buttons ist zum Start der Anwendung im normalen Modus zuständig. Dabei ist kein Debugging möglich, es werden beim Ausführen der Anwendung jedoch auch weniger Ressourcen des Systems benötigt. Der Rechte der beiden Buttons ist der Debug Button. Durch diesen können im Code Breakpoints gesetzt werden, die die Anwendung auch stoppen. Dieser Modus benötigt jedoch mehr Systemressourcen. Anzumerken ist ebenfalls, dass beide Modi das erwähnte Hot-Deployment aus Kapitel 4.1 unterstützen.

## Einrichtung des *dev*-Modus

Die IDE *IntelliJ IDEA* erleichtert es ungemein, eine Einstellung für die Entwicklung im *dev*-Modus zu erstellen. Hierfür öffnet man lediglich das im rechten oberen Bildrand befindliche Dropdown und wählt *Edit Configurations...* aus. Danach erscheint folgendes Menü:

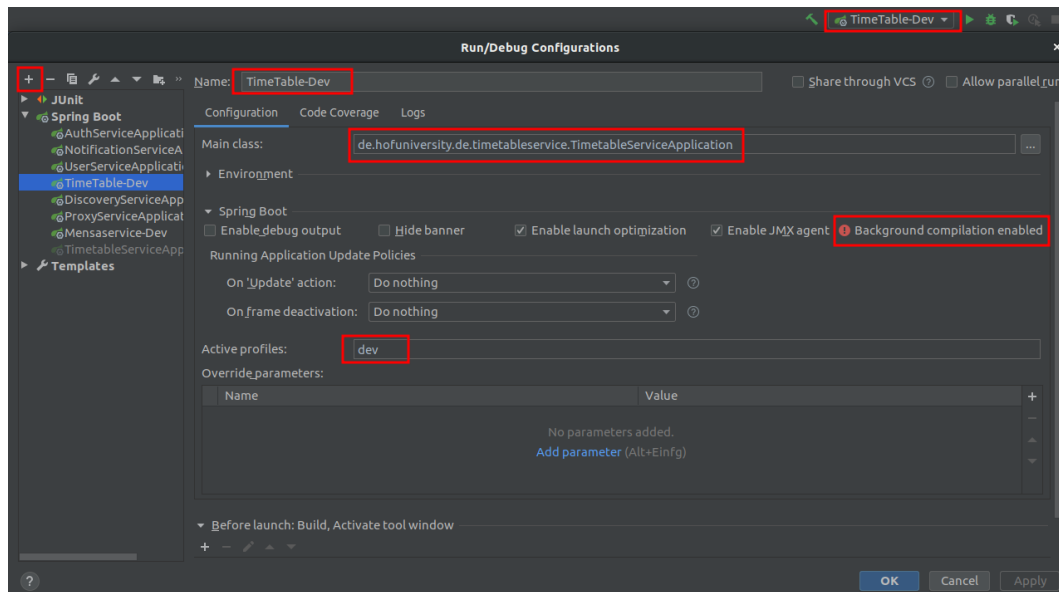


Abbildung 5.3: Einstellen der Test-Umgebung in der IDE

Die in der Abbildung 5.3 markierten Felder werden nun von oben nach unten und von links nach rechts erläutert:

- Auswählen von *Edit Configurations...* im Dropdown
- Erstellen einer neuen *Spring* Konfiguration
- Vergeben eines Namens für die Konfiguration
- ! *Background compilation enabled* sollte angezeigt werden, da dies in der Einrichtung zum Hot-Deployment in Kapitel 4.1 eingerichtet wurde.
- Zuordnung zum *Active Profile* DEV (Der Name ist hier frei wählbar, wird aber später gebraucht)

Zusätzlich muss nun in den Ressourcen des Microservices eine neue Properties-Datei erstellt werden, die nach folgendem Schema benannt werden muss:  
*application-DEV.properties*

DEV ist hierbei der Name des *Active Profile*, welches vorher konfiguriert wurde. In diesem Properties-File können nun alle Konfigurationen für die Testumgebung hinterlegt werden, die *dev*-Konfiguration, die in der IDE vorgenommen wurde, lädt diese dank der Namensgebung automatisch. Das Properties-File hat den selben Aufbau und Zweck wie das in Kapitel 4.2 erklärte File.

## 5.2 Prod-Environment

Das starten der Services auf der produktiven Umgebung bedarf einer festgelegten Vorgehensweise. Hierbei ist die Reihenfolge der Schritte, die im folgenden kurz aufgelistet werden, kritisch.

1. Start des *Eureka*-Servers
2. Start des API-Gateways
3. Start der anderen Services

Das Starten der Services erfolgt nach dem Erstellen der einzelnen *\*.jar*-Dateien für jeden Service. Diese werden dann mit folgendem Befehl gestartet:

### Code 5.3: Starten eines *jar*-Files

```
1 ~: java -jar [dateiname].jar
```

Da die Reihenfolge dieses Vorgangs kritisch ist und beliebig viele Services gestartet werden können, bietet sich es an, diesen Vorgang per Skript zu automatisieren. In diesem Zug kann man auf die Ausgabe der Programme in eine Log-Datei weiterleiten und die Anwendung im Allgemeinen beobachten. Ein passendes Skript auf einer UNIX-basierten Umgebung könnte wie folgt aussehen:

## Code 5.4: Skript zum Verwalten der Microservices

```
1  #!/bin/sh
2  #repeat each step per *.jar files
3  #needed in correct order
4
5  #Start API packaged in microservice.jar
6  start(){
7      #Create .log directory
8      mkdir .log > /dev/null 2>&1
9      echo "Starting API on port 8080"
10     #Start timetableAPI.jar, Redirect Error and Output to
        logfile
11     java -jar ~/microservice.jar >>
        ~/.log/microservice.log 2>&1 &
12     echo "Started API"
13 }
14
15 #Stop API Process running on tcp port 8080
16 stop(){
17     echo "Stopping API on port 8080"
18     #fuser finds processfile, -k kills process
19     fuser -k 8080/tcp
20     echo "Stopped API"
21 }
22
23 #Checking status of API running on tcp port 8080
24 status(){
25     #if fuser finds process for port 8080
26     if fuser 8080/tcp then
27         echo "API running on port 8080"
28     else
29         echo "API not running"
30     fi
31 }
32
33 restart(){
34     echo "Stopping API to reload"
35     stop
36     start
37     echo "Successfully restarted API"
38 }
39
40 #Switch params to call functions
41 case $1 in
```



```
42 start)
43     start
44 ;;
45 stop)
46     stop
47 ;;
48 status)
49     status
50 ;;
51 restart|reload)
52     restart
53 ;;
54 *)
55     echo "Usage: $0 (start|stop|status|restart|reload)"
56     exit 1
57 esac
58
59 #End script
60 exit 0
```

## 6 Microservice Dokumentation

Wie bereits an mehreren Stellen in dieser Arbeit erwähnt wurde, wird zur Dokumentation der Microservices und deren Endpunkte das Framework *Spring-Fox* verwendet. Dieses baut auf dem weit verbreiteten Standard *Swagger* auf. *Swagger* ermöglicht es, Endpunkte standardisiert in Form von YAML-Dokumenten zu dokumentieren, deren Nutzung zu erklären und die möglichen Datenabfragen aufzulisten. Im folgenden Kapitel soll nun erläutert werden, wie das *Spring-Fox*-Framework eingebunden wurde, wie die Endpunkte damit dokumentiert wurden und wie ein Nutzen aus diesem Framework gezogen werden kann.

### 6.1 Konfiguration im Microservice

Wie bereits im Kapitel 4.2 beschrieben wurde, werden für die Nutzung von *Spring-Fox* im Prototypen der Hochschul-App zwei Abhängigkeiten benötigt. Diese werden im POM.XML der einzelnen Services folgendermaßen eingebunden:

Code 6.1: Einbinden der *Spring-Fox*-Abhängigkeiten

```
1 <dependency>
2     <groupId>io.springfox</groupId>
3     <artifactId>springfox-swagger2</artifactId>
4     <version>2.9.2</version>
5 </dependency>
6 <dependency>
7     <groupId>io.springfox</groupId>
8     <artifactId>springfox-swagger-ui</artifactId>
9     <version>2.9.2</version>
10 </dependency>
```

Zusätzlich muss nun noch im *src*-Package des Quellcodes ein Konfigurations-Bean erstellt werden, das mit der Annotation `@ENABLESWAGGER2` und `@CONFIGURATION` markiert werden muss. Diese Bean-Klasse kann wie folgt aussehen:

Code 6.2: Bean zur Swagger Konfiguration im Mensa-Service

```
1  @Configuration
2  @EnableSwagger2
3  public class SwaggerConfig {
4
5      @Bean
6      public Docket api() {
7          return new Docket(DocumentationType.SWAGGER_2)
8              .useDefaultResponseMessages(false)
9              .directModelSubstitute(LocalDate.class,
10                 String.class)
11              .select()
12              .apis(RequestHandlerSelectors
13                 .basePackage("de.hofuniversity.mensaservice"))
14              .paths(regex("/menu.*"))
15              .build()
16              .apiInfo(metaInfo())
17              .tags(new Tag("Mensa Controller", "The
18                 resource manages the creation, reading and
19                 deletion of all dishes"),
20                 new Tag("Dish Controller", "The resource
21                 manages the update, reading and
22                 deletion of specific dish"),
23                 new Tag("Filter Controller", "The
24                 resource manages the creating,
25                 update, reading and deletion of
26                 possible filter query parameters"),
27                 new Tag("Menu Date Controller", "The
28                 resource manages the reading of all
29                 dishes depending on dates"),
30                 new Tag("Menu Week Controller", "The
31                 resource manages the reading of all
32                 dishes depending on calender week and
33                 also day of the week"));
34
35      private ApiInfo metaInfo() {
36          return new ApiInfoBuilder()
37              .title("Mensa-Service API")
38              .description("The service provides the client
39                 with all information about dishes and also
40                 allows them to filter, sort and to
41                 personalize")
42              .version("1.0.0")
43              .build();
44      }
45  }
```

```
28         .contact(new Contact("Dennis Brysiuk", "",  
29                               "dennis.brysiuk@hof-university.de"))  
30         .termsOfServiceUrl(  
31             "https://www.hof-university.de/impressum.html"  
32         ).build();  
33     }
```

Es werden kurz die Methoden und deren Inhalte erläutert:

- **new Docket(...)**  
Erstellt neues Docket, das API Informationen generiert
- **apis(...)**  
Fügt dem Docket eine Liste von API-Dokumentationen hinzu
- **paths(...)**  
Zeigt dem Docket, für welche Endpunkte Doku generiert werden muss
- **tags(...)**  
Teilt die Endpunkte in Kategorien ein, die im Controller definiert werden
- **new ApiInfoBuilder()**  
Erstellt allgemeine Infos zur API
- **title(...)**  
Gibt der API einen Titel
- **description(...)**  
Beschreibt die API
- **version(...)**  
Gibt die Versionsnummer der API an
- **contact(...)**  
Fügt der Dokumentation eine Referenzperson an
- **termsOfServiceUrl(...)**  
Erstellt einen Link zum Impressum

## 6.2 Controller Beschreibung

In der Konfigurations-Klasse wurden bereits allgemeine Informationen zur gesamten API hinterlegt, darunter auch die Information, welche Endpunkte gescant und dokumentiert werden sollen. Dennoch müssen nun genauere Informationen zu den Endpunkten in den Controller Klassen einer API annotiert werden. Das sieht am auf Klassenebene folgendermaßen aus:

Code 6.3: Swagger Konfiguration auf Controller Ebene am Beispiel des Timetable Controllers

```

1  @RestController
2  @RequestMapping("/lectures")
3  @ApiResponses(value = {
4      @ApiResponse(code = 400, message = "Incorrect
      request from the client", response =
      HttpReturnErrorPattern.class),
5      @ApiResponse(code = 401, message = "Request for
      the resource requires authorization", response
      = HttpReturnErrorPattern.class),
6      @ApiResponse(code = 403, message = "You do not
      have sufficient authorization for this action",
      response = HttpReturnErrorPattern.class),
7      @ApiResponse(code = 404, message = "The requested
      resource was not found by the server", response
      = HttpReturnErrorPattern.class),
8      @ApiResponse(code = 500, message = "The request
      cannot be processed dues to an unexpected error
      on the server", response =
      HttpReturnErrorPattern.class)
9  })
10 @Api(tags = {"Timetable Information"})
11 public class TimetableController {
12     ...

```

Folgende Annotationen können auf Controller Ebene nun die Endpunkte genauer beschrieben:

- **@ApiResponses(value=[Array of ApiResponse])**  
Erstellt eine Sammlung von HTTP-Responses, die für alle in der Klasse definierten Endpunkt-Methoden gelten
- **@ApiResponse(...)**  
Erstellt genaue Informationen zu einer HTTP-Response

- **@Api(tags=[tags])**

Ordnet die Controller Klasse zu einem in der Konfiguration definiertem Tag hinzu

## 6.3 Dokumentation der Funktionen

Die auf Controller Ebene definierten Dokumentationen werden auf alle Methoden der Controller Klasse angewendet. Dennoch kann es sein, dass eine der Methoden für eine bereits auf Controller Ebene definierte Antwort eine andere Definition dokumentieren will oder eine HTTP-Response produziert, die alle anderen Methoden so nicht produzieren können. Diese Responses können nun auf Methoden Ebene überschrieben oder neu definiert werden. Des weiteren können hier die Dokumentationen zu den einzelnen Parametern festgelegt werden. Eine annotierte Methode könnte nun folgendermaßen aussehen:

Code 6.4: Swagger Konfiguration auf Methoden Ebene am Beispiel des Semester Controllers

```

1  @ApiOperation(value="Get number of semesters for given
    program", response = SemesterListT0.class)
2      @ApiResponses( value={
3          @ApiResponse(code=OK, message = "Successfully
    retrieved semesters for given program",
    response = SemesterListT0.class),
4      })
5      @ResponseStatus(value = HttpStatus.OK)
6      @GetMapping
7      public SemesterListT0 getSemesters(
8          @ApiParam(name = "Faculty Id", value = "Unique
    identifier for a faculty", required = true,
    example = "INF")
9          @PathVariable("faculty") String facultyId,
10         @ApiParam(name = "Program Id", value = "Unique
    identifier for a program", required = true,
    example = "MB")
11         @PathVariable("program") String programId){

```

- **@ApiOperation(...)**

Liefert genaue Informationen zur Methode und der zu erwartenden Antwort

- **@ApiResponse(value=[Array of ApiResponse])**  
Erstellt eine Sammlung von HTTP-Responses, die für die annotierte Methode gelten
- **@ApiResponse(...)**  
Erstellt genaue Informationen zu einer HTTP-Response
- **@ApiParam(...)**  
Beschreibt den zu übergebenen Parameter genau

## 6.4 Export der Dokumentation

Die große Stärke des *Swagger*-Standards ist das einfache lesen der erstellten YAML-Dateien. Jedoch wurde *Swagger* dahingehend erweitert, dass es nun auch Dokumentationen in anderen Formaten unterstützt, die von Maschinen leichter prozessiert werden können. Unter anderem werden nun auch Dokumentationen in JSON unterstützt. Eine genau solche Dokumentation kann für jeden Service automatisch generiert und exportiert werden, indem man bei laufendem Service folgende URL aufruft:

### Code 6.5: Swagger Export URL

```
1 http://localhost:[port-of-service]/[version]/api-docs
```

## 6.5 Nutzen der grafischen Oberfläche

Aus der oben genannten Dokumentation kann das *Spring-Fox* Framework ebenfalls eine grafische Oberfläche generieren, mit welcher die Dokumentation besser gelesen und die Endpunkte sogar getestet werden können. Diese grafische Oberfläche kann über folgende URL aufgerufen werden:

### Code 6.6: Swagger Export URL

```
1 http://localhost:[port-of-service]/swagger-ui.html
```

Beim Aufrufen dieser URL wird dem Nutzer folgende Oberfläche angezeigt. Für dieses Beispiel wurde der Timetable-Service genutzt.

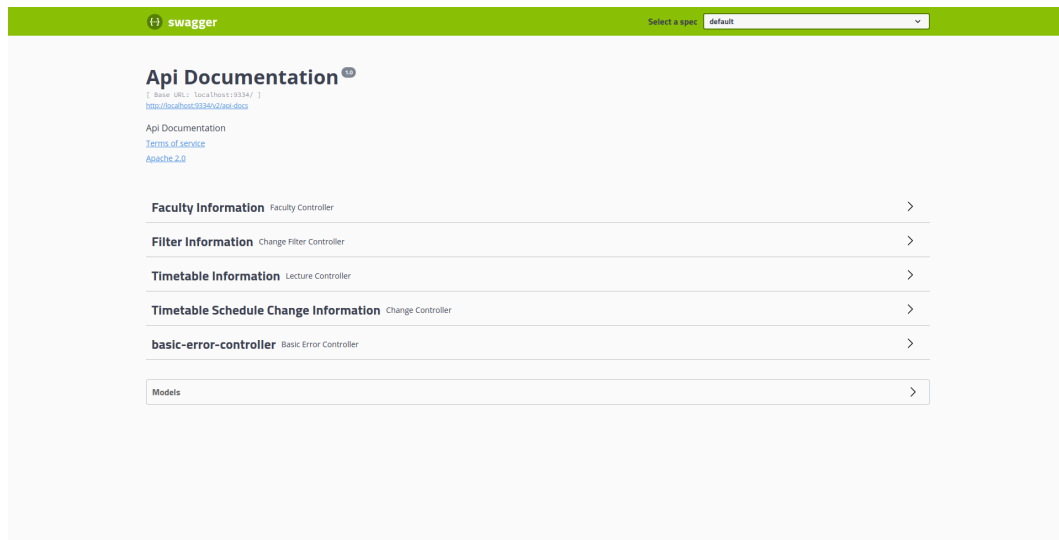


Abbildung 6.1: Swagger UI am Beispiel des Timetable-Service

Klappt man nun eines der Controller Felder auf, so werden alle in diesem Controller verfügbaren Methoden und Endpunkte mitsamt ihrer Beschreibungen und URLs aufgelistet.

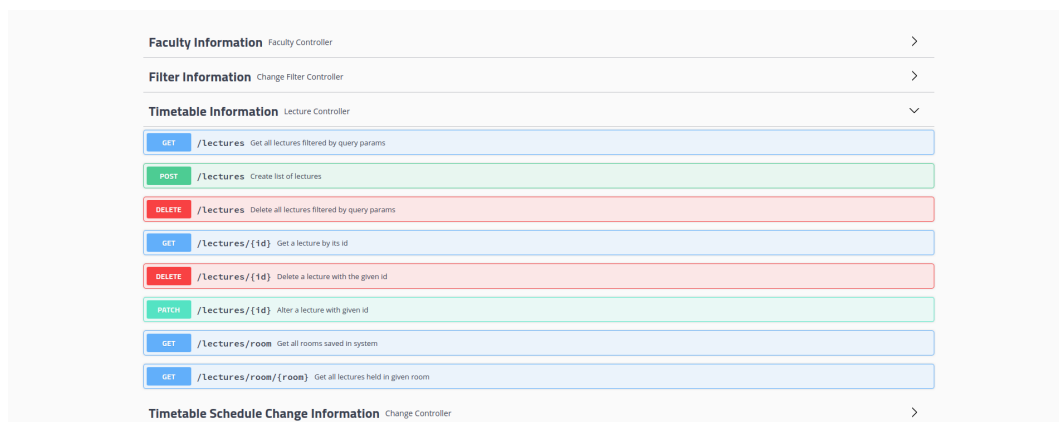



Abbildung 6.2: Swagger UI Timetable Methoden

Beim Auswählen einer der Methoden werden alle Details zu dieser einzelnen Methode angezeigt. Dazu gehören unter anderem auch alle möglichen Rückgabewerte, die Parameter, die von der Ressource akzeptiert werden und die zulässigen Werte, die in diesen Parametern übergeben werden können.




swagger

Select a spec:

default

## Api Documentation <sup>33</sup>

[View Docs \(SwaggerUI View\)](#)
[https://docbox933362.azure.io](#)

[Api Documentation](#)
[Terms of service](#)
[Apache 2.0](#)

Faculty Information

Faculty Controller

>

Filter Information

Change Filter Controller

>

Timetable Information

Lecture Controller

>

GET /lectures

Get all lectures filtered by query params

Parameters

Try it out

No parameters

Responses

Response content type: 

JSON

Code

Description

200

Successfully retrieved lectures

Example Value: Model

```
{
  "lessons": [
    {
      "lessonBegin": "2020-05-20T16:10:20.072Z",
      "lessonEnd": "2020-05-20T16:10:20.072Z",
      "lessonId": "2020-05-20T16:10:20.072Z",
      "lessonRoom": "2020-05-20T16:10:20.072Z",
      "lessonStatus": "2020-05-20T16:10:20.072Z",
      "lessonType": "2020-05-20T16:10:20.072Z",
      "lessonWeek": "2020-05-20T16:10:20.072Z",
      "lessonYear": "2020-05-20T16:10:20.072Z",
      "lessonRoom": "2020-05-20T16:10:20.072Z",
      "lessonStatus": "2020-05-20T16:10:20.072Z",
      "lessonType": "2020-05-20T16:10:20.072Z",
      "lessonWeek": "2020-05-20T16:10:20.072Z",
      "lessonYear": "2020-05-20T16:10:20.072Z",
      "lessonRoom": "2020-05-20T16:10:20.072Z",
      "lessonStatus": "2020-05-20T16:10:20.072Z",
      "lessonType": "2020-05-20T16:10:20.072Z",
      "lessonWeek": "2020-05-20T16:10:20.072Z",
      "lessonYear": "2020-05-20T16:10:20.072Z"
    }
  ]
}
```

400

Incorrect request from the client

Example Value: Model

```
{
  "errors": [
    {
      "error_code": 0,
      "error_message": "string"
    }
  ],
  "status_code": 0,
  "status_message": "string",
  "time_stamp": "2020-05-20T16:10:20.072Z"
}
```

401

Request for the resource requires authorization

Example Value: Model

```
{
  "errors": [
    {
      "error_code": 0,
      "error_message": "string"
    }
  ],
  "status_code": 0,
  "status_message": "string",
  "time_stamp": "2020-05-20T16:10:20.072Z"
}
```

403

You do not have sufficient authorization for this action

Example Value: Model

```
{
  "errors": [
    {
      "error_code": 0,
      "error_message": "string"
    }
  ],
  "status_code": 0,
  "status_message": "string",
  "time_stamp": "2020-05-20T16:10:20.072Z"
}
```

404

The requested resource was not found by the server

Example Value: Model

```
{
  "errors": [
    {
      "error_code": 0,
      "error_message": "string"
    }
  ],
  "status_code": 0,
  "status_message": "string",
  "time_stamp": "2020-05-20T16:10:20.072Z"
}
```

500

The request cannot be processed due to an unexpected error on the server

Example Value: Model

```
{
  "errors": [
    {
      "error_code": 0,
      "error_message": "string"
    }
  ],
  "status_code": 0,
  "status_message": "string",
  "time_stamp": "2020-05-20T16:10:20.072Z"
}
```

POST /lectures

Create list of lectures

DELETE /lectures

Delete all lectures filtered by query params

GET /lectures/{id}

Get a lecture by its id

DELETE /lectures/{id}

Delete a lecture with the given id

PATCH /lectures/{id}

Alter a lecture with given id

GET /lectures/room

Get all rooms saved in system

GET /lectures/room/{room}

Get all lectures held in given room

Timetable Schedule Change Information

Change Controller

>

basic-error-controller

Basic Error Controller

>

Models

>

Abbildung 6.3: Swagger UI Timetable Methoden Details

Wählt man nun im oberen rechten Eck der ausgewählten Ressource den Button *Try it out!*, so wird dem Nutzer eine Ansicht präsentiert, die ihm die Möglichkeit gibt, alle Übergabeparameter einzugeben, die erlaubten Header für die HTTP Anfrage zu konfigurieren und die Anfrage abzusenden. Die Anfrage wird dann an den Server weitergeleitet und ausgeführt. Hierbei ist anzumerken, dass die Anfragen unumkehrbar ausgeführt werden und nicht nur simuliert sind. Darauf präsentiert die *Swagger-UI* dem Nutzer folgende Darstellung der Antwort des Servers:

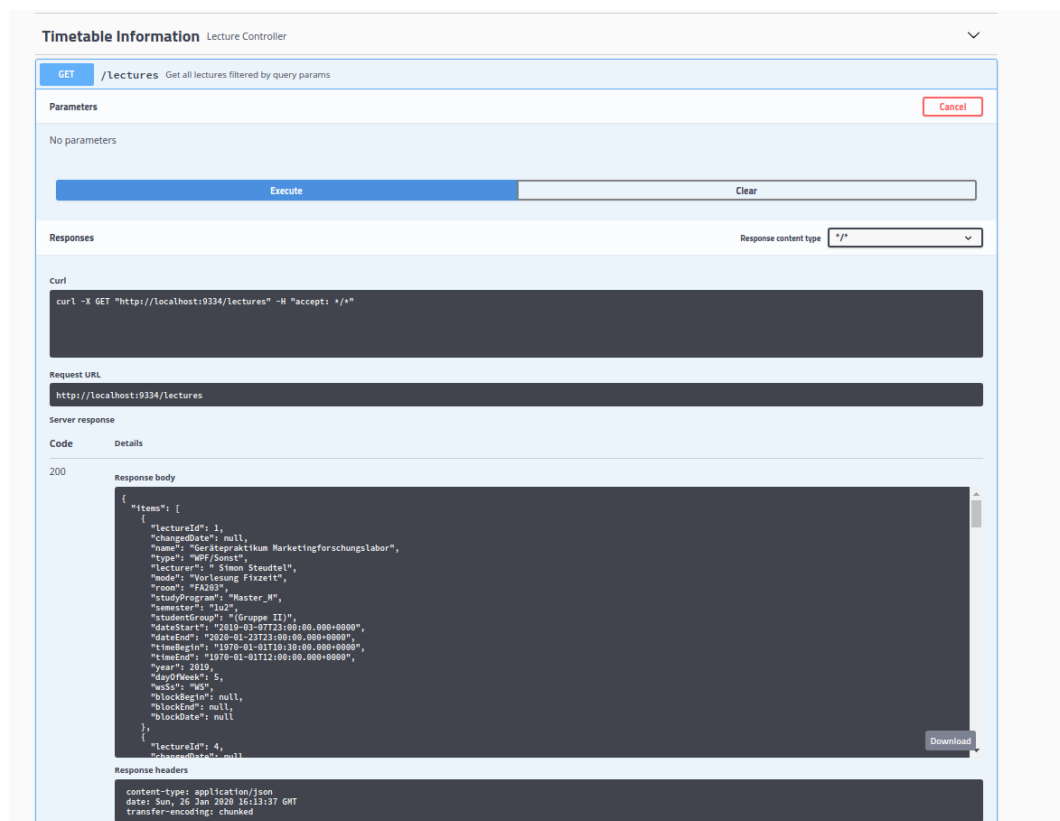


Abbildung 6.4: Testen einer Anfrage in der Swagger UI

## 7 Testing

Wie in jedem Software Projekt wurden auch im Falle des Prototypen der neuen Hochschul-App Tests durchgeführt. Diese ermöglichen es, unvorhergesehene Fehler oder Sicherheitslücken im Vornherein zu einem gewissen Grad auszuschließen. Um Microservices in *Spring* zu Testen reichen zwei Arten von Tests, Integrationstests und Unit-Tests. Integrationstests dienen dazu, verschiedene voneinander abhängige Komponenten eines komplexen Systems im Zusammenspiel zu testen. Wichtig ist hierbei, dass manche Komponenten integriert werden müssen, also werden fehlende Module simuliert und andere Komponenten werden gleichzeitig zusammengefügt und in das zu testende System integriert. In Unit-Tests hingegen werden lediglich die Einzelteile des Systems isoliert auf ihre korrekte Funktionalität geprüft.

Die Test-Frameworks für Integrationstests und Unit-Tests stellt *Spring Boot* mit dem Framework *spring-boot-starter-test* zu Verfügung. Dieses muss lediglich über eine Maven-Dependency in das Projekt eingefügt werden. Außerdem muss noch eine *H2*-Datenbank als *Maven-Dependency* eingefügt werden. Eine *H2*-Datenbank ist eine In-Memory Datenbank. Diese wird für das Testen der Datenschicht benötigt und wurde bereits ausführlicher im Kapitel 5 beleuchtet.

Da bei vielen Software Projekten die maximal mögliche Testabdeckung erreicht werden muss, werden dort in der Regel unzählige Stunden und Ressourcen in den Testvorgang investiert. Da dies den Rahmen dieser Arbeit deutlich sprengen würde werden nicht alle Schichten der Gesamtsoftware getestet. Um dennoch eine möglichst große Wahrscheinlichkeit zu erreichen, alle groben Fehler eliminiert zu haben, werden sowohl die Serviceschicht, die Controller-Schicht als auch die Util Klassen getestet.

Im folgenden werden nun nicht alle Testfälle angeführt, da dies oft redundante Fälle sind, die sich nur in Details unterscheiden. Stattdessen wird von jeder Art der Tests ein Fall exemplarisch erläutert und genauer betrachtet. Allgemein gilt für alle

Testfälle, dass die Annotation `@RunWith(SpringRunner.class)` an die Testklasse angehängt werden muss, damit das *Spring Boot* Framework eine Verbindung zu den Funktionen des Test-Framework *JUnit* aufbauen kann.

## 7.1 Controller Tests

Klassische Unit-Tests können an Controller-Klassen nicht angewendet werden. Diese bilden einen Sonderfall, denn sie behandeln im Laufe der Anwendung Anfragen mit dem HTTP-Protokoll, reichen diese an die Serviceschicht weiter und liefern lediglich eine Antwort an den aufrufenden Client. Hierbei ist es also notwendig, dass das *Spring Boot* Framework die nötige HTTP-Unterstützung bereitstellt. Hierbei geht es konkret um die Funktionen zur Serialisieren und Deserialisieren von Eingabeparametern, die durch die Annotations `@RequestBody`, `@PathParam` und `@PathVariable` markiert sind und durch die Annotation `@Valid` validiert werden müssen.<sup>1</sup>

Des Weiteren werden alle benötigten Abhängigkeiten oder Beans bei den Tests ignoriert, man kann sogar so weit gehen und die Autokonfiguration des Controllers durch die Annotation `@WebMvcTest(Controller.class)` verhindern, wenn dies gewünscht ist. So können alle benötigten Konfigurationen simuliert werden. Auch das Starten des Servers kann ausgespart werden, wenn die Test-Klasse mit `@MockMvc` annotiert wurde. Dies spart wertvolle Zeit und Rechenressourcen. Das Deserialisieren von Objekten, die den Austausch von Daten über die HTTP-Schnittstelle simulieren sollen, wird durch sogenannte *ObjectMapper* realisiert.

---

<sup>1</sup>Testing Spring MVC Web Controllers with `@WebMvcTest` (2020).

Ein Beispielhafter Test wird im folgenden anhand des Tests für den Controller der Mensa-Informationen dargestellt.

Code 7.1: Basis Initialisierung für Controller Tests

```
1 @RunWith(SpringRunner.class)
2 @WebMvcTest(DishController.class)
3 class DishControllerTest {
4
5     @Autowired
6     private MockMvc mockMvc;
7
8     @Autowired
9     private ObjectMapper objectMapper;
10    @MockBean
11    private DishService dishService;
12
13    //Tests
14 }
```

Generell ist der Controller für das Routen der Anfragen zuständig. Folgende Punkte sind hierbei wichtig für die Tests:<sup>2</sup>

- Verifizierung einer gültigen HTTP-Anfrage
- Verifizierung der Parameter Serialisierung einer HTTP-Anfrage (RequestBody, PathParam und PathVariable)
- Gültigkeitsprüfung der Parameter
- Verifizierung des Aufrufs der Business Logik

Alle anderen Szenarien werden in anderen Schichten behandelt, denn wie bereits erwähnt ist der Controller nur für das Routen zuständig und das Ziel ist hierbei lediglich die Abdeckung aller REST-URLs des Controllers.

## Verifizierung einer HTTP-Anfrage

Es wird getestet, ob die aufgerufene Ressource mit der passenden HTTP-Methode und einem gültigen Input funktioniert (Pfad Abdeckung).

<sup>2</sup>Testing Spring MVC Web Controllers with @WebMvcTest (2020).

## Code 7.2: Gültige Anfrage Test

```

1  @Test
2  public void verifyValidRequest() throws Exception {
3      mockMvc.perform(get("/menu/{id}", 42L))
4              .andExpect(status().isOk());
5  }

```

Es wird stets erwartet, dass die Ressource den Status 200 zurück gibt.

## Verifizierung der Parameter Serialisierung

Es wird getestet, ob die übergebenen Parameter an die Ressource in das gewünschte Java-Object umgewandelt werden können und wie sich die Ressource bei fehlenden Parametern Verhält.

**Fall 1:** Bei einem falschen Parameter Typ sollte die Ressource mit dem Internal Error 12 und einem HTTP-Status 400 Bad Request antworten (z.B. String anstatt von Int).

## Code 7.3: Falscher Typ Test

```

1  @Test
2  public void wrongParamType() throws Exception {
3      //get request (id as string instead of long)
4      MvcResult mvcResult = mockMvc.perform(get("/menu/{id}",
5          "test"))
6          .andExpect(status().isBadRequest())
7          .andReturn();
8
9      //current response
10     String response =
11         mvcResult.getResponse().getContentAsString();
12
13     //expected response (internal error code 12)
14     InternalErrorPattern expectedResponse = new
15         InternalErrorPattern(UNEXPECTED_FORMAT_CODE,
16             UNEXPECTED_FORMAT_MSG);
17
18     //comparison
19     assertThat(response)
20         .containsIgnoringCase(objectMapper
21             .writeValueAsString(expectedResponse));
22 }

```

**Fall 2:** Es werden nicht alle geforderten Parameter an die Ressource übergeben. Die Ressource soll mit dem Internal Error 13 und einem HTTP-Status 400 Bad Request antworten.

Code 7.4: Fehlende Parameter Test

```

1  @Test
2  public void noRequestBody() throws Exception {
3      //patch request (missing body param)
4      MvcResult mvcResult =
5          mockMvc.perform(patch("/menu/{id}", 42L)
6              .contentType("application/json"))
7              .andExpect(status().isBadRequest())
8              .andReturn();
9
10     //current response
11     String response =
12         mvcResult.getResponse().getContentAsString();
13
14     //expected response (internal error code 13)
15     InternalErrorPattern expectedResponse = new
16         InternalErrorPattern(UNEXPECTED_COUNT_OF_PARAM_CODE,
17             UNEXPECTED_COUNT_OF_PARAM_MSG);
18
19     //comparison
20     assertThat(response)
21         .containsIgnoringCase(objectMapper
22             .writeValueAsString(expectedResponse));
23 }

```

## Gültigkeitsprüfung der Parameter

Für die Übergabeparameter des Typs *RequestBody*, *PathParam* und *PathVariable* können Regeln definiert werden. Die möglichen Regeln sind:

Annotation	Definition
@NotNull	Variable darf nicht null sein
@AssertTrue	Variable muss true sein
@Size	Minimale und Maximale Länge einer Variable. Kann bei String, Collection, Map oder Array verwendet werden
@Min	Minimale Länge einer Variable
@Max	Maximale Länge einer Variable
@Email	Variable muss gültigen Email Format beinhalten
@NotEmpty	Variable darf nicht leer oder null sein. Kann bei String, Collection, Map oder Array verwendet werden
@NotBlank	Variable darf nicht null sein oder nur Leerzeichen beinhalten. Kann nur bei Text Variablen verwendet werden
@Positive oder @PositiveOrZero	Variable muss positive oder positive inklusiv 0 Werte enthalten
@Negative oder @NegativeOrZero	Variable muss negative oder negative inklusiv 0 Werte enthalten
@Past oder @PastOrPresent	Variable muss in der Vergangenheit oder in der Gegenwart liegen. Kann nur bei Datum Variablen verwendet werden
@Future oder @FutureOrPresent	Variable muss in der Zukunft oder in der Gegenwart liegen. Kann nur bei Datum Variablen verwendet werden

Tabelle 7.1: Validation Rules

Diese Regeln reduzieren das Sicherheitsrisiko und verhindern unerwartete Fehler beim Ausführen des Services. Außerdem können bei den Parameter Grenzwerte festgelegt werden. Beispielsweise können so Attribute mit Minimal- und Maximalwerten definiert werden. Es kann auch definiert werden, ob NULL-Werte akzeptiert werden, die Werte positiv sein müssen oder ob Whitespaces erlaubt werden. Für die Einhaltung der Regeln können ebenfalls Nachrichten definiert werden, die dem Client bei einem Fehler signalisieren, was er falsch gemacht hat. Ein Attribut kann auch mehr als eine Regel haben, die sie definiert. Wird eine dieser Regeln verletzt,



so wird eine Exception mit dem Internal Error 14 und dem HTTP-Status 400 Bad Request geworfen. Das Aktivieren einer solchen Validierung wird über die Annotation `@VALIDATED` initiiert, mit welcher die Controller Klasse beschriftet sein muss.

Code 7.5: Validierung Aktivieren

```
1 @RestController
2 @Validated
3 public class DishController {
4     ...
5 }
```

*PathParam* und *PathVariable* Variablen können direkt im Controller mit den Validierungsregeln annotiert werden.

Code 7.6: Validierung einer PathVariable

```
1 @GetMapping("/{id}")
2 public ResponseEntity<DishTO> getDish(
3     @PathVariable("id")
4     @Positive(message = "ID must be a positive value")
5     @Max(value = Long.MAX_VALUE, message = "ID is to long")
6     Long id){dish
7     ...
8 }
```

Für *RequestBody* Objekte wird die Validerung mit der Annotation `@VALID` aktiviert und die Validierungsregeln im passenden Transfer Objekt definiert.

## Code 7.7: Validierung von RequestBody

```

1  @PatchMapping("/{id}")
2  public ResponseEntity<DishT0> changeDish(
3      @Valid @RequestBody DishT0 dishT0) {
4      ...
5  }
6
7  public class DishT0 {
8
9      @NotNull(groups = OnPatchValidation.class, message = "ID
      is required for updating the resource")
10     @Positive(message = "ID must be a positive value")
11     @Max(value = Long.MAX_VALUE, message = "ID is too long")
12     private Long id;
13
14     @NotNull(message = "Date is required")
15     private LocalDate date;
16
17     @NotBlank(message = "Category is required")
18     private String category;
19
20     @NotEmpty(message = "Prices are required")
21     private HashMap<String, Double> prices;
22
23     ...
24 }

```

Werden die Validierungsregeln vom Client nicht eingehalten wird ein HTTP-Error 400 mit der Nachricht der jeweiligen Regeln an den Client weitergeleitet. Dies kann bei Tests geprüft werden.

## Code 7.8: Parameter Regeln Test

```

1  @Test
2  public void ParamRuleViolation() throws Exception {
3      //get request (id is negative but must be positive value)
4      mockMvc.perform(get("/menu/{id}", -12L))
5          .andExpect(status().isBadRequest());
6  }

```

## Verifizierung des Aufrufs der Business Logik

Beim Aufruf einer Ressource sollte sichergestellt werden, dass die Methode der Business Logik aus der Service Schicht, an die die Bearbeitung delegiert wird, nur

maximal einmal oder gar nicht aufgerufen wird und dass keine weiteren Aktionen mit der Business Logik erfolgen.

Code 7.9: Verifizierung der Business Logik

```
1 ...
2 //Method in Service must be called only once from
   Controller
3 Mockito.verify(dishService, times(1)).getDish(anyLong());
4 //There is no interaction more allowed with this Service
5 Mockito.verifyNoMoreInteractions(dishService);
6 ...
```

## 7.2 Service Tests

Durch Controller Tests wird zwar sichergestellt, dass die Business Logik mit gültigen Parametern aufgerufen wird, aber nicht, ob der Inhalt dieser Parameter fachlich korrekt und somit verwendbar ist. Hierfür muss die Pfadabdeckung der zu testenden Business Logik gewährleistet werden. Das bedeutet, dass beim Aufruf einer Methode mit bestimmten Parametern auch der gewünschte Programmpfad abgedeckt wird. Dafür gibt es im Test Framework die Methoden `WHEN()` und `THEN()`. Ähnlich wie in der Controller Schicht ist hierbei auch zu beachten, dass die Service Schicht bei der Verarbeitung der Anfragen abhängig von den Daten ist, die die darunter liegende Persistenz-Schicht liefert. Um die benötigten Beans zu simulieren bietet das *Spring* Framework die Annotation `@MOCKBEAN`.

Um die zu testende Methode in der Service Schicht auch aufrufen zu können, muss die Klasse, in der sie definiert ist, als Bean zur Verfügung gestellt werden. Hierbei wird - wie in den eigentlichen Klassen - die Annotation `@AUTOWIRED` verwendet. Wie in Kapitel 4.2 bereits erklärt wurde wird die Implementation der Bean, die injiziert werden soll, durch die *Spring*-Konfiguration bereitgestellt. Diese ist bei Testklassen jedoch nicht verfügbar, weshalb zusätzlich die Annotation `TESTCONFIGURATION` benötigt wird. Durch diese Annotation können die benötigten Abhängigkeiten manuell bereitgestellt werden. Die konkrete Verwendung sieht dann wie folgt aus:

## Code 7.10: Basis Initialisierung für Service Tests

```

1  @RunWith(SpringRunner.class)
2  public class MensaServiceImplTest {
3      @TestConfiguration
4      static class MensaServiceImplTestContextConfiguration {
5          @Bean
6          public MensaService mensaService() {
7              return new MensaServiceImpl();
8          }
9      }
10     @Autowired
11     private MensaService mensaService;
12     @MockBean
13     private MensaDao mensaDao;
14 }

```

**Fall 1:** Ein richtiges Verhalten testen. Hierbei wird die Businesslogik mit korrekten Parametern aufgerufen, wobei getestet wird, ob darauf die Persistenz Schicht aufgerufen ist, insofern dies erwünscht ist.

## Code 7.11: Testen von richtigem Verhalten

```

1  @Test
2  public void whenValidId_thenReturnValidValue(){
3      //Simulate expected JPA call
4      //When Logic call method with ID that exists
5      when(mensaJPA.findById(1L))
6          .thenReturn(Optional.ofNullable(dishDO));
7      DishTO expectedValue = DishTO.createTOfromDO(dishDO,
8          null);
9      //Then return dish with that ID
10     DishTO responseValue = dishService.getDish(1L);
11     assertThat(responseValue.toString())
12         .isEqualTo(expectedValue.toString());
13 }

```

**Fall 2:** Ein falsches Verhalten testen. Die Business Logik wird inkorrekt aufgerufen, wobei eine bestimmte Exception erwartet wird.

Code 7.12: Testen von Fehlverhalten

```

1  @Test
2  public void whenNotValidId_thenReturnClientException(){
3      //Simulate clientException
4      //When Logic call method with ID that not exists
5      when(mensaJPA.findById(2L)).thenReturn(null);
6      //Then return ClientException
7      Throwable exception = catchThrowable(() -> {
8          dishService.getDish(2L);
9      });
10     assertThat(exception)
11         .assertInstanceOf(ClientException.class);
12 }

```

## 7.3 Persistenz Tests

Anders als bei den darüber liegenden Schichten ist es nicht notwendig, die Persistenz-Schicht ausgiebig zu testen, da der Großteil ihrer Funktionen durch das *Spring Data JPA* Framework bereitgestellt wird. Die eigentlichen Aufrufe an die physische Datenquelle werden deshalb nicht getestet. Dennoch gibt es Teile der Persistenz, die manuell erstellt wurden, um eine höhere Flexibilität zu erreichen. Konkret sind das die sogenannten *Specifications*, die dem *Spring* Framework beim Aufruf an die Datenquelle übergeben werden, um die gesuchte Ressource genauer zu beschreiben. Diese Spezifikationen können durchaus sehr komplex ausfallen, weshalb es ratsam ist, diese auch zu testen. Ein solcher Test könnte folgendermaßen aussehen:

Code 7.13: Testen von Fehlverhalten

```

1  @Test
2  public void validationOfFilterSpecification(){
3      //When filter contains only category types
4      MenuFilterTO filters = new MenuFilterTO();
5      filters.setCategories("Maindish", "Dessert");
6      Specification<DishDO> filterSpecification =
7          FilterSpecification.getSpecification(filters);
8      List<DishDO> responseValue
9          = mensaJPA.findAll(filterSpecification);
10     //Then return only dishes with given category types
11     assertThat(responseValue)
12         .isNotEmpty().hasSize(4).extracting("category")
13         .containsOnly("maindish", "dessert");
14 }

```

Um die Spezifikationen testen zu können, wird die Annotation `@DATAJPATEST` benötigt. Diese kümmert sich um die Konfiguration der benötigten *H2*-Testdatenbank, *Spring Data*, *Hibernate* und des Loggers, der die SQL-Statements loggt. Die *H2*-Datenbank wird benötigt, um beim simulierten Aufruf an die Datenbank die benötigten Daten zurückliefern zu können. Um diese Daten dann aus der *H2* Datenbank aufrufen zu können, wird ein *TestEntityManager* benötigt, der die Methoden bereitstellt, die im eigentlichen Programmfluss vom *Spring Data JPA* Framework zur Verfügung gestellt werden.<sup>3</sup>

Code 7.14: Testen von Fehlverhalten

```

1 @RunWith(SpringRunner.class)
2 @DataJpaTest
3 public class MensaJPATest {
4     @Autowired
5     private TestEntityManager entityManager;
6     @Autowired
7     private MensaJPA mensaJPA;
8 }

```

Die Konfiguration der *H2*-Datenbank wird durch die im folgenden gezeigte `setUp()` Methode durchgeführt. Diese ist mit der Annotation `@Before` markiert, was aussagt, dass sie vor jeder Test-Methode aufgerufen wird.

Code 7.15: Testen von Fehlverhalten

```

1 @Before
2 public void setUp () {
3     List<DishDO> items = new ArrayList<>(...);
4     entityManager.persistAndFlush(items);
5 }

```

## 7.4 Funktionstests

Im Package *Utils* sind einige Klassen und Funktionen hinterlegt, welche nicht zur eigentlichen Logik der Anwendung zugeordnet werden können, die jedoch trotzdem im Gebrauch sind, um komplexe Aufgaben auszuführen. Diese Funktionen sind in der Regel leicht durch Unit-Tests zu testen, denn hier soll lediglich abgeglichen werden, ob sich die Klasse oder Funktion genau so verhält, wie gewünscht. Ein Beispiel für den Anwendungsfall einer Util-Funktion ist die Methode `GETNEXTWEEKDA-`

<sup>3</sup>Testing in Spring Boot (2020).

TEBYDAY() der Klasse DAYDATEUTIL.CLASS, welche für einen übergebenen Tag den Start der darauf folgenden Woche als Datum zurückgeben soll.

Code 7.16: Testen von Fehlverhalten

```
1  @Test
2  public void getNextWeekDateByDay() {
3      //Instance of testing object
4      DayDateUtil dayDateUtil = new DayDateUtil();
5
6      //Expected Value
7      LocalDate nextWeekDate = LocalDate.now().plusWeeks(1);
8      DayOfWeek nextWeekDateAsDay =
9          nextWeekDate.getDayOfWeek();
10     String expectedValue = nextWeekDate
11         .format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
12
13     //Res
14     String responseValue =
15         dayDateUtil.getNextWeekDate(dayOfWeek);
16
17     //comparison
18     assertEquals(expectedValue, responseValue);
19 }
```

Bei einem solchen Unit-Test reicht es vollkommen aus, eine Instanz der benötigten Klasse zu erzeugen und dann die Methode mit verschiedensten Parametern aufzurufen, um das zurückgelieferte Ergebnis mit dem erwarteten Ergebnis zu vergleichen.

## 8 Probleme

Da im Verlaufe des Projektes zur Implementierung des Prototypen der neuen Hochschul-App ebenfalls Probleme aufgetreten sind, deren Ursache nicht oder nur schwer zu beseitigen waren wird im folgenden Kapitel genauer betrachtet, welche Schwierigkeiten in der Entwicklungsphase gefunden wurden und wie sich der endgültige Prototyp aufgrund dieser Schwierigkeiten von der geplanten Version unterscheidet. Im Allgemeinen sind zwei schwerwiegende Probleme aufgetreten, deren Lösung den Rahmen der Abschlussarbeit deutlich sprengen würden, weshalb sie hier nur dokumentiert wurden. Das erste ist die kontinuierliche Integration des Source Codes in den laufenden Serverbetrieb. Das zweite ist die Implementierung der HATEOAS-Funktionen des Web-Servers. Das letzte Problem betrifft weniger die Implementierung selbst, da die betroffene Funktion des Notification Services durchaus umgesetzt wurde. Es betrifft lediglich die Nutzung dieser Funktion.

### 8.1 Kontinuierliche Integration und Deployment

Kontinuierliche Integration ist ein Konzept aus dem Software Engineering, welches es ermöglicht, Änderungen am Programmcode und neue Funktionen schnellstmöglich und ohne manuellen betriebenen Aufwand in die Entwicklungsumgebung des laufenden Servers zu übernehmen. Hierbei wird der aktuelle Source Code in der Regel durch ein Versionsverwaltungstool aktuell gehalten, Änderungen an diesem können dann in festgelegten Intervallen in den Programmcode übernommen werden, der bereits auf dem Server läuft und automatisch deployed werden, um danach den Server - falls nötig - neu zu starten. Ein solcher Vorgang kann zu Zeiten mit niedriger Serverauslastung durchgeführt werden, um den laufenden Betrieb unter Last nicht zu stören.

#### Kontinuierliche Integration

Continuous Integration (CI) ist die automatische Übernahme von neuem, getesteten Source Code in die Code-Version, die bereits auf dem Server läuft. Hierbei können



Entwickler jederzeit Änderungen am Code übernehmen, diese testen und anschließend einen Integrationsprozess anstoßen, wenn genug Änderungen vorhanden sind. Bei diesem Prozess wird der aktualisierte Source Code auf dem Server hinterlegt, um für den Schritt des kontinuierlichen Deployments zur Verfügung zu stehen.

## Kontinuierliche Bereitstellung

Continuous Deployment (CD) ist ein Konzept, das in der Regel Hand in Hand mit CI geht. Wie bereits erwähnt können Entwickler den auf dem Server liegenden Source Code jederzeit durch CI aktualisieren, um ihn anschließend mit CD auf dem Server zu deployen. Dabei ist die Idee hinter dem CD in der Regel, dass das Deployment weitgehend automatisiert abläuft und entweder nur von einem Entwickler angestoßen oder durch einen festgelegten Zeitplan ausgeführt wird.

## Jenkins

Das Konzept des CI/CD ist eine Idee, die durch viele verschiedene Lösungen realisiert werden kann. Darunter zählen unter anderem das in GitLab integrierte CI/CD-Modul und *Jenkins*. Viele andere Lösungen haben sich in den vergangenen Jahren zwar durchaus bewährt, sind aber durch teure Lizenzen mit hohen Kosten verbunden und deshalb keine Option für dieses Projekt.

Aus diesem Grund wurde die Umsetzung der Funktionalitäten rund um *Jenkins* geplant, da *Jenkins* eine open-source Lösung ist, welche im vollen Umfang für die Hochschul-App nutzbar ist. *Jenkins* ist ein Java-basiertes Programm, welches auf jedem beliebigen System verwendbar ist. Durch die Anbindungsmöglichkeit verschiedenster Plugins ist es ebenfalls frei erweiterbar und äußerst flexibel. Es wurden durch die verbreitete Nutzung und die Starke Community von *Jenkins* bereits zahlreiche, frei nutzbare Features veröffentlicht. Zudem stellt *Jenkins* ein web-basiertes Interface zur Verfügung, durch welches die Build- und Deployment-Konfigurationen eingestellt und ausgeführt werden können. Einige der Stärken des Frameworks sind die automatisierte Ausführung von Tests in Echtzeit, der Möglichkeit einer Remote-Verwaltung und die Möglichkeit der Verteilung von User-Rollen.

Leider konnte diese Funktion nicht implementiert werden, da die vollständige Einrichtung dieser Software weit führende Vorbereitungen benötigt hätte. Dazu gehören die Verteilung von Administrationsrechten am Server, an dem *Jenkins* ausgerollt

werden soll, was bei der Arbeit mit Studierenden häufig Probleme verursacht. Diese Rechte werden für die Installation der Software, aber auch zur Konfiguration des Servers im Allgemeinen gebraucht. Zudem werden Server Zertifikate benötigt, die die Kommunikation des Servers mit den Remote-Usern und dem Applikations-Servern verschlüsseln. Allen voran scheitert diese Funktion jedoch am schieren Umfang der Einrichtung, die nahe an der eigentlichen Implementierung des Prototypen herankommen würde.

Deshalb hat sich das Entwicklerteam dazu entschieden, die Möglichkeit einer kontinuierlichen Bereitstellung aktueller Software zu dokumentieren und die aktuell am besten geeignete Software dafür vorzuschlagen. Aus den beschriebenen Problemen können spätere Entwickler die nötigen Schlüsse ziehen, um bei der Vorbereitung zur Umsetzung dieser Funktion auf keine unvorhergesehenen Probleme zu stoßen.

## 8.2 Hypermedia

In der parallel zu dieser Arbeit angefertigten Bachelorarbeit wurde ausführlich analysiert, warum HATEOAS eine sinnvolle Erweiterung für den Funktionsumfang des Prototypen der neuen Hochschul-App sein kann. Aufgrund der Konfiguration und der Zusammenstellung der verwendeten Bibliotheken ist es aber derzeit nicht möglich, das in Kapitel 4.2 beschriebene HATEOAS-Framework einzubinden, da sich bei der Initialisierung der *Spring Beans* die Abhängigkeiten des *Spring-Fox*-Frameworks mit den benötigten Abhängigkeiten des HATEOAS-Frameworks überschneiden. Da beide Frameworks eine eigene `PLUGINREGISTRY` verwenden, kann *Spring Boot* zur Laufzeit nicht bestimmen, welche der beiden Registries verwendet werden soll. Der produzierte Fehler sieht dann bei Programmstart wie folgt aus:

## Code 8.1: PluginRegistry Bean Konflikt

```

1  *****
2  APPLICATION FAILED TO START
3  *****
4
5  Description:
6
7  Parameter 0 of method linkDiscoverers in
      org.springframework.hateoas.config.HateoasConfiguration
      required a single bean, but 17 were found:
8  - modelBuilderPluginRegistry: defined in null
9  - modelPropertyBuilderPluginRegistry: defined in null
10 - typeNameProviderPluginRegistry: defined in null
11 - syntheticModelProviderPluginRegistry: defined in null
12 - documentationPluginRegistry: defined in null
13 - apiListingBuilderPluginRegistry: defined in null
14 - operationBuilderPluginRegistry: defined in null
15 - parameterBuilderPluginRegistry: defined in null
16 - expandedParameterBuilderPluginRegistry: defined in null
17 - resourceGroupingStrategyRegistry: defined in null
18 - operationModelsProviderPluginRegistry: defined in null
19 - defaultsProviderPluginRegistry: defined in null
20 - pathDecoratorRegistry: defined in null
21 - apiListingScannerPluginRegistry: defined in null
22 - relProviderPluginRegistry: defined by method
      'relProviderPluginRegistry' in class path resource
23     [org/springframework/hateoas/config/
24         HateoasConfiguration.class]
25 - linkDiscovererRegistry: defined in null
26 - entityLinksPluginRegistry: defined by method
      'entityLinksPluginRegistry' in class path resource
27     [org/springframework/hateoas/config/
28         WebMvcEntityLinksConfiguration.class]
29
30 Action:
31
32 Consider marking one of the beans as @Primary, updating
      the consumer to accept multiple beans, or using
      @Qualifier to identify the bean that should be consumed
33
34 Process finished with exit code 0

```

Diese Framework Fehler sind in beiden Frameworks bekannt und werden jeweils in den Framework-Dokumentationen als eigene Issues geführt.<sup>1</sup> Ursprünglich wurde dieses Problem bei der Nutzung des HATEOAS-Frameworks in Verbindung mit *Spring* gefunden.<sup>2</sup> Jedoch wurde dieser Fehler behoben, wobei die Lösung nun nur noch Fehler in der Nutzung von Frameworks verursacht, die eine eigene PLUGIN-REGISTRY verwenden. Da dieser Fehler innerhalb der Frameworks liegt, kann er im Rahmen dieses Projektes nicht behoben werden.

Es musste also entschieden werden, welches Framework, *Spring-Fox* oder HATEOAS, einen größeren Mehrwert für die Anwendung bringen. Da das *Swagger*-Framework, wie in Kapitel 4.2 bereits erklärt wurde, eine grafische Nutzeroberfläche liefert, welche sowohl die Endpunkte dokumentiert, als auch diese aufrufen kann und somit ein wichtiges Tool zum einfachen Testen darstellt, wurde das *HATEOAS*-Framework aus dem derzeitigen Umfang der Arbeit entfernt. Jedoch ist es wünschenswert, das spätere Entwicklerteams die referenzierten Issues weiterhin verfolgen und die Funktion, sobald diese behoben wurden, endgültig einfügen.

## 8.3 Notification Service

Der Notification Service, der Änderungen in den Daten der REST-Services erkennt und diese dann an die registrierten Clients weitergibt, wurde in vollem Umfang implementiert und oberflächlich getestet. Seine Funktion, die Einrichtung und die Nutzung des Services sind in Kapitel 4.7 ausführlich dargestellt, weshalb hier auf eine erneute Erklärung des Notification Services verzichtet wird.

Das ausführliche Testen dieses Services konnte aufgrund eines Funktionsfähigen Clients nicht getestet werden. Dieser wird benötigt, da sich zum Testen mindestens ein Client beim Service registrieren muss, damit dieser eine Benachrichtigung an seine Abonnenten schickt. Des weiteren wurden die Funktionen des Notification Services lediglich implementiert, jedoch nicht in die anderen Services eingebaut, da dies den Umfang der Arbeiten deutlich vergrößert hätte und somit den Umfang der gesamten Arbeit deutlich gesprengt hätte. Jedoch wurden alle benötigten Maßnahmen ergriffen, um eine spätere Einbindung der Benachrichtigungsfunktionalitäten in die anderen Microservices weitgehend zu erleichtern.

---

<sup>1</sup>Siehe [spring-projects/spring-hateoas \(2020a\)](#); Siehe [springfox/springfox \(2020\)](#).

<sup>2</sup>Siehe [spring-projects/spring-hateoas \(2020b\)](#).

## 9 Weiterentwicklung

Für eine sinnvolle und nachhaltige Pflege einer Anwendung muss von Anfang an dafür gearbeitet werden, dass spätere Entwickler einen möglichst leichten Einstieg in den Programmcode und dessen Funktionsweise finden. Das bedeutet auch, dass diese Neuanfänger möglichst viele Ressourcen haben, mit denen sie sich schnellstmöglich in den Ablauf des Programmes einarbeiten können. Ist dies nicht der Fall, so läuft man bei einem Hochschulinternen Projekt Gefahr, dass die freiwilligen Helfer des Projekts die Motivation verlieren, weiter an der Pflege und der Entwicklung der Anwendung zu arbeiten.

Des weiteren sollte man bei solchen Projekten, die oftmals auf freiwilliger Basis unterstützt werden oder zumindest auf der Hilfe von Studierenden beruhen, auch darauf achten, dass ständig neue Hilfe in das Projekt eingegliedert werden kann. Das Ziel dieses Kapitels ist es deshalb, Möglichkeiten zu erschließen, andere Studierende von dem Projekt zu begeistern und diesen danach auch eine solide Grundlage zu verschaffen, auf der sie dem Projekt den größtmöglichen Nutzen zukommen lassen können.

### 9.1 Pflege der Anwendung

Da diese Praxisarbeit und die dazugehörige Anwendung der web-basierten Hochschul-App im Rahmen einer Abschlussarbeit des Bachelorstudiums geschrieben werden, ist es klar, dass in Zukunft andere Entwickler an dem Projekt weiter arbeiten werden, als die, die es ins Leben gerufen haben. Deshalb werden nun die vorgeschlagenen Vorgehensweisen für neue Entwickler dargestellt, mit denen diese sich in das Projekt einarbeiten können und durch die sie anschließend auch daran weiterarbeiten können.

## **Dokumentation der Anwendung**

Die verschiedenen Phasen des Projektes sind im Laufe der Umsetzung dessen auch dokumentiert worden. Der Übersichtlichkeit halber wurden aber nicht alle Phasen in ein großes Projekthandbuch geschrieben, sondern in mehrere Teile aufgeteilt. Zu diesen Teilen gehören die Analyse, die eigentliche Implementierungsarbeit und die Inbetriebnahme, zu der auch die Schritte zur Weiterentwicklung gehören. Die einzelnen Dokumentationsquellen werden im folgenden kurz erläutert.

### **Architektur**

Im Rahmen der Analyse zu Beginn des Projektes der neuen HochschulApp wurden einige wichtige Designentscheidungen getroffen. Diese Beruhen auf den Erkenntnissen, die aus einer Befragung der zukünftigen Nutzergruppen gewonnen wurden und auf dem Lastenheft, das aus den funktionalen Anforderungen des Auftraggebers, des International Office und des Sprachenzentrums der Hochschule Hof entstanden ist.

Die Umfrageanalyse und die funktionalen Anforderungen sind in der parallel zu dieser Praxisarbeit entstandenen Bachelorarbeit gesammelt.<sup>1</sup> Dort sind außerdem auch die wichtigsten Grundlagen zum Einarbeiten in die genutzten Techniken und Prinzipien sowie die Entscheidungsfindung zu genau diesen Techniken dokumentiert. Entwickler, die neu in das Projekt mit eingegliedert werden sollten diese Arbeit zuerst kurz überfliegen. Für die reine Entwicklung sind dort der Mittelteil und der Schlussteil besonders wichtig. Im Mittelteil werden die genutzten Techniken genauer erläutert und im Schlussteil wird anhand der gewonnenen Erkenntnisse die Architektur der Hochschul-App entworfen.

Besonders wichtig ist auch das REST-Handbuch, das in dieser Arbeit dokumentiert ist. Hier werden alle Regeln zum Entwurf von neuen Ressourcen und zum Arbeiten mit dem HTTP beziehungsweise mit dem Hypertext Transfer Protocol (Secure) (HTTPS) geschildert.

### **Implementierung**

Nachdem die Analyse der Hochschul-App bereits in der im Kapitel 9.1 beschriebenen Bachelorarbeit dokumentiert wurde, ist der Entwicklungsprozess in der dazu

---

<sup>1</sup>Siehe Brysiuk; Lehmann (2019a).

parallel entstandene Praxisarbeit, genau dieser Arbeit, festgehalten worden.<sup>2</sup> Wie in den vorherigen Kapiteln bereits zu erkennen war, werden hier alle verwendeten und nicht trivialen Frameworks und Techniken genauer betrachtet. Des weiteren kann man alle Spezifikationen des ersten Prototypen der web-basierten Hochschul-App dieser Arbeit entnehmen.

Da alles weitere bereits mehrfach in dieser Arbeit erklärt wurde, wird an dieser Stelle nicht weiter auf die Inhalte der Praxisarbeit eingegangen.

### Programmcode

Um einen besseren Einstieg in den Aufbau und die Funktionsweise eines Programmes zu erhalten, empfiehlt es sich oftmals nicht nur den Programmcode an sich zu betrachten, sondern auch das Programm einmal selbst zu starten und damit zu testen. Da der Programmcode selbst auf dem GitLab-Server der Hochschule Hof verfügbar ist, kann sich der Entwickler den Code lokal auf seinen eigenen Rechner herunterladen. Alle Anleitungen zum Starten der Services finden sich im dazugehörigen *Readme.md*. Die Funktionsweise der einzelnen Services sind ebenfalls in *Readme*-Dateien in den einzelnen Modulen des Programmcodes beschrieben.

Des weiteren existiert auf dem GitLab-Repository der Hochschul-App auch ein sogenanntes *Wiki*, welches genauere Einsicht in den Programmcode und die Funktionsweise der Anwendung verschafft. Dort wird auch beschrieben sein, wie der Programmcode lokal auf dem persönlichen Rechner ausgeführt werden kann. Dazu bieten die in Kapitel 5 beschriebenen *Environments* - auf Deutsch *Umgebungen* - eine Lösung.

### Endpunkte

Da die Datenbereitstellung der Hochschul-App über Microservices realisiert wurde sind die Endpunkte nach den REST-Prinzipien definiert. Das bedeutet, dass jede Ressource einen eigenen URL-Pfad besitzt, über den sie aufgerufen werden kann. Jede Ressourcenanfrage benötigt doch auch einige Parameter, um genaue Daten liefern zu können. Dies können zum Beispiel Suchparameter, Authentifizierungsdaten oder Filteranweisungen sein. Da diese Parameter sich aber zwischen den Ressourcen unterscheiden, sind die Endpunkte durch *Swagger-Files* dokumentiert worden.

---

<sup>2</sup>Siehe Brysiuk; Lehmann (2019b).

Genauerer kann dazu in Kapitel 6 gelesen werden.

Der genaue Zugriff auf diese Files und die dazu generierten, interaktiven grafischen Oberflächen ist in den *Readme*-Dateien der einzelnen Microservices hinterlegt. Die grafischen Oberflächen können nicht nur zur Spezifikation der einzelnen Endpunkte genutzt werden, sie bieten auch die Funktion, echte Anfragen an die Endpunkte zu stellen, ohne die dazugehörige URL händisch eingeben zu müssen.

## Quellcodeverwaltung

Wie bereits in Kapitel 3.4 angesprochen wurde, wird für das Projekt das Versionsverwaltungstool *git* verwendet. Die genaue Nutzung kann dazu ebenfalls in Kapitel 3.4 gelesen werden. Das GitLab-Repository der Hochschul-App beinhaltet somit stets den aktuellen Code des Programmes - inklusive der älteren Versionen. Möchte ein Studierender oder externer Helfer somit am Quellcode der Anwendung arbeiten, so braucht er zuerst ein Konto oder gültige Zugangsdaten für den GitLab-Server der Hochschule.

Danach muss er vom Verwalter des Hochschul-App-Repositories zur passenden Entwickler Gruppe hinzugefügt werden. Ist das dann erledigt, so kann er den Programmcode *clonen*, also das Repository lokal auf seinen Rechner laden und die Anwendung dann starten. Danach kann er Änderungen ausführen und diese dann wieder in das Repository hochladen. Auch hierzu wird auf Kapitel 3.4 verwiesen, wo die genaue Vorgehensweise dokumentiert ist.

Anzumerken ist jedoch auch, dass alle Studierenden der Hochschule Hof mit einem Zugang zum GitLab-Server der Hochschule den Code einsehen können und auch Änderungsvorschläge beisteuern können. Genauerer hierzu wird in den folgenden Kapiteln 9.2, 9.4 und 9.5 erläutert.

## 9.2 Programmcode Veröffentlichung

Die Veröffentlichung des Quellcodes einer Anwendung birgt Anfangs zwar einige Risiken, bringt aber auf lange Sicht gesehen einige Vorteile mit sich. Bei der Überlegung, ob der Programmcode der web-basierten Hochschul-App öffentlich oder zumindest Hochschulintern verfügbar sein soll, sind folgende Punkte mit eingeflossen:



- **Öffentliche Darstellung von Sicherheitslücken**

Da der Programmcode von jedermann einsehbar ist, kann es sein, dass in der Prototyp Phase, aber auch zu späteren Versionen der Anwendung, Sicherheitslücken gefunden werden. In den falschen Händen angekommen können diese Sicherheitslücken auch ausgenutzt werden. Dies schließt die Veröffentlichung des Quellcodes außerhalb der Hochschule selbst aus. Dennoch können Interessierte Studierende diese Sicherheitslücken finden und melden, wodurch sie schneller behoben werden können. Lösungen zum Melden solcher Lücken sind in Kapitel 9.4 zu finden.

- **Eigenständige Suche nach Programmfehlern**

Der große Vorteil einer Open-Source-Community ist es, dass alle Interessierten den Quellcode sehen und somit auch verbessern können. Sollte zum Beispiel ein Fehler im Ablauf gesehen werden, so kann dieser nicht nur gemeldet werden, der Finder selbst könnte sich den Code genauer ansehen und den Fehler selbst beseitigen. Die Lösung des Problems kann er als Vorschlag dann an die Entwickler schicken. Genauer dazu wird in Kapitel 9.5 beschrieben.

- **Vorschläge zu Erweiterungen**

Die aktuellsten und beliebtesten Anwendungen der heutigen Zeit bieten große Mengen an Funktionen, implementieren aber auch jeweils die gängigen Designphilosophien. Bei der Entwicklung des Prototypen der Hochschul Anwendung können dabei nicht alle diese Faktoren mit einfließen. Findet somit ein Nutzer etwas, das ihm an der Anwendung nicht gefällt, sei es in der Datenbereitstellung oder im dazu entwickelten Frontend, dann kann er dafür eine Lösung implementieren und diese dann den Entwicklern zur Verfügung stellen<sup>3</sup>. Diese können dann in Absprache mit dem Auftraggeber entscheiden, ob die Lösung sinnvoll ist, sie gegebenenfalls verbessern oder anpassen und sie dann in der nächsten Version der Hochschul-App mit veröffentlichen. So haben die Nutzer ein deutlich besseres Verhältnis zur Anwendung, da sie wissen, dass sie noch in Bearbeitung ist und dass sie selber einen Unterschied machen können.

---

<sup>3</sup>Siehe Kapitel 9.5

- **Weiterbildung für Interessierte**

Die web-basierte Hochschul-App wurde aufwändig entwickelt und so gestaltet, dass sie mit den neuesten Techniken und Frameworks arbeitet<sup>4</sup>. Somit kann der Programmcode für viele auch nur dafür dienen, sich ein Bild über die Funktionsweise der Techniken zu machen. Außerdem kann der Quellcode so auch als Beispiel für Themenverwandte Vorlesungen dienen.

Betrachtet man all diese Punkte, so ist klar festzustellen, dass die Offenlegung ein Risiko mit sich bringt, da alle Sicherheitslücken frei einsehbar sind. Dafür wurden aber weiterführende Maßnahmen ergriffen, die die Manipulation der Software und des Servers auf dem sie läuft erschwert und Angriffe auf den Angreifer zurückführen lassen. Dazu kann in Kapitel 9.3 mehr gelesen werden. Außerdem liegt es im Interesse der Studierenden, eine zuverlässige Anwendung zu haben, die ihnen den Alltag an der Hochschule vereinfacht. Die Vorteile, die bei der Veröffentlichung des Quellcodes angebracht wurden, überwiegen die Sicherheitsbedenken in diesem Fall. Sie ermöglichen es den interessierten und neugierigen Studierenden, sich in das Projekt einzufinden und mit der Software zu identifizieren. So weckt das open-source Projekt *web-basierte Hochschul-App* das Interesse seiner Nutzer. Der Code wird selbstverständlich aus oben genannten Gründen nur für Studierende der Hochschule Hof oder Entwickler mit einem Zugang zum GitLab-Server der Hochschule Hof veröffentlicht.

## 9.3 Verteilung von Zugangsdaten

Wie aus den beiden Bachelorarbeiten, die parallel zum Projekt der *web-basierten Hochschul-App* angefertigt wurden, hervorgeht, besteht die Datenbereitstellung aus einem REST-Backend, welches die Daten über HTTP-Zugriffe zugänglich macht.<sup>5</sup> Selbstverständlich sind die Endpunkte jedoch gegen unautorisierten Zugriff abgesichert. Jede Anfrage benötigt einen gültigen API-Schlüssel, um vom Server überhaupt bearbeitet zu werden.

Diese API-Schlüssel regeln nicht nur die Zugriffsberechtigung auf die Endpunkte im Allgemeinen, sie unterscheiden zusätzlich auch, welche Aktionen ein Anwender ausführen darf und welche nicht. So kann ein Administrator der Anwendung und der Daten beispielsweise einen speziellen Schlüssel nutzen, der ihm das Ändern und

---

<sup>4</sup>Siehe Kapitel 3

<sup>5</sup>Brysiuk; Lehmann (2019a); Glaser (2019a).

Löschen von Daten über die Schnittstelle erlaubt. Ein normaler Nutzer hingegen - oftmals ist ein solcher Nutzer keine konkrete Person, sondern eine Anwendung - hätte hingegen nur einen Schlüssel, mit dem er nur Daten lesen kann.

Diese API-Schlüssel sollten an alle Interessierten Studierende weitergegeben werden, denn sie bieten Nutzern die Möglichkeit, eine eigene Anwendung zu schreiben, die die Daten der Hochschul-App verarbeiten. So können Studierende die Daten auch dann sinnvoll nutzen, wenn sie die dazu entwickelte grafische Oberfläche nicht nutzen wollen oder können.

Selbstverständlich haben solche API-Schlüssel gewisse Restriktionen. Sie haben unter anderem nur eine gewisse Lebensdauer. Außerdem kann der Schlüssel-Administrator jederzeit Schlüssel für ungültig erklären, wenn sie unsauber genutzt werden. Das kann unter anderem auch automatisiert erkannt werden. Des weiteren können die Schlüssel bei Herausgabe an die Studierenden mit deren Namen oder Matrikelnummer verknüpft werden. Das ermöglicht eine Rückverfolgung, falls die Schnittstelle unsachgemäß genutzt wurde.

Abschließend kann man sagen, dass die Herausgabe von API-Schlüsseln eine sinnvolle Herangehensweise ist. So wird das Interesse der Studierenden an der Anwendung gefördert, was die Weiterentwicklung nach der Fertigstellung des Prototypen deutlich einfacher macht und auch gerechtfertigt. Auch für Lehrzwecke bieten sich solche Schlüssel an.

## **9.4 Problemfindung und Beseitigung**

Um eine einfache Fehlerbeseitigung zu ermöglichen, sollte den Nutzern ermöglicht werden, Fehler, die ihnen beim Nutzen der Anwendung aufgefallen sind, über verschiedene Wege an die Entwickler weiterzureichen. Ein klassischer Weg hierbei ist die Möglichkeit dem Entwicklerteam eine Mail zukommen zu lassen, die den Fehler beschreibt und auch einen Weg formuliert, wie man ihn rekonstruieren kann. Das ist jedoch oft sehr ungenau, außerdem können die Entwickler so die gefundenen Fehler nur schwer organisieren. Diese Herangehensweise sollte dringend implementiert werden, da es Nutzer gibt, die den technischen Hintergrund nicht besitzen, einen Fehler auf andere Arten zu berichten.

Für die technisch versierten Nutzer der App soll es möglich sein, Issues auf der *gitLab* Seite der Anwendung zu erstellen. Dabei muss der Fehler klar beschrieben werden, worauf dem Fehler eine Fehlernummer zugeordnet wird. Über diese Ticketnummer kann der Entwickler dann einen Source Code Branch erstellen, in dem er den Fehler behebt und testet, worauf er den Branch dann wieder in den Hauptbranch der Anwendung merged. Im nächsten Release ist der Fehler dann nicht mehr in der Anwendung zu finden. Der offene Fehler kann somit geschlossen werden.

Nutzer, die Erfahrung im Bereich der Programmierung haben können dank des open-source Ansatz ebenfalls den Source Code auf ihren persönlichen Computer laden und den Fehler selbst beheben. Die Lösung kann dann als Pull-Request auf GitLab hochgeladen werden. Die Entwickler der Anwendung können die Lösung dann prüfen, gegebenenfalls anpassen und dann in die Anwendung übernehmen.

## 9.5 Neue Funktionen

Ähnlich wie bei der Fehlersuche soll es technisch versierten und an der Anwendung interessierten Studierenden der Hochschule Hof möglich sein, neue Funktionen für die Anwendung vorzuschlagen. Hierfür gibt es analog zur Fehlersuche wieder drei Lösungsansätze. Der klassische Weg bleibt hierbei wieder die E-Mail an das Entwicklerteam. Da hier die gleichen Probleme wie bei der Fehlersuche auftreten soll deshalb nicht weiter auf diese Möglichkeit eingegangen werden, auch wenn sie definitiv in den Umfang der Pflege der Anwendung aufgenommen werden soll. Die zwei weiteren Möglichkeiten, neue Features vorzuschlagen, werden im folgenden betrachtet.

### Feature Requests

Ähnlich wie bei der Fehlersuche können Studierende, die verstehen, wie das GitLab Repository der Hochschule Hof funktioniert, einen sogenannten Issue erstellen, in dem sie das gewünschte Feature ausführlich beschreiben. Entwickler können dann bei Bedarf durch die Sammlung der Issues gehen und Features für neue Releases suchen. Das läuft analog zur Fehlersuche ab und muss deswegen nicht nochmals genauer beschrieben werden.

## **Code Contributions**

Besonders wünschenswert ist ein großes Interesse der Studierenden, besonders aus den technischen Studiengängen, am Verlauf und den Funktionen der neuen Hochschul-App. Schließlich profitieren alle Studierenden vom Engagement der Entwickler an der Anwendung. Da nicht alle Studierenden dauerhaft die Ressourcen haben, um Zeit in die Entwicklung der App zu investieren, soll es die Möglichkeit geben, eigene Funktionen zu implementieren und die Lösungen dann im Repository des GitLab Servers hochzuladen. Solche Vorschläge werden allgemein als Code Contribution bezeichnet und dienen als Möglichkeit, die Entwickler in ihrer Arbeit zu unterstützen und auch eigene Einflüsse in das Produkt einfließen zu lassen. Interessierte Studierende können im GitLab Repository den aktuellen Source Code der Anwendung auf ihre Systeme herunterladen und dann nach eigenem Empfinden modifizieren oder erweitern. Sobald keine weiteren Änderungen mehr nötig sind können sie den Code als Pull Request im Repository hochladen. Das Entwicklerteam hat dann die Möglichkeit diesen Vorschlag zu prüfen und gegebenenfalls anzupassen und danach in den eigentlichen Source Code zu mergen. Diese Möglichkeit der Mitarbeit soll den Studierenden die Anwendung näher bringen und ihnen klar machen, dass jeder die Möglichkeit hat, zu einer erfolgreichen und interessanten Hochschul-App beizutragen.

# 10 Ausblick und Fazit

Zum Abschluss dieser Praxisarbeit zur Implementierung des Prototypen der neuen, web-basierten Hochschul-App soll auf die großen Ziele dieser Neuimplementierung eingegangen werden. Hierbei werden nochmals die Beweggründe aufgelistet, die zur erneuten Entwicklung einer Anwendung geführt haben, die bereits in anderer Form bestand. Danach soll evaluiert werden, ob die gesetzten Ziele erreicht werden konnten.

## 10.1 Ausblick

Wie in der parallel zu dieser Arbeit entstandenen Bachelorarbeit mehrfach geäußert wurde, existieren im Hochschulumfeld bereits mehrere Implementierungen einer allgemeinen Hochschul-App. Die erneute Implementierung ist aus dem Wunsch heraus entstanden, eine plattformübergreifende Anwendung zu erschaffen, die die großen Probleme der bereits bestehenden Anwendungen eliminiert. Diese Probleme liegen besonders in der Plattformabhängigkeit der Anwendungen. Nachdem jede dieser Plattformen, darunter iOS, Android und Windows, spezielle Kenntnisse für die Entwicklung erfordern, die im allgemeinen Studium der Fakultät Informatik nicht zwingend gelehrt werden, wurde es immer schwerer, Entwickler zu finden, die das nötige Know-How hatten, diese Apps am Leben zu erhalten. Mit der neuen Hochschul Anwendung wird dies in Zukunft kein Problem mehr sein. Das Backend, um das es in dieser Praxisarbeit primär geht, wurde in der Programmiersprache Java geschrieben, eine Sprache, die in allen Studiengängen der Fakultät Informatik ausgiebig gelehrt wird und deren aller Studierenden dieser Studiengänge mächtig sein sollten. Das dabei verwendete *Spring Boot* Framework erleichtert das Arbeiten mit den komplexen Konzepten des Java EE Umfelds ungemein. Um keine Probleme bei der Entwicklung der Oberfläche zu erzwingen, wurde eine REST-Schnittstelle entwickelt, die Abhängigkeiten zur verwendeten Oberfläche komplett eliminiert. Dies garantiert die Zukunftsfähigkeit der Datenquelle und die dauerhafte Austauschbarkeit der Oberfläche.

Ein weiteres Problem der Nutzer der bestehenden Anwendungen ist der Mangel an neuen Funktionen. Durch fehlende Entwickler können Fehler und neue Funktionen an den plattformabhängigen Anwendungen nur schwer umgesetzt werden. Auch dieses Problem soll mit der neuen Hochschul Anwendung in Zukunft eliminiert sein. Einerseits bringt die Datenschnittstelle des REST Servers bereits durch sein großes Potential an Kombinationen der Daten weit mehr Funktionen mit, als die alten Anwendungen der Hochschule, andererseits ermöglicht die Microservice Architektur der App eine problemlose und extrem einfache Integration neuer Features. Diese Arbeit erläutert zudem, wie neue Funktionen von jedermann hinzugefügt werden können. Der Fokus auf die ständige Weiterentwicklung in Kombination mit der Robustheit der Anwendung steht klar im Zentrum dieses Projektes und wird in Zukunft auch den Erfolg der Anwendung garantieren.

Zu all den Vorteilen und Ausblicken, die bereits vorgestellt wurden, soll dennoch der Ursprung der Anwendung nicht vergessen werden. Sie ist im Rahmen einer Abschlussarbeit entstanden und lieferte somit eine Basis, auf der die Verfasser dieser Arbeit die Möglichkeit hatten, sich neues Wissen in den aktuellen Techniken im Bereich der web-basierten Programmierung anzueignen. Von dieser Möglichkeit sollen nun in Zukunft auch andere Studierende der Hochschule Hof und anderer Einrichtungen profitieren. Dies wird durch die Code Contributions und der Mitarbeitsmöglichkeiten ermöglicht. Zudem bietet die REST-Schnittstelle ebenfalls eine Datenquelle, an denen Studierende testen und verstehen können, wie die moderne Kommunikation zwischen Clients und Servern abläuft.

## **10.2 Fazit**

In dieser Praxisarbeit zum Thema der web-basierten Hochschul-App wurde ein solides Fundament für eine erfolgreiche Produktionsumgebung der neuen Anwendung geschaffen. Entwicklerteams, die die Arbeit an der App übernehmen, haben so die Möglichkeit, alles, das es über diese Software zu wissen gibt, nachzuverfolgen und zu verstehen. Im Rahmen der Rechtfertigung der verwendeten Ressourcen wie der Programmiersprache und des Microservice Frameworks wurden bereits wertvolle Erkenntnisse und Erklärungen dargestellt, welche den späteren Entwicklern die Einarbeitung in die Thematik deutlich vereinfachen werden. Des weiteren wurden die grundlegenden Konzepte und Konfigurationen der Anwendung und speziell des

*Spring Boot* Frameworks erläutert. Das ist besonders wichtig, um zu verstehen, welche Konfigurationen eventuelle Fehler verursachen könnten. Ebenso wichtig sind diese Informationen jedoch auch bei der Erstellung neuer Microservices, die den Funktionsumfang der Anwendung erweitern sollen, denn hierbei kann nach dem Vorbild der bereits vorhandenen Services vorgegangen werden.

Um dann produktiv an der Weiterentwicklung der neuen Hochschul-App arbeiten zu können wurden den späteren Entwicklern einige hilfreiche und äußerst wertvolle Werkzeuge an die Hand gegeben, die die Entwicklung an der Software deutlich vereinfachen. Hierzu wurde die IDE *IntelliJ IDEA* vorgestellt. Außerdem wurden die benötigten Einstellungen erklärt, welche zum Nutzen des vollen Funktionsumfangs dieser IDE notwendig sind. Darauf aufbauend wurden dann die verschiedenen Umgebungen erläutert, in der die Anwendung laufen wird. Hierbei wurde besonders Wert darauf gelegt, die Unterschiede zwischen der Entwicklungs- und der Produktionsumgebung aufzuzeigen.

Abschließend wurde dann die eigentliche Anwendung genauer untersucht. Es wurde die Dokumentation zu den REST-Endpunkten erklärt und gezeigt, wo diese zu finden ist und wie sie den Entwicklern bei der Arbeit hilft. Um die qualitativen Anforderungen der App zu verdeutlichen wurde ebenfalls detailliert dargestellt, wie das Backend der Anwendung getestet wurde und im späteren Verlauf auch weiterhin getestet werden soll. Die Probleme, die beim Testen und bei der Entwicklung aufgetreten sind, wurden daraufhin ebenfalls dargestellt. Besonders für die spätere Suche nach Fehlern soll dieses Kapitel eine wertvolle Ressource darstellen. Jedoch sollen spätere Probleme nicht nur von den Entwicklern selbst behoben werden können. Nach dem Erläutern des Weiterentwicklungsplans wurde auch nochmals betont, welche Rolle jeder einzelne Nutzer in der Pflege und der Qualitätssicherung der Anwendung hat.

Allgemein kann man somit sagen, dass das große Ziel, eine neue Hochschul-App zu schaffen, welche die Bedürfnisse eines jeden Studierenden der Hochschule Hof bestmöglich erfüllt, erreicht wurde. Mit dieser Arbeit soll der Prototyp nun abgeschlossen werden, womit der erste Kontakt der neuen Anwendung mit den Studierenden bevorsteht.



# Anhang

## Pflichtenheft

Bei klassischen vertraglichen Einigungen definieren Auftragnehmer und Auftraggeber in der Regel ein gemeinsames Lastenheft. Aus diesem wird dann ein Pflichtenheft erstellt, welches die Lasten in technische Umsetzungen wandelt. Vor Vertragsbeginn muss der Auftraggeber dieses nun abnehmen. Da dies beim Prototypen nicht umgesetzt wurde, da eine Einigungen zu den zu erledigenden Pflichten innerhalb einer Abschlussarbeit nur bedingt sinnvoll ist, wird im folgenden das Lastenheft erneut aufgenommen und auf seine Umsetzung untersucht. Dies soll als Pflichtenheft dienen.

Ref.	Anforderung	Quelle	Erledigt
1	Plattform	Auftraggeber	
1.1	Betriebssystemunabhängigkeit	Auftraggeber	Ja
1.2	Geräteunabhängigkeit	Auftraggeber	Ja
1.3	Mobile First	Auftraggeber	Teils
2	Stundenplan	Auftraggeber	
2.1	Abrufen des allgemeinen Stundenplans	Auftraggeber	Ja
2.2	Personalisierung nach Fakultät	Auftraggeber	Ja
2.3	Personalisierung nach Studiengang	Auftraggeber	Ja
2.4	Personalisierung nach Fachsemester	Auftraggeber	Ja
2.5	Einbindung nicht regulärer Vorlesungen	Auftraggeber	Nein
2.6	Einbindung des erweiterten Vorlesungsspektrums	Auftraggeber	Teils

Tabelle X.1: Pflichtenheft

Ref.	Anforderung	Quelle	Erledigt
3	Stundenplan Änderungen	Auftraggeber	
3.1	Unterscheidung einmalige/ langfristige Änderungen	Auftraggeber	Nein
3.2	Automatische Anpassung langfristiger Änderung in Stundenplan	Auftraggeber	Ja
3.3	Benachrichtigung der Nutzer bei Änderungen	Auftraggeber	Teils
3.4	Anzeige einmaliger Änderungen in Stundenplan	Auftraggeber	Ja
4	Speiseplan	Auftraggeber	
4.1	Anzeige des Speiseplans (Studentenwerk Oberfranken)	Auftraggeber	Ja
4.2	Filterung der Anzeige von Speiseplan Informationen	Auftraggeber	Ja
4.3	Anzeige zusätzlicher Informationen pro Gericht (z.B. Allergene)	Auftraggeber	Ja
5	Anwenderverwaltung	Auftraggeber	
5.1	Speicherung Nutzer spezifischer Informationen	Auftraggeber	Ja
5.2	Anmeldung notwendig bei Speicherung der Daten	Auftraggeber	Ja
5.3	Aufteilung der Nutzer in verschiedene Gruppen	Auftraggeber	Teils
5.4	Anmeldung durch Hochschul-E-Mail- Adresse	Auftraggeber	Ja
5.5	Anmeldung auch für Studierende ohne FH-E-Mail	Auftraggeber	Teils
5.6	Automatisiertes Löschen alter Daten	Auftraggeber	Teils

Tabelle X.2: Pflichtenheft

Ref.	Anforderungen	Quelle	Erledigt
6	Mehrsprachigkeit	Auftraggeber/ International- Office	
6.1	App in deutscher Sprache	Auftraggeber	Ja
6.2	App in englischer Sprache	Auftraggeber/ International- Office	Ja
6.3	Einfache Erweiterung der App um weitere Sprachen	International- Office	Ja
7	Einfache personalisierte Stundenplanerstellung	Auftraggeber/ International- Office	
7.1	Fakultäten unabhängige Stundenplanerstellung	Auftraggeber/ International- Office	Ja
7.2	Studiengang unabhängige Stundenplanerstellung	Auftraggeber/ International- Office	Ja
7.3	Einfache Einbindung externer Kurse	Auftraggeber/ Sprachzentrum	Teils
8	Einbindung des Sprachenzentrums	Sprachzentrum	
8.1	Einbinden von Sprachkursinformationen	Sprachzentrum	Ja
8.2	Mehrsprachige Sprachkursinformationen	Sprachzentrum	Teils
8.3	Einheitliche Darstellung der Sprachkursinformationen	Sprachzentrum	Ja
8.4	Vollständige Informationsdarstellung	Sprachzentrum	Nein

Tabelle X.3: Pflichtenheft

## Autoren Referenz

Diese wissenschaftliche Arbeit ist eine Zusammenarbeit der beiden Autoren Dennis Brysiuk und Noah Lehmann. Um daher einen besseren Überblick über die Zuordnung der Inhalte zu haben wird im folgenden eine Tabelle dargestellt, die die Kapitel ihren jeweiligen Verfassern zuordnet.

Kapitel Nr.	Kapitel Bezeichnung	Autor
<b>1</b>	<b>Einleitung</b>	
1.1	Beweggründe	Noah Lehmann
1.2	Zielsetzung	Noah Lehmann
1.3	Zielgruppe	Noah Lehmann
1.5	Vorgeschlagene Nebenlektüre	Noah Lehmann
<b>2</b>	<b>Vorgehensweise</b>	
2.1	Grundkonzept	Dennis Brysiuk
2.2	Umsetzung	Noah Lehmann
<b>3</b>	<b>Technologien</b>	
3.1	Programmiersprache	Noah Lehmann
3.2	Microservice Framework	Dennis Brysiuk
3.3	Build Management Tool	Noah Lehmann
3.4	Versionsverwaltung Tool	Noah Lehmann
3.5	API-Gateway	Dennis Brysiuk
3.6	Service Registrierung und Discovery	Dennis Brysiuk
3.7	Benachrichtigung	Dennis Brysiuk
<b>4</b>	<b>Konfiguration</b>	
4.1	Entwicklungsumgebung	Noah Lehmann
4.2	Spring Boot	Dennis Brysiuk
4.3	Spring Cloud Gateway	Dennis Brysiuk
4.4	Eureka	Dennis Brysiuk
4.5	API Filter	Noah Lehmann
4.6	Error Handling	Noah Lehmann
4.7	Firebase Cloud Messaging	Dennis Brysiuk

Tabelle X.4: Autoren Referenz

Kapitel Nr.	Kapitel Bezeichnung	Autor
<b>5</b>	<b>Entwicklungsumgebungen</b>	
5.1	Dev-Environment	Dennis Brysiuk
5.2	Prod-Environment	Noah Lehmann
<b>6</b>	<b>Microservice Dokumentation</b>	
6.1	Konfiguration im Microservice	Noah Lehmann
6.2	Controller Beschreibung	Noah Lehmann
6.3	Dokumentation der Funktionen	Noah Lehmann
6.4	Export der Dokumentation	Dennis Brysiuk
6.5	Nutzen der grafische Oberfläche	Dennis Brysiuk
<b>7</b>	<b>Testing</b>	
7.1	Controller Tests	Dennis Brysiuk
7.2	Service Tests	Dennis Brysiuk
7.3	Persistenz Tests	Dennis Brysiuk
7.4	Funktionstests	Noah Lehmann
<b>8</b>	<b>Probleme</b>	
8.1	Kontinuierliche Integration und Deployment	Noah Lehmann
8.2	HATEOAS	Dennis Brysiuk
8.3	Notification Service	Dennis Brysiuk
<b>9</b>	<b>Weiterentwicklung</b>	
9.1	Pflege der Anwendung	Noah Lehmann
9.2	Programmcode Veröffentlichung	Noah Lehmann
9.3	Verteilung von Zugangsdaten	Dennis Brysiuk
9.4	Problemfindung und Beseitigung	Dennis Brysiuk
9.5	Neue Funktionen	Dennis Brysiuk
<b>10</b>	<b>Ausblick und Fazit</b>	
10.1	Ausblick	Noah Lehmann
10.3	Fazit	Noah Lehmann

Tabelle X.5: Autoren Referenz

# Literaturverzeichnis

- Brysiuk, D.; N. Lehmann (2019a):** “Web-basierte Hochschul-App (Bachelorarbeit). Modulare Web-Architektur”. Hochschule für angewandte Wissenschaften Hof
- **(2019b):** “Web-basierte Hochschul-App (Praxisarbeit). Modulare Web-Architektur”. Hochschule für angewandte Wissenschaften Hof
- Christian Wenz, T. H. (2019):** PHP 7 und MySQL. Das umfassende Handbuch. Rheinwerk Verlag
- Glaser, A. (2019b):** “Web-basierte Hochschul-App (Praxisarbeit). Authentifizierung und Personalisierung”. Hochschule für angewandte Wissenschaften Hof
- **(2019a):** “Web-basierte Hochschul-App. Authentifizierung und Personalisierung”. Hochschule für angewandte Wissenschaften Hof
- Richard Hightower Warner Onstine, P. V.D.P.J.D. G. (2004):** Professional Java Tools for Extreme Programming. Ant, XDocklet, JUnit, Cactus and Maven. Wiley Publishing Inc,
- Ullenboom, C. (2019):** Java ist auch eine Insel. Einführung, Ausbildung, Praxis. Rheinwerk Verlag
- Wenz, C. (2014):** JavaScript. Grundlagen, Programmierung, Praxis. Galileo Press
- Wolff, E. (2007):** Spring 2. Framework für die Java-Entwicklung. dpunkt.verlag GmbH

# Internetquellen

**Apache Struts** (11. Jan. 2020): Apache Struts. The Apache Software Foundation.  
URL: <https://struts.apache.org/>

**Building RESTful Web Services with JAX-RS** (11. Jan. 2020): Building RESTful Web Services with JAX-RS. The Java EE 6 Tutorial. Oracle. URL: <https://docs.oracle.com/javaee/6/tutorial/doc/giepu.html>

**Entities** (19. Jan. 2020): Entities. The Java EE 6 Tutorial. Oracle. URL: <https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html> (besucht am 19.01.2020)

**Firestore console** (26. Jan. 2020): Firestore console. URL: <https://console.firebase.google.com>

**IntelliJ IDEA** (18. Jan. 2020): IntelliJ IDEA. Die Java-IDE von JetBrains für professionelle Entwickler. JetBrains s.r.o. URL: <https://www.jetbrains.com/de-de/idea/> (besucht am 18.01.2020)

**Jersey** (11. Jan. 2020): Jersey. Eclipse Foundation. URL: <https://eclipse-ee4j.github.io/jersey/>

**Jetbrains** (18. Jan. 2020): JetBrains. TOOLBOX SUBSCRIPTION AGREEMENT FOR EDUCATION. JetBrains s.r.o. URL: [https://www.jetbrains.com/student/license\\_educational.html](https://www.jetbrains.com/student/license_educational.html) (besucht am 18.01.2020)

**Kumuluz EE** (11. Jan. 2020): Kumuluz EE. A lightweight open-source microservice framework. KumuluzDigital. URL: <https://ee.kumuluz.com/>

**Open-Source Service Discovery** (27. Jan. 2020): Open-Source Service Discovery. URL: <http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/>

**PYPL Popularity of Programming Language** (4. Jan. 2020): PYPL Popularity of Programming Language. URL: <http://pypl.github.io/PYPL.html>

**Project Lombok** (18. Jan. 2020): Project Lombok. The Project Lombok Authors. URL: <https://projectlombok.org/> (besucht am 18.01.2020)

**Sending Web Push Notifications with Java** (27. Jan. 2020): Sending Web Push Notifications with Java. URL: <https://golb.hplar.ch/2019/08/webpush-java.html>

**Spring Boot** (11. Jan. 2020): Spring Boot. Pivotal Software. URL: <https://spring.io/projects/spring-boot>

**Spring Initializr** (12. Jan. 2020): Spring Initializr. Bootstrap your application. Spring Initializr und Pivotal Web Services. URL: <https://start.spring.io/index.html>

**Testing Spring MVC Web Controllers with @WebMvcTest** (24. Jan. 2020): Testing Spring MVC Web Controllers with @WebMvcTest. URL: <https://reflectoring.io/spring-boot-web-controller-test> (besucht am 24.01.2020)

**Testing in Spring Boot** (25. Jan. 2020): Testing in Spring Boot. Baeldung. URL: <https://www.baeldung.com/spring-boot-testing> (besucht am 25.01.2020)

**Trello** (26. Jan. 2020): Trello. Atlassian. URL: <https://trello.com/> (besucht am 26.01.2020)

**What is Gradle?** (12. Jan. 2020): What is Gradle? Gradle 6.0.1. Gradle Inc. URL: [https://docs.gradle.org/current/userguide/what\\_is\\_gradle.html](https://docs.gradle.org/current/userguide/what_is_gradle.html)

**spring-projects/spring-hateoas** (26. Jan. 2020a): spring-projects/spring-hateoas. Issues 731. GitHub, Inc. URL: <https://github.com/spring-projects/spring-hateoas/issues/731> (besucht am 26.01.2020)

**spring-projects/spring-hateoas** (26. Jan. 2020b): spring-projects/spring-hateoas. Issues 1094. GitHub, Inc. URL: <https://github.com/spring-projects/spring-hateoas/issues/1094> (besucht am 26.01.2020)

**springfox/springfox** (26. Jan. 2020): springfox/springfox. Issues 2932. Github, Inc. URL: <https://github.com/springfox/springfox/issues/2932> (besucht am 26.01.2020)



# Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich meinen Beitrag zur vorliegenden Gruppenarbeit (Kapitel 1, 2.2, 3.1, 3.3, 3.4, 4.1, 4.5, 4.6, 5.2, 6.1, 6.2, 6.3, 7.4, 8.1, 9.1, 9.2, 10) selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; das gleiche gilt für die von den auf dem Titelblatt der Arbeit genannten Autoren gemeinsam verfassten Teile (Kapitel 2.1, 3.2, 3.5, 3.6, 3.7, 4.2, 4.3, 4.5, 4.7, 5.1, 6.4, 6.5, 7.1, 7.2, 7.3, 8.2, 8.3, 9.3, 9.4, 9.5). Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hof, den 29.01.2020