

Entwicklung eines Studierendenportals als Progressive Web App mit Angular

Marcel Bastian

15. Oktober 2018
Version: Finale Fassung

Johannes Gutenberg-Universität Mainz



Institut für Informatik



Marcel Bastian

Entwicklung eines Studierendenportals als Progressive Web App mit Angular

Matrikelnr.: 2687696

Erstgutachter Prof. Dr. André Brinkmann
Zentrum für Datenverarbeitung
Johannes Gutenberg-Universität Mainz

Zweitgutachter Dr. Hans-Jürgen Schröder
Institut für Informatik
Johannes Gutenberg-Universität Mainz

15. Oktober 2018

Marcel Bastian

Entwicklung eines Studierendenportals als Progressive Web App mit Angular

Matrikelnr.: 2687696, 15. Oktober 2018

Gutachter: Prof. Dr. André Brinkmann und Dr. Hans-Jürgen Schröder

Johannes Gutenberg-Universität Mainz

Institut für Informatik

Staudingerweg 9b

55128 Mainz

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	2
1.3 Aufbau der Arbeit	2
2 Technologien, Frameworks/Libraries, Programmiersprachen	5
2.1 Progressive Web Apps	5
2.1.1 Manifest	7
2.1.2 Service Worker	8
2.1.3 Kompatibilität	9
2.2 Frameworks und Libraries: Angular, React und Vue	9
2.2.1 Angular	10
2.2.2 React	11
2.2.3 Vue.js	13
2.2.4 Warum Angular	14
2.3 Das Angular-Framework im Detail	15
2.3.1 Components	16
2.3.2 ngModules	19
2.3.3 Services/Injectables	21
2.3.4 Directives	22
2.3.5 Pipes	22
2.3.6 Data Binding	23
2.3.7 Observables	24
2.3.8 Routing	26
2.3.9 Zusammenfassung der Architektur	28
2.3.10 Das Befehlszeilen-Interface (CLI)	29
2.4 TypeScript	30
2.5 Sonstiges	30
3 Programmierung der Webanwendung	33
3.1 Inhalte	33
3.2 Besondere Schwierigkeiten	34
3.2.1 Same Origin Policy (SOP) - Cross-Origin Ressource Sharing (CORS)	34

3.2.2	JavaScript und XML	36
3.3	Die eigentliche Programmierung	36
3.3.1	Vorbereitungen	37
3.3.2	Das Grundgerüst der Anwendung	37
3.3.3	Spezielle Module und Services	40
3.4	Die eigentlichen Inhalte	41
3.4.1	Nachrichten	42
3.4.2	Campus-Karte	45
3.4.3	Busfahrpläne	49
3.4.4	Personensuche	53
3.4.5	Veranstaltungen	56
3.4.6	Speisepläne	59
3.4.7	Wetter	62
3.4.8	Öffnungszeiten	63
3.4.9	Authentifizierung	64
3.4.10	Bibliotheksausweis	65
3.4.11	PC-Pools	66
3.5	Umwandlung in eine Progressive Web App	67
4	Zusammenfassung	71
4.1	Rückblick	71
4.2	Ausblick	72
Literaturverzeichnis		75

Einleitung

“ *That's the thing about people who think they hate computers. What they really hate is lousy programmers.*

— Larry Niven
(Amerikanischer Science-Fiction Autor)

In diesem Kapitel wird zunächst die Motivation, welche hinter dieser Arbeit steht, erläutert, dann die Zielsetzung definiert und zuletzt ein Ausblick über Inhalt und Struktur gegeben.

1.1 Motivation

Im Rahmen des Studiums gibt es viel Wissenswertes rund um die Universität, das für Studierende - insbesondere für Studienanfänger - relevant oder zumindest interessant ist. Dazu zählen allgemeine Informationen wie Nachrichten über die Universität und deren Fachbereiche und Institute, Veranstaltungen, Gebäude, Personen etc., aber auch wichtige Daten wie z.B. Informationen zu Vorlesungen, Anmeldefristen oder Prüfungsterminen. All diese Informationen sind aktuell auf verschiedene Webseiten verteilt, sodass man leicht den Überblick verlieren kann, wo welche Informationen zu finden sind und welche überhaupt erhältlich sind.

So gibt es eine Seite der Universität, auf der über allgemeine Dinge wie beispielsweise Veranstaltungstermine, Nachrichten aus diversen Forschungsabteilungen oder Details für Studieninteressierte informiert wird. Dann verfügen die meisten Institute ebenfalls über eine eigene Webseite, auf der unter anderem Kontaktdaten der DozentInnen und MitarbeiterInnen, aber auch Details zu den angebotenen Lehrveranstaltungen gefunden werden können. Weiterhin gibt es die Webseite des Zentrums für Datenverarbeitung (ZDV), auf welcher die IT-Dienstleistungen der Universität dokumentiert sind, darunter etwa, wo es auf dem Campus PC-Pools oder Drucker gibt. Darüber hinaus gibt es dann noch Portale wie *JOGU-StINe*, der *JGU-Reader* oder Moodle, mit welchen die Anmeldung und Organisation von Vorlesungen, Seminaren, Übungen oder Prüfungen durchgeführt wird.

Um all diese Quellen sowie deren Informationen und Angebote zu bündeln, soll eine Webanwendung entwickelt werden, die es den Studierenden ermöglichen soll, einen Überblick über Interessantes, Wissenswertes und Wichtiges rund um die Universität und das Studium zu erhalten. Damit soll eine Erstanlaufstelle geschaffen werden, über welche man zukünftig (möglichst) alle Informationen rund um das Studium an der Johannes Gutenberg-Universität Mainz beziehen und selbiges organisieren können soll.

Gleichzeitig sollen neue Möglichkeiten im Bereich der Webentwicklungen untersucht werden, mit denen man eine solche Plattform realisieren kann. Das beinhaltet auch eine Analyse der damit lösbarsten Probleme als auch der Vorteile gegenüber bisherigen Vorgehensweisen. Des Weiteren soll eine Alternative zur derzeit entwickelten App "Uni-Mainz" geschaffen werden, damit später ein Vergleich der beiden Ansätze möglich ist.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, eine Webanwendung zu programmieren, welche Daten aus verschiedenen Quellen bündelt. Damit soll eine Anlaufstelle geschaffen werden, die übersichtlich darstellt, welche Informationen es gibt und - sofern sie nicht in der Anwendung selbst angeführt werden - wo diese zu finden sind. Diese Webanwendung soll als sogenannte Progressive Web App (PWA) implementiert werden. Als PWA werden Webanwendungen bezeichnet, die nicht nur in den gängigen Internetbrowsern auf verschiedenen Plattformen aufgerufen werden können, sondern (sofern dies vom verwendeten Betriebssystem unterstützt wird) auch als eigenständige Anwendung installiert werden können. Dadurch lässt sich eine Progressive Web App bedienen wie eine native Anwendung. Weitere Eigenschaften von PWAs werden in Kapitel 2.1 aufgelistet und erläutert. Daraus ergibt sich, dass die Webanwendung auf Desktop- und Mobilgeräten (möglichst) gleichermaßen benutzbar sein soll. Diese Webanwendung wird mit dem Framework "Angular" implementiert. Warum gerade dieses Framework verwendet wird und welche Alternative es gäbe, wird in Kapitel 2.2 diskutiert; eine ausführliche Beschreibung von Angular erfolgt in Kapitel 2.3.

1.3 Aufbau der Arbeit

Kapitel 2

Dieses Kapitel beinhaltet eine Übersicht über die verwendeten Technologien, die in Frage kommenden Frameworks/Libraries und die eingesetzten Programmiersprachen. Insbesondere wird hier das für dieses Projekt verwendete Framework Angular ausführlich beschrieben und diskutiert, welche Vorteile es gegenüber möglichen

Alternativen bietet. Außerdem werden zusätzlich genutzte Pakete und Ressourcen angeführt und kurz beschrieben.

Kapitel 3

Die Programmierung der Webanwendung sowie die dafür zu treffenden Vorbereitungen werden in Kapitel 3 wiedergegeben und geschildert. Dabei wird dort zunächst eine Übersicht über die Inhalte gegeben, die diese Webanwendung umfassen soll sowie besondere Hindernisse angeführt, die die Entwicklung etwas beeinflusst haben.

Kapitel 4

Abschließend fasst Kapitel 4 die Arbeit zusammen und gibt einen Ausblick darüber, welche Ziele erreicht, welche Bestandteile eventuell weiter ausgebaut und welche Inhalte der Webanwendung noch hinzugefügt werden könnten.

Technologien, Frameworks/Libraries, Programmiersprachen

„Users do not care about what is inside the box, as long as the box does what they need done.

— Jef Raskin

(US-Amerikanischer Informatiker, Experte für Mensch-Computer-Interaktion)

In diesem Kapitel werden die für diese Arbeit verwendeten Technologien aufgelistet und beschrieben. Zunächst wird das Konzept der *Progressive Web App* erklärt und erläutert welche Idee dahinter steckt. Dann werden mit Angular, React und Vue.js drei Frameworks beziehungsweise Libraries kurz vorgestellt und gleichermaßen deren Gemeinsamkeiten und Unterschiede als auch deren Vor- und Nachteile diskutiert. Anschließend wird begründet, warum die Wahl auf Angular gefallen ist und dieses Framework dann ausführlich beschrieben. Nach kurzer Vorstellung der Programmiersprache TypeScript, werden noch weitere Pakete und Materialien aufgelistet, welche für dieses Projekt verwendet wurden.

2.1 Progressive Web Apps

Mobile Plattformen - also Smartphones und Tablet-PCs - haben in den letzten Jahren stark an Bedeutung gewonnen und auch wenn Desktop-PCs ihre Daseinsberechtigung in ein paar Jahren nicht verloren haben werden, so werden sie immer mehr als Werkzeug zum “einfachen Surfen” von wesentlich kompakteren Geräten abgelöst. Wer sich lediglich ein paar Informationen aus dem Internet beschaffen oder Dienstleistungen über dasselbe in Anspruch nehmen will, der kann dies viel einfacher mit einem Smartphone erledigen. Zudem hat man dieses heutzutage fast immer und überall dabei.

Auch wenn mit der steigenden Popularität solcher mobilen Geräte das *Responsive Web Design*, bei dem sich das Layout einer Seite an das verwendete Gerät anpasst, immer wichtiger wurde, so waren *Native Apps* zunächst das Mittel der Wahl, um

Informationen auf mobilen Geräten darzustellen. Schließlich boten diese gegenüber den - zumindest in der Anfangszeit der Smartphones - eingeschränkten Funktionalitäten von mobilen Webbrowsern mehr Möglichkeiten und eine komfortablere Bedienung. Nachteil hierbei war (und ist), dass für die marktbeherrschenden Plattformen iOS von Apple und Android von Google auf Grund der unterschiedlichen Betriebssysteme unabhängige Versionen einer App entwickelt werden mussten; unter Umständen sogar noch für *Windows Mobile* oder *BlackBerry OS*, die allerdings beide in den letzten Jahren stark an Bedeutung verloren haben. Diese Mehrgleisigkeit brachte also einen Aufwand mit sich, der umso lästiger war, wenn weiterhin eine responsive Webseite betrieben werden sollte, die als Erstanlaufstelle dient.

Eine Lösung dieses Problems bestand in der Entwicklung von Werkzeugen, die es ermöglichen, eine App für mehrere Plattformen gleichzeitig zu entwickeln. Als Beispiel sei hier *Xamarin*[@1] genannt, eine Sammlung von Werkzeugen, mithilfe derer Apps in der Programmiersprache C# programmiert und diese dann für iOS, Android und Windows Phone exportiert werden können. Ein weiteres Beispiel ist das von Adobe veröffentlichte Framework *PhoneGap*[@2]. Hierbei werden Apps im Prinzip als responsive Webseiten mit HTML, CSS und JavaScript entwickelt, die dann auf den Endgeräten in einer Art virtueller Maschine laufen. Der Vorteil dieses und vergleichbarer Frameworks liegt ganz offensichtlich darin, dass Webentwickler, die nie zuvor native Apps entwickelt haben, nach relativ kurzer Einarbeitungszeit in der Lage sind, Apps für verschiedene Plattformen zu programmieren.

Ein Haken bleibt bei der Verwendung von Tools wie Cordova aber bestehen: Mit solchen Frameworks entwickelte Apps müssen weiterhin über den *Google Play Store* (im Falle von Android-Geräten), über den *App Store* (für iOS-Geräte) oder die entsprechende Vertriebsplattform ausgeliefert und installiert werden. Das ist zum einen mit Lizenzauflagen für die Entwickler verbunden und kann zum anderen ein Hemmnis für Nutzer darstellen, die nicht für jede Kleinigkeit eine App auf ihrem Gerät installieren wollen.

Hier kommt das Konzept der Progressive Web App (PWA) ins Spiel. Dabei handelt es sich um Webseiten, die Funktionen und Vorteile von responsiven Webseiten und nativen Apps bzw. Desktopanwendungen kombinieren. Alex Russel, der zusammen mit Frances Berriman das Konzept der PWAs ausgearbeitet hat nennt als Mindestanforderung die folgenden Kriterien: [@3]

- Sie funktionieren (zumindest eingeschränkt) auch ohne Internetverbindung.
- Sie werden über eine sichere Verbindung (also via HTTPS) ausgeliefert.
- Sie enthalten ein Manifest, dass bestimmte Eigenschaften der Webseite angibt.

Diesen Hauptkriterien müssen erfüllt sein, damit kompatible Browser den Benutzer zum Installieren der Webseite als Desktop- oder Mobile-App auffordern können. Darüber hinaus gibt es wünschenswerte Kriterien, wie zum Beispiel, dass PWAs möglichst schnell laden oder von Plattform und Browser unabhängig sind. Dabei heißt „schnell laden“ nicht zwangsläufig, dass die ganze Seite geladen worden ist, sondern das möglichst schnell ein rudimentärer Inhalt angezeigt wird, sodass der Nutzer eher geneigt ist etwas länger zu warten, bis die Seite dann vollständig aufgebaut wurde.

Eine PWA muss zu den eigentlichen HTML-, JavaScript- und CSS-Dateien noch eine Manifest-Datei und eine JavaScript-Datei, in welcher der „Service Worker“ implementiert wird, enthalten. In der Manifest-Datei werden bestimmte Eigenschaften angegeben und über den Service Worker werden die Funktionalitäten implementiert, welche eine Progressive Web App von einer normalen Webseite unterscheiden. Beides wird im Folgenden vorgestellt und erläutert.

2.1.1 Manifest

Bei der Manifest-Datei handelt es sich um eine Datei im JSON-Format, in der mindestens folgende Eigenschaften der PWA definiert werden müssen:[@4]

- Der Name der Web App (in Langform)
- Web App-Name in Kurzform für die Beschriftung der Verknüpfung
- Der Wurzelpfad der Webseite (damit bei „Installation“ der PWA nicht der aktuelle Pfad als Link gespeichert wird)
- Die Art, wie die „installierte“ Seite geöffnet werden soll: in einem normalen Browser-Fenster, in einem Browser-Fenster mit minimalen Steuerelementen, als eigenständige Anwendung in einem normalen Fenster oder im Vollbildmodus.
- Eine Liste von Icons für die Verknüpfung: typischerweise wird das Icon in mehreren Auflösungen hinterlegt, damit es auch auf hochauflösenden Geräten vernünftig dargestellt wird

Zusätzlich können hier Attribute definiert werden, wie Farbe für Hintergrund, sowie Fenster und Taskleiste auf Android-Geräten, Sprache und Leserichtung des Namens in Kurz- und Langform oder ein „Scope“, der angibt, welche Seiten zur Progressive Web App gehören und welche in einem normalen Browserfenster geöffnet werden sollen.

2.1.2 Service Worker

Die Hauptaufgabe des Service Worker besteht darin, Anfragen nach Ressourcen (Skripte, CSS-, XML-, JSON- oder sonstige Dateien), die eigentlich an einen Server geschickt werden sollen, abzufangen, und stattdessen im Cache gespeicherte Versionen der Dateien zu verwenden. Hierbei ist zu konfigurieren, welche Ressourcen überhaupt im Cache gespeichert werden sollen, wie lange sie dort maximal gespeichert werden dürfen, bevor sie als veraltet zu betrachten und durch neuere Versionen zu ersetzen sind, und wie viel Speicherplatz der Cache maximal belegen darf. Damit diese Funktionen genutzt werden können muss - nachdem beim Seitenaufruf alle Ressourcen geladen worden sind - überprüft werden, ob der verwendete Browser die Service Worker-API überhaupt unterstützt. Ist dies der Fall, kann das Skript, in dem der eigentliche Service Worker implementiert wird, registriert werden.

Durch das Speichern der Ressourcen im Cache kann die Webseite dann auch aufgerufen werden, wenn keine Internetverbindung vorhanden ist. Aber auch mit Internetverbindung ist so ein großer Performance-Gewinn möglich; eine Seite kann dann quasi augenblicklich geladen werden, weil auf lokal gespeicherte Ressourcen zurückgegriffen werden kann, statt auf die Antwort eines Servers warten zu müssen. Das kann vor allem auf Mobilgeräten von Vorteil sein, wenn nur eine schwache Mobilfunkverbindung verfügbar ist[@5].

Zudem können über den Service Worker sogenannte Push-Benachrichtigungen implementiert werden, da dieser immer im Hintergrund aktiv ist. Somit können die Benutzer über Neuigkeiten informiert werden, wie es auch bei nativen Anwendungen und Apps möglich ist. Es kommt sogar häufig vor, dass ein Service Worker nur für solche Benachrichtigungen genutzt und gar keine Daten im Cache gespeichert werden.

Außerdem unterstützt der Service Worker sogenanntes "Background Sync". Hierbei werden Daten, die an einen Server geschickt werden sollen, bei nicht vorhandener Internetverbindung gespeichert und sobald eine Verbindung wiederhergestellt wurde, dann versendet.

Geplant ist darüber hinaus, dass Service Worker in Zukunft auch periodisches Synchronisieren unterstützen, sodass in Intervallen, zu bestimmten Zeiten aber auch in Abhängigkeit des Akkuladestandes oder der Art der Netzwerkverbindung eine Kommunikation zwischen Server und Webanwendung erfolgt. Hierzu gibt es allerdings noch keine offizielle Spezifikation[@6].

2.1.3 Kompatibilität

Zwar sind Progressive Web Apps prinzipiell “nur“ Webseiten und als solche im Allgemeinen nicht vom Endgerät abhängig, aber noch unterstützen nicht alle Browser auf allen Plattformen die Technologie der Service Worker und auch die Manifest-Datei einer PWA wird noch von einigen Browsern (zumindest auf manchen Plattformen) ignoriert. Während sowohl auf Geräten mit den Betriebssystemen Windows 10, Android oder einer Linux-Distribution PWAs von den dort verfügbaren Browsern fast vollumfänglich unterstützt werden, gibt es auf Geräten mit dem Apple-Betriebssystemen iOS noch teils erhebliche Einschränkungen. Lediglich der Apple-eigene Safari-Browser unterstützt auf dieser Plattform die Service Worker-Technologie, aber selbst hier sind keine Push-Benachrichtigungen und keine Hintergrundaktualisierungen möglich. Auf MacOS-Geräten ist die Unterstützung des Safari-Browsers ähnlich schwach, allerdings sind sowohl Chrome als auch Opera (bis auf das Installieren als Desktop-Anwendung) vollständig kompatibel.[@7]

2.2 Frameworks und Libraries: Angular, React und View

Beim Entwickeln von Software jeglicher Art sind Frameworks und Libraries weitverbreitete Hilfsmittel, um nicht jede einzelne Funktion neu programmieren zu müssen, sondern auf vorhandene Komponenten zurückgreifen zu können, sodass viel Arbeit gespart werden kann. Auch wenn die Bezeichnungen “Framework“ und “Library“ vor allem im Bereich der Webentwicklung häufig synonym verwendet werden, kann man doch differenzieren, was die Funktions- bzw. Verwendungsweise betrifft. Allgemeiner Konsens ist: Eine *Library* wird verwendet, um Lücken in der eigenen Anwendung zu schließen, während man bei Verwendung eines *Frameworks* die Lücken desselben füllen muss. Genauer bedeutet dies, dass man die Funktionen einer Library in der eigene Anwendung aufrufen muss, während man bei einem Framework den eigenen Code in ein vorgegebenes Grundgerüst einbinden muss[@8].

Im Folgenden werden mit Angular, React und Vue drei derzeit sehr populäre Frameworks/Libraries vorgestellt und miteinander verglichen[@9] und anschließend begründet, warum Angular für dieses Projekt verwendet wird. Bei allen drei handelt es sich dabei um unter MIT-Lizenz veröffentlichte *Open Source*-Software, die zum Programmieren von Benutzerschnittstellen konzipiert ist. Außerdem sind alle drei “komponentenbasiert“; man entwickelt *Components*, die als Bausteinen dienen, um damit eine Webseite aufzubauen.

2.2.1 Angular

Das von Google entwickelte Framework erschien ursprünglich im Oktober 2010 unter dem Namen “AngularJS”; mit der Veröffentlichung der Version 2.0.0 im September 2016, die eine grundlegende Überarbeitung mit sich brachte, wurde der Name in “Angular“ geändert, um sich klar von den alten Versionen zu distanzieren, schließlich sind diese nicht miteinander kompatibel[@10].

Während AngularJS in JavaScript geschrieben wurde, basiert Angular auf der von Microsoft entwickelten Programmiersprache TypeScript. Diese Programmiersprache wird in Kapitel 2.4 kurz vorgestellt. Verwendet man also Angular empfiehlt es sich, in TypeScript zu programmieren, auch wenn es grundsätzlich möglich ist, weiterhin JavaScript zu nutzen. Die offizielle Dokumentation beschränkt sich allerdings auf die Verwendung von TypeScript. Angular baut auf der in JavaScript programmierten *NodeJS*-Serverumgebung auf und benötigt den Paketverwaltungsmanager *npm*, mit welchem dem Projekt ganz einfach weitere Softwarepakete hinzugefügt werden können.

Ein Nachteil von Angular ist, dass es relativ umständlich und schwierig ist, mit diesem Framework entwickelte Komponenten in eine bestehende Webseite einzufügen. Während es durchaus Anleitungen gibt, welche beschreiben wie das möglich ist, beschränkt sich die offizielle Dokumentation darauf, zu erklären, wie mit Angular *Single-Page Applications (SPAs)* erstellt werden können. Außerdem ist Angular auch zu umfangreich, als dass es sinnvoll wäre, dieses Framework “nur” für ein paar Komponenten zu einer bestehenden Webseite hinzuzufügen. Es einfach eher dafür konzipiert, eigenständig verwendet zu werden. Allerdings ist es durchaus möglich, eine Angular-Web-App in einem Unterverzeichnis einer bestehenden Webseite einzubinden - dann aber wieder als in sich geschlossene Einheit.

Auch wenn Angular an sich schon relativ komplex und umfangreich ist - was sich auch auf die Dokumentation des Frameworks auswirkt -, können mit dem *ng add*-Befehl des Angular-CLI ganz einfach neue Pakete und Funktionalitäten hinzugefügt werden. Das Angular-CLI wurde mit Version 2.0.0 von Angular eingeführt, ist aber nicht fester Teil von Angular, sondern muss manuell installiert werden. Mit diesem Command-Line Interface können u.a. die verschiedenen Komponenten von Angular generiert werden, aber auch ein Testserver mit *ng serve* gestartet oder mit *ng build* der *Build-Prozess* initiiert werden [@11]. Eine ausführlichere Beschreibung erfolgt in Kapitel 2.3.10.

Vorteilhaft sind die vom Entwicklerteam von Angular selbst verwalteten Pakete wie der Router oder der HTTP-Client zur Kommunikation mit einer API, die bei anderen Frameworks oder Libraries von Drittquellen bezogen werden müssen. Dadurch kann

Code-Ausschnitt 2.1: Code-Beispiel für React mit einem *Functional Component*

```
1 function Welcome(props) {
2   return <h1>Hello, {props.name}</h1>;
3 }
4
5 const element = <Welcome name="Sara" />;
6 ReactDOM.render(
7   element,
8   document.getElementById('root')
9 );
```

eine höhere Stabilität und Sicherheit gewährleistet werden und bei Updates gibt es keine Kompatibilitätsprobleme. Weitere Details und Eigenschaften von Angular werden in Kapitel 2.3 erläutert.

2.2.2 React

Mit Facebook steht hinter React ebenfalls ein großes Unternehmen; bei diesem Projekt handelt es sich jedoch um eine *Library*. Diese wurde erstmalig im Mai 2013 veröffentlicht und kann - im Gegensatz zu Angular - ganz einfach zu bestehenden Projekten hinzugefügt werden. Es wurde aber auch schon vor der Veröffentlichung von Facebook selbst seit 2011 genutzt. Zwar wird auch hier ein komponentenbasierter Ansatz verfolgt, allerdings sind die Komponenten bei React oftmals kleiner und nicht so umfangreich, da sie lediglich einzelne Elemente repräsentieren. Wie bei Angular können auch bei React Komponenten ineinander verschachtelt werden.

React verwendet JavaScript XML (JSX), eine Erweiterung von JavaScript, bei der im Prinzip HTML-Code in Javascript-Code eingebettet wird[@12]. Im Gegensatz dazu werden bei Angular die *HTML-Templates* von Komponenten in der Regel in eigene Dateien ausgelagert (siehe Kapitel refsec:technologies:angular:component:html), was bei größeren *Templates* von Vorteil ist. und auch dem Konzept des *Separation of Concerns* entspricht. Allerdings sind Komponenten in der Regel so gestaltet, dass deren Template überschaubar ist. Anders als Angular ist React nicht abhängig von *NodeJs* oder *npm*. Es gibt aber dennoch ein CLI-Paket, das über *npm* bezogen werden kann, mit dem unter anderem ein neues Projekt erzeugt oder dieses für die Auslieferung “gepackt“ werden kann[@13].

Diese Library ist nicht so komplex wie Angular, dementsprechend ist auch die Lernkurve weniger steil. Da diese Library eher für das Erstellen von Benutzer-Interfaces gedacht ist, fehlen Funktionalitäten wie z.B. ein Router zum Navigieren zwischen “virtuellen Seiten“ oder ein HTTP-Client. Dafür gibt es dann Pakete aus einer aktiven Community, die diese Features nachliefern.

Code-Ausschnitt 2.2: Code-Beispiel für ein *Class Component* bei React

```
1 class Welcome extends React.Component {  
2   constructor() {  
3     /* Initialisierungen */  
4   }  
5   render() {  
6     return <h1 onClick={() => this.handleClick()}>Hello {  
7       this.props.name }</h1>  
8   }  
9   handleClick() {  
10    /* Auf 'Klick' reagieren */  
11  }  
12}  
13 const element = <Welcome name="Sara" />;  
14 ReactDOM.render(element, document.getElementById('root'));
```

Ein weiterer Unterschied zu Angular ist der, dass diese Library ganz einfach zu einer bestehenden Webseite hinzugefügt werden kann, indem der Quellcode von React und ReactDOM (über das die Manipulation des Document Object Model (DOM) geregelt wird) mittels *Script*-Tag in das HTML-Dokument eingebunden wird. Dann können Komponenten in einer oder mehreren JavaScript-Dateien implementiert werden[@14].

Anders als Angular verwendet React das Konzept des sogenannten *Virtual DOM*. Hierbei wird ein virtuelles Abbild des eigentlichen DOMs erzeugt und bei Änderung von Daten des zugrunde liegenden Modells bestimmt, welche Teile des DOMs aktualisiert werden müssen. Da es sich bei dem DOM um eine Datenstruktur in Form eines Baumes handelt, ist das Durchsuchen eines solchen zwar relativ schnell möglich, das Aktualisieren und Umstrukturieren kann aber mitunter sehr aufwändig sein. Mit Hilfe des Virtual DOMs sowie eines speziellen Algorithmus kann React diese Prozedur beschleunigen[@15].

Im Code-Beispiel 2.1 wird gezeigt, wie eine simple React-Komponente aussehen kann[@16]. Das in Zeile 5 definierte Element wird beim *rendern* der Seite als Kindelement des DOM-Elements mit der ID 'root' in das DOM eingebunden (Zeile 6). Dadurch, dass als Wert des *name*-Attributes des Elements "Sara" festgelegt wurde, bekommt die Seite also die Überschrift "Hello, Sarah". In Zeile 1-3 ist ein Beispiel für *JSX* geben; JavaScript und HTML werden hier miteinander vermischt. Soll die Komponente zusätzliche Funktionen besitzen, um beispielsweise das Anklicken dieser zu verarbeiten, so würde man eine Klasse definieren, die von *React.Component* "erbt". Das *return-Statement* würde dann in eine *render*-Methode verschoben. Code-Beispiel 2.2 zeigt, wie das aussehen kann. Entscheidend ist hier die *render*-Methode, die definiert sein muss, weil sie aufgerufen wird, wenn die Komponente ins DOM eingebunden werden soll und daher als Rückgabe ein Template liefern muss.

Code-Ausschnitt 2.3: Code-Beispiel für ein *Component* in Vue.js

```
1  Vue.component('heading', {  
2    data: function() {  
3      [...]  
4      return {name: user.name};  
5    },  
6    template: '<h1>Wilkommen, {{ name }}</h1>'  
7  });  
8 new Vue({el: '#app'});
```

2.2.3 Vue.js

Zwar handelt es sich bei Vue.js (gesprochen wie “view“) wie bei Angular um ein Framework, dieses kann jedoch im Gegensatz zu Angular ohne große Probleme in bestehende Webseiten bzw. Webanwendungen eingebunden werden. Das funktioniert analog zur Vorgehensweise von React. Vue.js ist, vor allem im Vergleich mit Angular, sehr kompakt und deswegen besser geeignet, wenn lediglich ein paar interaktive, dynamische Elemente ergänzt werden sollen.

Anders als bei React und Angular steht hinter diesem Projekt kein großes Unternehmen, sondern ein vergleichsweise kleines Team. Erstmalig veröffentlicht wurde diese Library im Februar 2014[@17]. Damit ist Vue.js, das zunächst vom ehemaligen Google-Mitarbeiter Evan You alleine entwickelt wurde - auch heute noch leitet er das Projekt -, jünger als Angular und React. Aufgrund des kleinen Entwicklerteams und weil es noch nicht ganz so alt ist, ist diese Library jedoch noch nicht ganz so umfangreich. Während zum Beispiel wie bei React ein HTTP-Client fehlt, gibt es bei Vue.js ein offizielles Routing-Paket. Ähnlich wie bei React gibt es auch hier eine sehr aktive Community, die eigene Pakete entwickelt, sodass das Framework bei Bedarf um weitere Funktionalitäten erweitert werden kann. Eine weitere Gemeinsamkeit von React und Vue.js ist die Verwendung eines *Virtual DOM*.

Wie eine Komponente bei Vue.js definiert werden kann, ist in Beispiel 2.3 dargestellt[@18]. Hier wird eine Vue-Komponente erzeugt, deren Template jedes DOM-Element mit dem Tag-Namen “heading“ ersetzt. Das funktioniert allerdings erst, wenn wie hier in Zeile 8 ein Scope definiert wird. Sollte es außerhalb des Elements mit der ID “app“ Elemente mit dem Tag-Namen “heading“ geben, werden diese nicht ersetzt.

Bei großen Projekten bzw. komplexen Komponenten empfiehlt es sich, sogenannte *Single File Components* zu verwenden. Hierbei wurde einen Zwischenweg von Angular und React gewählt: Sowohl Template- als auch Skript-Code befinden sich in einer Datei mit der Endung .vue , jedoch werden diese beiden nicht wie bei der JSX-Syntax von React miteinander vermischt. Stattdessen stehen in einer .vue-Datei

Code-Ausschnitt 2.4: Code-Beispiel für ein *Single File Component* in Vue.js

```
1 <template>
2   <h1>Wilkommen, {{ name }}</h1>
3 </template>
4 <script>
5 /*...*/
6 module.exports = {
7   data: function() {
8     /*...*/
9     return { name: user.name };
10  }
11 }
12 </script>
13 <style>
14 h1 {
15   color: blue;
16 }
17 </style>
```

der Template-Code und der JavaScript-Code (alternativ auch TypeScript-Code) in getrennten Abschnitten und es können noch CSS-Regeln, die nur für diese Komponente gelten, ergänzt werden. Diese Abschnitte werden durch XML-artige Tags ausgezeichnet. Durch die Nutzung dieses speziellen Datei-Typs sind IDEs zudem in der Lage, besseres *Syntax-Highlighting* zu liefern. Das Code-Beispiel 2.4 zeigt, wie eine solche Datei aussehen kann. Will man allerdings solche *Single File Components* verwenden, dann ist man auf *Webpack* angewiesen, mit dem die einzelnen Dateien in einer einzigen zusammengefasst werden. Dieses Werkzeug ermöglicht generell das Aufteilen von Code in Module, die dann vor dem Ausliefern gebündelt werden und wird ebenfalls von Angular verwendet.

Die Reihenfolge der Abschnitte ist dabei egal, es ist allerdings empfehlenswert, in allen Dateien die gleiche Reihenfolge beizubehalten[@19].

2.2.4 Warum Angular

Entscheidet man sich bei der Entwicklung eines Projektes für die Verwendung eines Frameworks oder einer Library, so können mehrere Faktoren für die Wahl wichtig sein. Generell sollte man sich überlegen, wie groß der Anteil der genutzten Funktionalitäten ist und ob sich auf Grund dessen der Einsatz überhaupt lohnt. Insbesondere sollte der Mehraufwand, welcher durch die Verwendung eines Frameworks entsteht, in einem vernünftigen Verhältnis zum eigentlichen Projektumfang stehen. Bei der Wahl eines Frameworks sollte man im Hinterkopf behalten, dass man sich einer (oftmals) starren Struktur fügen muss, während man bei einer Library in der Regel flexibler arbeiten kann. Weiterhin ist ein Abgleich der an das Projekt gestellten Anforderungen und den vom Framework bzw. der Library angebotenen Möglichkeiten sinnvoll.

Ein weiteres Kriterium auf das man bei der Auswahl Wert legen kann, ist die Populärität. Je weiter verbreitet ein Framework oder eine Library ist, desto einfacher ist es, bei Schwierigkeiten Hilfe in entsprechenden Foren zu bekommen und umso wahrscheinlicher ist es, dass bereits jemand das selbe Problem hat oder hatte. Außerdem gibt es bei populäreren Projekten auch mehr zusätzliches Material zur Einarbeitung. Weiterhin kann es auch von Vorteil sein, wenn ein Framework oder eine Library von einem größeren Unternehmen entwickelt wird, weil dieses zum einen schneller bestehende Fehler beseitigen kann und zum anderen eine gewisse Zukunftssicherheit mit sich bringt: Während es bei "Ein-Mann-Projekten" durchaus passieren kann, dass dieses quasi unvermittelt eingestellt wird, ist das bei populären Frameworks und Libraries, hinter denen eine größere Firma steht, eher unwahrscheinlich.

Auch wenn Angular recht komplex und umfangreich ist und zudem auch eine Einarbeitung in die Programmiersprache TypeScript notwendig bzw empfehlenswert ist, gibt es dennoch überzeugende Gründe, welche für dieses Framework sprechen. Zunächst einmal ist es von Vorteil, dass Angular einen HTTP-Client mitbringt, der für die Nutzung der verschiedenen APIs notwendig ist. Weiterhin ist Angular für die Entwicklung von SPAs durch das ebenfalls vom Angular-Entwicklerteam gepflegte Router-Paket prädestiniert. Dies ist zwar keine zwingende Anforderung an die zu entwickelnde Webanwendung, es ist allerdings einfacher, eine Single-Page Application als Progressive Web App umzusetzen als eine herkömmliche Webseite, da unter anderem weniger HTML-Dokumente im Cache gespeichert werden müssen. Außerdem soll sich eine PWA möglichst schnell anfühlen, was mit einer SPA erreicht wird, indem nur Teile einer Webseite neu geladen werden (mehr dazu in Kapitel 2.3.8). Bei React wird mehr Wert darauf gelegt eine Möglichkeit zu bieten, interaktive und dynamische Elemente zu einer Seite hinzufügen zu können; Router und HTTP-Client, gibt es beide nur aus Dritt-Quellen. Die Entwickler von Vue.js bieten zumindest ein offizielles Routing-Paket an.

Darüber hinaus kann mit dem vom Angular-Team entwickelten Paket `@angular/pwa` mit dem Angular-CLI-Befehl `ng add` einem Projekt hinzugefügt werden, sodass aus diesem ganz einfach eine Progressive Web App gemacht wird - eine Kernanforderung dieses Projekts.

2.3 Das Angular-Framework im Detail

In diesem Kapitel wird das Angular-Framework und seine Funktionsweise detaillierter vorgestellt. Dabei wird auch erklärt, aus welchen Bausteinen eine mit Angular entwickelte Web-Anwendung besteht und welche Funktion(en) diese erfüllen. Sämtliche Beschreibungen beziehen sich hierbei auf die Verwendung von TypeScript. Darüber hinaus können natürlich auch Klassen oder Interfaces implementiert wer-

den, bei denen es sich nicht um Angular-spezifische Elemente handelt, sondern um solche, die dem Modellieren und Verarbeiten der empfangenen oder zu sendenden Daten dienen.

Abgesehen von einem generell notwendigen *HTML*-Dokument, typischerweise mit dem Dateinamen *index.html*, und sonstigen *Stylsheets* oder Skripten (Analytics, Werbeanzeigen etc.) besteht eine Angular-Web-App mindestens aus einem *NgModule* (vgl Kapitel 2.3.2) und einem *Component*.

2.3.1 Components

Sogenannte *Components* - also Komponenten - sind die Hauptbausteine einer jeden Angular-Web-App. Eine solche Komponente kann ein rechtes einfaches Element wie Textbausteine (vergleichbar mit *div*- oder *p*-Tags) sein, Bedienelemente wie Buttons oder Textfelder repräsentieren, aber auch aus mehreren solcher Elemente zusammengesetzt und damit quasi eine ganze Seite repräsentieren. Streng genommen bezieht sich der Ausdruck “Component” dabei auf eine spezielle Klasse, zu der ein HTML-Template gehört, welches über Attribute und Methoden der Klasse manipuliert werden kann. Da das Template Teil des DOMs ist, ändert sich damit auch die Seite. Außerdem kann eine Komponente auch aus anderen Komponenten zusammengesetzt sein. Eine Komponente muss dabei immer Teil eines Moduls sein, also im *imports*-Array eines Moduls auftauchen - dazu mehr in Abschnitt 2.3.2.

2.3.1.1 Komponente.component.ts

Definiert wird eine Komponente in einer Datei mit der Endung “.component.ts”, wobei die Dateiendung *.ts* für TypeScript steht. Die Hauptkomponente, welche die ganze Web-App umfasst, wird beispielsweise standardmäßig in einer Datei mit dem Namen *app.component.ts* implementiert.

Der Code-Ausschnitt 2.5 zeigt, wie eine solche Datei aufgebaut ist. Zunächst erfolgen die Importe von notwendigen Modulen, Komponenten/Direktiven, und Services. Danach folgt der *Decorator*, durch den festgelegt wird, dass die anschließend implementierte Klasse ein *Component* ist, in dem aber auch verschiedene *Optionsmetadaten* definiert werden (können). Die wichtigsten davon sind:

- **selector:** Dieser CSS-Selektor gibt das DOM-Element an, zu dem das Template der Komponente als Kindelement hinzugefügt werden soll.
- **template/templateUrl:** Hier wird das zugehörige Template bzw. ein Pfad zu einer Datei, in der das Template definiert ist, angegeben.

Code-Ausschnitt 2.5: Beispiel für eine `AppComponent.ts`-Datei

```
1 // Importe anderer Module/Komponenten/...
2 import {Component, OnInit} from '@angular/core';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.css']
8 })
9 export class AppComponent implements OnInit {
10   // Klassenvariablen
11   heading = 'Willkommen';
12   names = ['Alice, Bob, Charlie'];
13
14   constructor() { }
15
16   ngOnInit() {
17     // Initialisierungen
18   }
19 }
```

- **style/styleUrls:** Die CSS-Regeln für das Design der Komponente bzw. ein Liste von Pfaden zu CSS-Dateien für diese Komponente werden hier aufgelistet.

Außerdem können Services, die von dieser Komponente genutzt werden, angegeben werden, falls dies nicht schon in einem übergeordneten Modul organisiert wird. Anschließend wird die eigentliche Klasse der Komponente angegeben. Damit diese an anderer Stelle *importiert* werden kann, muss sie hier auch *exportiert* werden. Das ist allerdings keine Eigenheit von Angular, sondern unterliegt der generellen Funktionsweise von JavaScript beziehungsweise TypeScript.

Typischerweise gibt es in der Klasse der Komponente eine Funktion `ngOnInit`, über welche die Initialisierung der Instanz der Komponente vorgenommen wird. Zum Beispiel können Daten von einem Server angefragt werden, um sie dann in die Komponente zu integrieren. Zwar kann auch eine `constructor`-Methode angegeben werden, diese ist allerdings hauptsächlich für *Dependency Injection* zuständig, also das Einbinden von Objekten, die nicht vom einbindenden Objekt selbst instanziert werden. Ansonsten werden, wie bei objektorientierter Programmierung üblich, Instanzvariablen und Methoden definiert[@20].

2.3.1.2 `Komponente.component.html`

Ist das *Template* der Komponente zu groß, wird es der Übersicht halber ausgelagert in eine Datei mit der Endung `.component.html`. Mittels *Two-Way Data Binding* kann bei Angular das DOM mit einer Komponente verknüpft werden, sodass sich das

Code-Ausschnitt 2.6: Beispiel für eine index.html-Datei einer Angular-App

```
1  <!doctype html>
2  <head>
3  <!--Meta-Tags, Stylesheets, Skripte...-->
4  </head>
5  <body>
6    <app-root>
7      <!-- Die Templates der Komponenten werden hier eingesetzt -->
8    </app-root>
9    <noscript>
10      Du musst JavaScript aktiviert haben, um diese Seite zu
11      verwenden.
12    </noscript>
13    <!-- Automatisch generierte Skript-Dateien werden hier
14      eingebunden -->
15  </body>
```

ersteres ändert, wenn sich die Komponente bzw. der Wert einer Instanzvariablen der Komponente ändert und umgekehrt.

Der *selector* im in Code-Ausschnitt 2.5, deutet an, dass es sich dabei um das *Root Component* - also die Wurzelkomponente - handelt, welche als Container für die ganze Web-App dient. Dieser Selektor bezieht sich auf ein HTML-Element mit dem Tag-Namen “app-component” wie zum Beispiel in Zeile 6 in Beispiel 2.6. Innerhalb dieses Elements wird dann das Template der Komponente eingesetzt, wobei in diesem wiederum weitere Komponenten bzw. deren Templates eingebunden werden können.

Der *noscript*-Tag ist dabei nicht zwingend notwendig, aber sinnvoll, um den Benutzer darauf hinzuweisen, dass die Seite ohne JavaScript nicht funktioniert. Vor dem schließenden *body*-Tag werden dann beim *Build*-Prozess die von Angular generierten Skripte importiert.

Ein *Template* für die Komponente aus Beispiel 2.5 würde in einer Datei namens *app.component.html* definiert und könnte aussehen wie in dem Code-Ausschnitt 2.7.

Hier fallen zunächst einmal doppelten geschweiften Klammern auf. Eine ähnliche Syntax findet sich auch bei Vue.js, React und anderen vergleichbaren Frameworks und Libraries und dient der Interpolation von Ausdrücken. Wenn beim Initialisieren der Komponente das *Template* gerendert wird, so wird der Ausdruck in Zeile 1 durch den Wert der Variablen *heading* ersetzt. Bei einem solchen Ausdruck kann es sich auch um einen Funktionsaufruf aber auch um arithmetische Operationen handeln. Weiterhin können solche Interpolationsausdrücke auch bei Attributzuweisungen verwendet werden, um beispielsweise die Farbe eines Elements vom Wert einer Va-

Code-Ausschnitt 2.7: Beispiel für ein *Template*

```
1 <h1>{{heading}}</h1>
2 <ul *ngIf="users.length>0; else #noNames">
3   <li *ngFor="let name of names">
4     {{name}}
5   </li>
6 </ul>
7 <ng-template #noNames>
8   <p>
9     Keine Nutzer vorhanden
10  </p>
11 <ng-template>
```

riblen abhängig zu machen. Sowohl Variablen als auch Funktionen müssen dabei Teil der Komponenten-Klasse sein und dürfen nicht als *private* deklariert sein[@21].

Weiterhin fallen die Ausdrücke **ngIf* und **ngFor* auf. Bei diesen Ausdrücken handelt es sich um sogenannte *structural directives*. Neben *structural directives* gibt es auch *attribute directives*. Beide werden in Kapitel 2.3.4 erläutert.

2.3.1.3 Komponente.component.css

Bei großen Komponenten kommt es vor, dass es viele CSS-Regeln gibt, sodass es sinnvoll ist, diese in eine eigene - oder gar mehrere - Datei(en) auszulagern. Dabei können neben dem Standard CSS auch *Sass* oder *Less* als Auszeichnungssprache genutzt werden. Eine Besonderheit hierbei ist, dass die in den Metadaten einer Komponente definierten CSS-Regeln nur auf das Template der Komponente selbst angewandt werden. Das bedeutet, dass in verschiedenen CSS-Dateien, die zu unterschiedlichen Komponenten gehören, ein und derselbe CSS-Selektor genutzt werden kann, um gleichen Attributen unterschiedliche Werte zuzuweisen. Für eine Komponente kann also mit dem CSS-Selektor *.header* dem Attribut *color* der Wert *Rot* zugeordnet werden, während mit dem selben Selektor in einer anderen Datei das Attribut *color* mit dem Wert *Blau* definiert wird, sodass für die eine Komponente Rot und für die andere Blau verwendet wird. Normalweise würden eine der beiden Regeln überschrieben, sodass beiden Komponenten dieselbe Farbe zugeordnet wird.

2.3.2 ngModules

Eine mit Angular entwickelte Web-App kann in mehrere Module unterteilt werden, was zum einen der Struktur und damit der Übersichtlichkeit dient und zum anderen *Lazy Loading* ermöglicht. Dabei enthält eine solche Anwendung mindestens ein Modul, das sogenannte *Root Module*, welches die Wurzel der Anwendung darstellt.

Code-Ausschnitt 2.8: Beispiel für eine *App.module.ts*-Datei

```
1 // Importe anderer Module, Komponenten oder Services
2 import { NgModule } from '@angular/core';
3 import { BrowserModule } from '@angular/platform-browser';
4 import { FormsModule } from '@angular/forms';
5 import { HttpClientModule } from '@angular/common/http';
6 import { AppComponent } from './app.component';
7 import { AppService } from './app.service';
8
9 @NgModule({
10   declarations: [
11     AppComponent
12   ],
13   imports: [
14     BrowserModule,
15     FormsModule,
16     HttpClientModule
17   ],
18   exports: [],
19   providers: [AppService],
20   bootstrap: [AppComponent]
21 })
22 export class AppModule { }
```

Über dieses Modul, meist in einer Datei mit dem Namen *app.module.ts* implementiert, werden alle weiteren Module (in Dateien der Endung *.module.ts*) und die jeweils zugehörigen Komponenten geladen.

Eine Single-Page Application funktioniert normalerweise so, dass beim ersten Seitenaufruf ein HTML-Dokument samt Skripten und Stylesheets übertragen wird und beim Anklicken von Links dann kein neues HTML-Dokument angefordert wird, sondern lediglich mittels *Templates* Teile des DOMs ersetzt werden. Damit wirkt es auf den Benutzer, als würde eine neue Seite augenblicklich geladen. Bei einer umfangreichen Webanwendung dieser Art kann es wünschenswert sein, diese in mehrere Module zu unterteilen und nur das Modul vom Server zu laden, das gerade aktiv ist. Diese als *Lazy Loading* bezeichnete Technik stellt einen Kompromiss dar, bei dem beim Übergang zwischen Seiten, die nicht zum selben Modul gehören, zwar eine umfangreichere Datenübertragung zwischen Server und Client stattfindet, der initiale Seitenaufruf aber etwas beschleunigt werden kann.

Beispiel 2.8 zeigt das *Root Module*, anhand dessen der Aufbau eines *ngModule* erklärt wird. Auch hier werden über einen *Decorator* mehrere Metadaten konfiguriert und die Klasse als *ngModule* ausgezeichnet. Unter *declarations* werden zu diesem Modul gehörenden Komponenten, Direktiven und Pipes aufgelistet. Das *imports*-Array gibt an, welche anderen Module in diesem Modul verwendet werden, sei es durch zugehörige Komponenten, Direktiven oder Pipes. Sollen Komponenten, Pipes oder Direktiven dieses Moduls auch in anderen Modulen genutzt werden können, müssen sie im *exports*-Array enthalten sein. Danach werden mit *providers* die Services

aufgelistet, welche in diesem Modul bereitgestellt werden. Services, die hier gelistet werden, sind dann in allen untergeordneten Modulen und Komponenten verfügbar. Als letztes werden diejenigen Komponenten im *bootstrap*-Array aufgelistet, welche beim Initialisieren des Moduls generiert werden sollen. Das *Root Module* nennt hier üblicherweise nur eine Komponente, welche die ganze Anwendung umfasst, während in anderen Modulen mehrere angegeben werden können. In der Klasse selbst werden dann üblicherweise keine Methoden oder Variablen deklariert[@22].

2.3.3 Services/Injectables

Services werden zum einen von Komponenten für den Datenaustausch mit einem Server genutzt, sodass beispielsweise Daten über eine API bezogen werden können. Zum anderen kommen sie zum Einsatz, wenn Komponenten untereinander kommunizieren sollen. Dies geschieht, indem eine Komponente ein *Event* auslöst, sodass über einen Service ein *Event Listener* einer anderen Komponente getriggert wird. Eine solche Service-Klasse wird durch den *Decorator* “@Injectable“ ausgezeichnet, welcher deklariert, dass diese Klasse Teil des *Dependency Injection*-Systems ist[@23]. Im Kern dient *Dependency Injection* dazu, dass verschiedene Klassen dieselbe Instanz einer anderen Klasse nutzen können. Wenn also eine Klasse die Dienste eines Services benötigt, instanziert sie diesen nicht selbst, sondern dem Konstruktor der Klasse wird eine Referenz zu einer Instanz des Services übergeben. Wäre das nicht der Fall, wäre beiden Klassen eine eigene Instanz des Services zugewiesen, wodurch sich die Zustände dieser unterscheiden könnten, was zu inkonsistenten Daten führt. Damit wird eine Umkehr der Abhängigkeiten (Inversion of Control, IoC) erreicht, weil der Zustand des Services nicht von dem der Klassen (bzw. deren Instanzen) abhängig ist, sondern umgekehrt[@24].

Auch wenn es nicht zwingend notwendig ist, diese Funktionalitäten in Services auszulagern, so ist es dennoch empfehlenswert, weil sie nicht direkt Teil der Komponenten sind. Da es außerdem möglich ist, dass mehrere Komponenten den selben Service nutzen können, kann somit die Dopplung von Code vermieden werden. Zudem können in einem Service Daten “gespeichert“ werden, was von Vorteil ist, wenn mehrere Komponenten dieselben Daten nutzen. In diesem Fall müssen die Daten nicht neu von einem Server angefordert werden, sondern es können die genutzt werden, die bereits von einer anderen Komponente bezogen worden sind.

Wird ein Service als Provider im *Root Moudle* deklariert, so bekommen alle Komponenten, die diesen Service nutzten, die selbe Instanz dieses Services zugewiesen. Wenn ein Service als Provider in einem anderen Modul deklariert wird, dann können nur die Komponenten darauf zugreifen, die auch dieses Modul importieren, der Service ist dann also an das Modul gebunden. Außerdem kann ein Service auch als

Provider auf Komponenten-Ebene definiert werden. Dann wird jeder Instanz der Komponente eine eigene Instanz des Services zugeordnet.

2.3.4 Directives

Wie in Kapitel 2.3.1.1 angedeutet, gibt es verschiedene Arten von Direktiven: strukturelle Direktiven, „Attributdirektive“ und die bereits vorgestellten *Components*. Dabei stellen letztere einen Spezialfall dar, weil es sich um eine Unterklasse von Direktiven handelt und daher werden diese nicht durch den „@Directive“-Decorator ausgezeichnet.

Ein paar strukturelle Direktiven wurden bereits vorgestellt: Die *ngIf-Direktive wird genutzt, um Teile eines Templates nur unter bestimmten Bedingungen anzuzeigen. In Beispiel 2.7 wird das *div*-Element in Zeile 2 beispielsweise nur dann angezeigt, wenn die Länge des Arrays *names* größer als Null ist. Andernfalls wird das ab der neunten Zeile definierte Template ins DOM eingebunden. Ist die Liste der Namen nicht leer, iteriert die *ngFor-Direktive (Zeile 4) über die Elemente des Arrays und somit wird jeder Name als Element einer ungeordneten Liste ausgegeben. Eine weitere strukturelle Direktive ist *ngSwitch, bei der in Abhängigkeit des Wertes einer Variablen unterschiedliche *Templates* ins DOM eingebunden werden.

Diese *Structural Directives* dienen also dazu, die Struktur und den Aufbau des DOMs zu verändern. Es sei hier noch angemerkt, dass falls das Array der Namen leer ist und somit keine Liste angezeigt wird, das ganze *ul*-Element nicht Teil des DOMs ist. Umgekehrt gilt dasselbe für das *ng-template*-Element, falls die Liste nicht leer ist[@25].

Die andere Art Direktive, *attributive directive*, ermöglicht es, aus einfach HTML-Elementen wie *div* oder *span*, durch ergänzen eines Attributes komplexere Interfacelemente zu machen. Mittels selbst implementierter *attribute directives* kann zum einen das Aussehen und zum anderen das Verhalten bei verschiedenen *Events* (click, mouseenter, scroll,...) definiert werden[@26]. Solche Direktiven werden also häufig genutzt, wenn die Darstellung eines Elements bei Interaktion des Benutzers geändert werden soll. *Attribute Directives* und *Components* sind sich also sehr ähnlich, unterscheiden sich aber vor allem dadurch, dass zu *Components* immer ein Template gehört.

2.3.5 Pipes

Pipes dienen dazu, Daten in bestimmten Formaten darzustellen, wie es beispielsweise bei Datumsangaben oder Währungen üblich ist[@27], bzw. generell Daten vor der

Code-Ausschnitt 2.9: Beispiel für die Verwendung einer *Pipe*

```
1 [...]  
2 <span>Das heutige Datum ist: {{ today | date: 'dd.MM.yyyy' }}  
3 [...]
```

Code-Ausschnitt 2.10: Beispiele für *Data-Binding*

```
1 [...]  
2 <span>{{ user.name }}</span>  
3 <img [src] = "user.profileImgUrl" >  
4 <button (click)= "doSomething()" >  
5 <input [(ngModel)] = "user.email" >  
6 <span>Deine E-Mail: {{ user.email }}</span>  
7 [...]
```

Ausgabe zu verarbeiten. Das Beispiel 2.9 zeigt einen Template-Ausschnitt, in dem eine Pipe verwendet wird, um das in der Variablen *today* gespeicherte Datum in dem dann definierten Format auszugeben. Vereinfacht ausgedrückt ist eine Pipe eine Funktion, der ein Wert und möglicherweise Optionen übergeben werden, welchen dann einen Ausgabewert bestimmt.

2.3.6 Data Binding

Wie auch vergleichbare Frameworks und Libraries bedient sich Angular des Konzepts *Data-Binding*. Hierhinter steckt die Idee, dass Logik - also Skripte - und Repräsentation - also das DOM - miteinander verknüpft werden sollen. Eine Form des *Data-Binding* wurde mit der Interpolation in Kapitel 2.3.1.2 bereits vorgestellt. Hierbei werden Informationen aus der Komponente im zugehörigen Template dargestellt. Sollten sich die Daten der Komponente ändern, so aktualisiert sich auch automatisch das DOM. Auf vergleichbare Weise können mittels *Property Binding* auch Attribute eines DOM-Elements definiert und geändert werden. In der Umkehrrichtung können mittels *Event Binding* auch Attribute der Komponente geändert werden, wenn der Benutzer mit der Seite interagiert, beispielsweise durch Auswählen eines Listenelements oder durch Ausfüllen eines Formulars.

In Code-Ausschnitt 2.10 sind Beispiele für alle drei Varianten des *Data Binding* angegeben werden. Zeile 2 zeigt zunächst die Interpolation; hier würde der Name des Benutzers angegeben. In der nächsten Zeile wird mittels *Property Binding* der Wert des *src*-Attributs des *img*-Elementes definiert. Wird ein anderer Nutzer - und damit ein anderes *user*-Objekt - ausgewählt, ändert sich das angezeigte Profilbild, weil das *src*-Attribut sich ändert. Zeile 4 gibt ein Beispiel für *Event Binding*: Klickt man den Button an, so wird die “doSomething”-Methode der zugehörigen Komponente ausgeführt.

In der 5. Zeile ist ein Beispiel für *Two-Way Data Binding* gegeben. In dem Eingabefeld steht zunächst die im System hinterlegte E-Mail des aktuellen Benutzers - es wurde also eine Information aus der Komponente in das DOM eingebaut. Sobald man den Inhalt des Eingabefeldes ändert, wird auch die Mail-Adresse des Benutzer-Objekts der Komponente aktualisiert und damit "fließen" Daten vom DOM zur Komponente. Die Syntax verdeutlicht, dass es sich bei der "Zwei-Wege Datenbindung" also um eine Kombination von *Property Binding* und *Event Binding* handelt. Eine Veränderung des Eingabefeldes hat in diesem Beispiel außerdem zur Folge, dass sich der Inhalt des *span*-Elementes in Zeile 6 ändert. Abbildung 2.1 veranschaulicht, in welche Richtung die Daten bei den unterschiedlichen Formen des Data-Binding fließen.

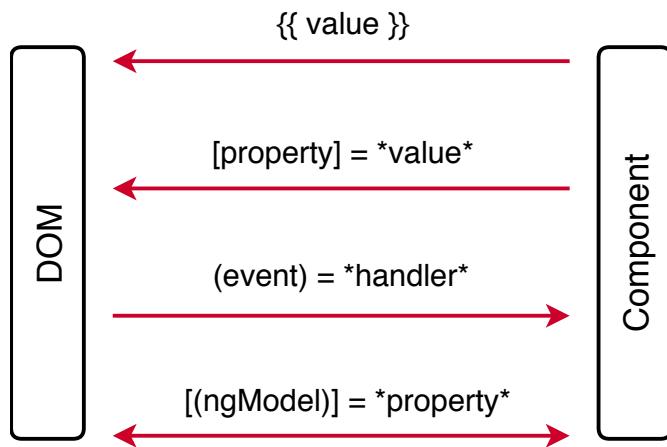


Abb. 2.1: Flussrichtung der Daten bei *Data-Binding*[@28]

2.3.7 Observables

Bei Frontend-Frameworks wie Angular kommt es häufig vor, dass Daten über eine API bezogen, verarbeitet und in die Seite eingebaut werden sollen. Das Verarbeiten der Daten kann dabei offensichtlich nur asynchron geschehen, also erst dann, wenn die Daten eintreffen und nicht, wenn die Methode zum Anfragen ausgeführt wird. Gleichermaßen gilt beispielsweise auch für das Behandeln von *Events* in Form von Benutzerinteraktionen. Hierfür werden bei Angular sogenannte *Observables* verwendet, die Teil der Drittanbieter-Library *RxJS* (Reactive Extensions for JavaScript) sind[@29].

Prinzipiell ist die asynchrone Ausführung von Code mittels *Callback*-Methoden oder *Promises* ebenso möglich. Der HTTP-Client von Angular, mit dem der Datenaustausch mit einem Server organisiert wird, verwendet aber *Observables*, welche auch sonst bei Angular gegenüber den Alternativen bevorzugt eingesetzt werden[@30]. Daher werden diese hier näher erläutert. Beim HTTP-Client ist es zwar nicht der Fall, ein *Observable* kann aber im Allgemeinen seinen Zustand - also im Prinzip seinen Wert - über die Zeit ändern. Beispiel hierfür ist die Verarbeitung von Benutzereingaben. Generell ähnelt die Anwendung von *Observables* der Nutzung der Events API; bei

Code-Ausschnitt 2.11: Beispiel für die Verwendung von *Observables*

```
1  @Injectable(/*...*/)
2  export class UserService {
3
4      private url:string = ""; /*URL*/
5      constructor(
6          private http: HttpClient
7      ) { }
8      getUsers:Observable<User []> {
9          return this.http.get<User []>(this.url);
10     }
11 }
```

Code-Ausschnitt 2.12: Beispiel für die Initialisierung eines *Subscribers*

```
1  @Component(
2      template: '<ul *ngIf="users.length>0"><li *ng-for="let user of
3         users">{{user.name}}</li></ul>'
4  [...])
5  export class UsersComponent implements OnInit {
6      users: User [] = [];
7
8      constructor(
9          private userService: UserService
10     ) { }
11     ngOnInit() {
12         this.userService.getUsers()
13             .subscribe(users => this.users = users)
14     }
15 }
```

Observables können allerdings *Events* (bzw. deren Daten) bearbeitet werden, bevor sie an den eigentlichen *Handler* weitergeleitet werden[@31].

Wie ein *Observable* genutzt werden kann, wird anhand der Datenübertragung mit dem HTTP-Client von Angular in Beispiel 2.11 demonstriert. In einem Service wird eine Methode definiert (Zeile 9), mit der eine Liste von Benutzern (also eine Array des Typs *User*) über eine API bezogen werden soll. Die *get*-Methode des HTTP-Client liefert ein *Observable* vom Typ *User[]* zurück. Beim Aufruf dieser Methode in einer Komponente (Beispiel 2.12, Zeile 12) wird dann mit der *subscribe*-Methode des zurückgegebenen *Observable*-Objekts ein *Subscriber* definiert. In diesem Fall ist dies eine Pfeilfunktion, in der die Antwort der Anfrage dem *user*-Attribut der Komponente zugewiesen wird. Dieser *Subscriber* wird logischerweise dann ausgeführt, wenn die Antwort vom Server eintrifft; der *Subscriber* beobachtet also das *Observable* bis es seinen Zustand ändert und wird dann ausgeführt. Dadurch, dass die Liste der Benutzer nach der Ausführung des *Subscribers* dann nicht mehr leer ist, wird auch die Liste des Templates der Komponente (Zeile 15) angezeigt und darin dann die einzelnen Namen aufgelistet.

2.3.8 Routing

Das sogenannte *Routing* ist eine elementarer Bestandteil bei Single-Page Applications (SPAs). Klickt man bei einer herkömmlichen Webseite auf einen Link, wird ein HTML-Dokument angefordert, das entweder statisch ist oder - was häufiger der Fall ist - auf einem Server aus Datenbankeinträgen generiert wird. Im Falle einer SPA werden nur „rohe“ Daten - in der Regel im JSON-Format oder als XML-Datei angefragt und diese dann Client-seitig verarbeitet. Abbildung 2.2 illustriert den Unterschied zwischen herkömmliche Seiten und SPAs. Da bei normalen Webseiten

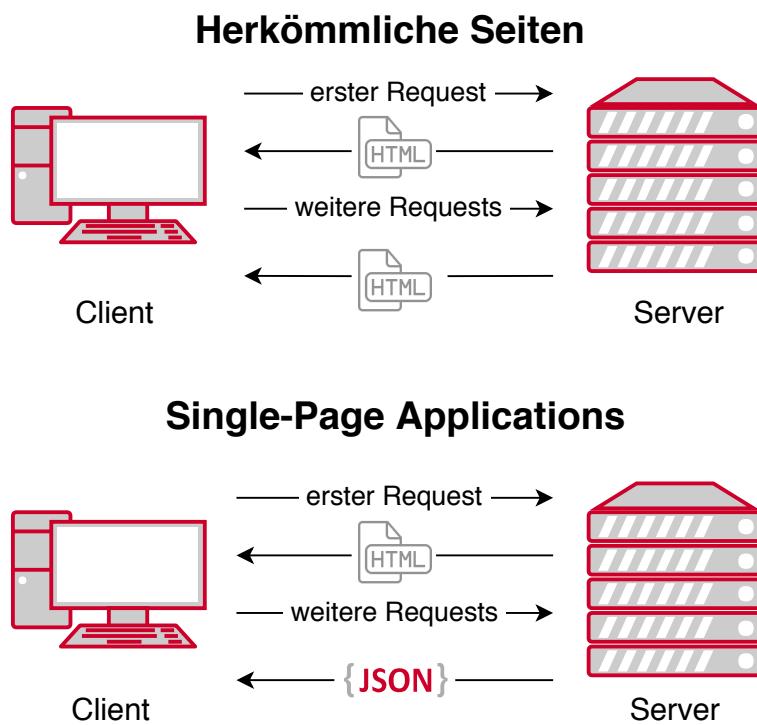


Abb. 2.2: Vergleich der Funktionsweise von herkömmlichen Seiten mit *Single-Page Applications*[@32]

für die Unterseiten oftmals der gleiche Kopf- und Fußbereich, gleiche Menüs oder Navigationsleisten sowie gleiche *Stylesheets* für diese Bereiche verwendet werden, kann hier ein *Overhead* vermieden werden, wenn beim Anklicken eines Links nur die Seitenbereiche geändert werden, die sich auch tatsächlich unterscheiden. Hierfür verhindert ein *Router* beim Anklicken eines Link den *Default* dieses Events, also das Anfragen eines neuen Dokuments und „holt“ stattdessen Daten, beispielsweise von einer API, die dann in ein Template eingesetzt werden, welches wiederum ins DOM - also die Seite - eingebaut wird.

Code-Ausschnitt 2.13 zeigt wie das Routing in Angular implementiert wird. In diesem Beispiel wurde dieses in ein eigenes Modul ausgelagert, es kann aber auch im *Root Module* implementiert werden. Das Auslagern in ein eigenes Modul ist

Code-Ausschnitt 2.13: Beispiel eines *RouterModule*

```
1 import { RouterModule, Routes } from '@angular/router';
2 import { UsersComponent } from 'users.component';
3
4 const routes: Routes = [
5   {path: 'users', component: UsersComponent},
6   {path: 'articles' loadChildren: './article.module#ArticleModule'},
7   {path: '**', component: 404Component},
8 ]
9 @NgModule({
10   imports: [
11     RouterModule.forRoot(routes)
12   ],
13   /*...*/
14 })
15 export class RouterModule {}
```

Code-Ausschnitt 2.14: Beispiel eines Templates für das *Root Component*, um Routing zu ermöglichen.[@33]

```
1 <h1>Willkommen</h1>
2 <nav>
3   <a routerLink="/users">Benutzer</a>
4   <a routerLink="/admin">Admin</a>
5 </nav>
6 <router-outlet></router-outlet>
```

z.B. dann sinnvoll, wenn sehr viele Pfade und Komponenten verwaltet werden müssen. In diesem Fall muss dann das *Routing Module* im *Root Module* importiert werden. Ab Zeile 3 wird eine Liste von *Routes* definiert, wobei einem Pfad eine Komponente zugeordnet wird, deren Template ins DOM eingebaut werden soll, wenn der Pfad aufgerufen wird. Ein Pfad wird dabei einem HTML-Link-Tag (“a”) über das *routerLink*-Attribut zugeordnet und nicht wie bei herkömmlichen Webseiten über das *href*-Attribut.

Besucht man also die URL *[FQDN]/users* durch Eingabe in die Adressleiste des Browsers oder durch Anklicken eines entsprechenden Links, dann wird das Template der Komponente “UsersComponent“ ins DOM eingesetzt. Damit das funktioniert, muss im Template des *Root Component* ein *router-outlet*-Element eingefügt werden wie in Zeile 6 des Beispiels 2.14. Das Template der aktiven Komponente wird dann als Geschwister-Element dieses Elements ergänzt. Zeile 5 aus Beispiel 2.13 zeigt zudem, wie *Lazy Loading* implementiert wird. Erst wenn man die URL *[FQDN]/articles* besucht, wird das *ArticleModule* nachgeladen, wobei dann in diesem weitere Pfade zu den Komponenten dieses Moduls analog zum *Routing Module* definiert sein können. Soll weiterhin ein Artikel direkt über eine ID geöffnet werden können, muss in diesem untergeordneten Modul ein Pfad ‘:id‘ definiert sein, sodass beim Aufruf

von [FQDN]/articles/1 der Artikel mit der ID '1' angezeigt werden kann. In Kapitel 3.4.4 wird hiervon Gebrauch gemacht.

Will man die Seite in mehrere Module unterteilen, ohne dass diese nachgeladen werden müssen, so kann man beim Importieren des *RouterModule* (Beispiel2.13, Zeile 10) die Option `{'preloadStrategy: PreloadAllModules'}` mitgeben. Dadurch werden beim ersten Seitenaufruf alle Module auf einmal geladen. Bei großen Modulen kann das allerdings entsprechend lang dauern.

2.3.9 Zusammenfassung der Architektur

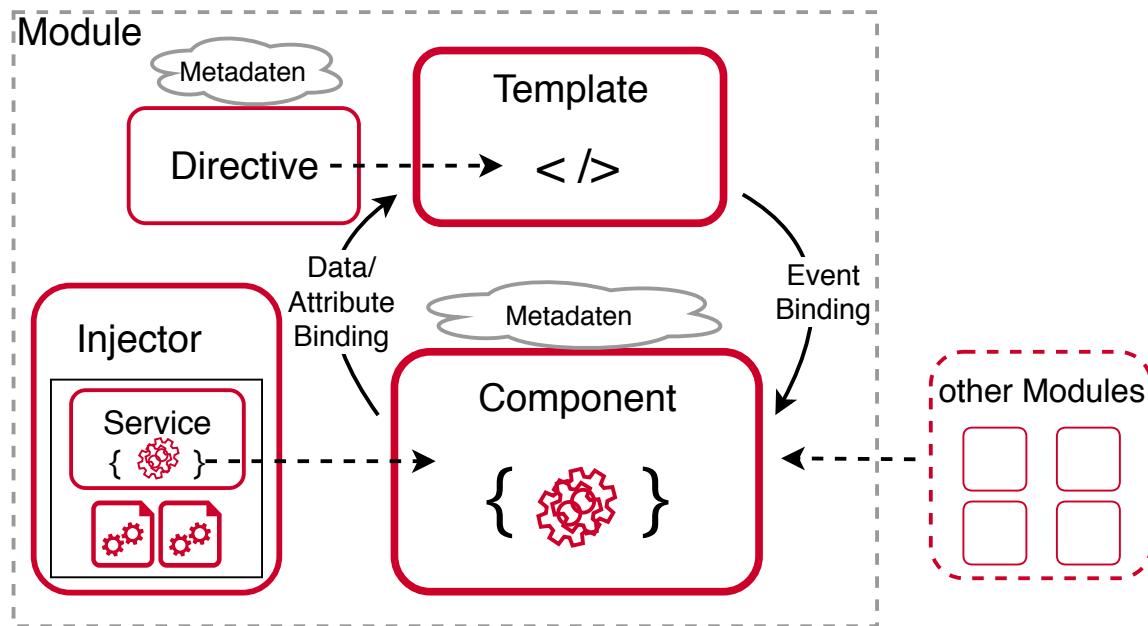


Abb. 2.3: Übersicht über die Architektur des Angular-Frameworks[@34]

Aus den bisher vorgestellten Bausteinen von Angular lässt sich eine grobe Architektur ableiten, die in Abbildung 2.3 dargestellt ist. Komponenten sind die Kernbausteine und in Modulen zusammengefasst, wobei die Metadaten der Komponenten Informationen über das zugehörige Template, CSS-Regeln, Services etc. enthält. Über Services, die in das Modul oder die Komponenten "injiziert" werden, können letztere Daten von einem Server oder anderen Komponenten beziehen. Diese Daten können mittels *Data Binding* oder *Attribute Binding* in dem zur Komponente gehörenden Template unter Zuhilfenahme von Direktiven eingebunden werden. Umgekehrt kann mit *Event Bindings* auf Benutzerinteraktionen reagiert und der Zustand der Komponente geändert werden. Außerdem können andere Module importiert und somit deren Funktionalitäten genutzt werden. Darüber hinaus kann mittels *Routing* das Navigieren zwischen Seiten simuliert werden, indem Templates verschiedener Komponenten gegeneinander ausgetauscht werden.

2.3.10 Das Befehlszeilen-Interface (CLI)

Mit der Version 2 von Angular - also der grundlegenden Überarbeitung - wurde für dieses Framework das Paket `@angular/cli` veröffentlicht, welches ein Befehlszeilen-Interface enthält. Dieses CLI soll die Entwicklung mit Angular vereinfachen und enthält im wesentlichen Befehle für:

- das Erzeugen einer neuen Anwendung
- das Generieren von *ngModules*, *Components*, *Services* etc.
- das Hinzufügen und Installieren zusätzlicher Pakete
- das Testen und “Zusammenbauen“ der Anwendung

Dieses Befehlszeilen-Interface wird über den Paketmanager *npm* bezogen, über den bei Erzeugung eines neuen Projektes mit dem Befehl *ng new* zusätzliche Pakete installiert werden, die für eine rudimentäre, auf Angular basierende, Webanwendung notwendig sind. Außerdem können mit *npm* dem Projekt im Verlauf der Entwicklung weitere Pakete und Software-Bibliotheken hinzugefügt werden.

Der Befehl *ng generate* (oder kurz *ng g*) bietet Optionen zum Generieren generischer Klassen und Interfaces, aber auch zum Erzeugen Angular-spezifischer Klassen wie *Components*, *ngModules*, *Services* etc. Bei der Generierung einer Angular-spezifischen Klasse wird neben der eigentlichen Datei, in der diese Klasse implementiert wird, eine weitere Datei erstellt, in der sich ein Gerüst für *Unit-Tests* der Klasse befindet. Im Falle einer Komponente werden diese Dateien in einem Ordner, der den gleichen Namen wie die Komponente hat, erzeugt und in diesem außerdem jeweils eine Datei für das zugehörige HTML-Template (2.3.1.2) sowie für die CSS-Regeln (2.3.1.3) erstellt - insgesamt also 4 Dateien. Allerdings kann über eine *Flag* erreicht werden, dass statt einer eigenen Datei für das Template ein Inline-Template in der *.component.ts*-Datei erstellt wird. Außerdem wird die Komponente dem *declarations*-Array des - auf die Ordnerhierarchie bezogen - nächstgelegenen Moduls hinzugefügt (vgl. 2.3.2). Dies gilt auch für das Erzeugen von Direktiven, Services und Pipes.

Am Anfang von Kapitel 2.3 wurde der Befehl *ng add* bereits oberflächlich vorgestellt. Im Unterschied zum Ergänzen neuer Pakete via *npm* werden bei Benutzung dieses Befehls die Pakete dem Projekt nicht einfach nur hinzugefügt, sondern sie können direkt über ein Installationsskript in das Projekt eingebunden werden.

Weiterhin wichtig sind die Befehle *ng test*, mit dem die Unit-Tests gestartet werden können, und *ng serve*, durch welchen ein Entwicklungsserver gestartet wird, mit

dem standardmäßig die Webanwendung neu geladen wird, wenn Dateien bearbeitet wurden. Hierfür wird die *NodeJS*-Umgebung genutzt.

Zuletzt gibt es noch den Befehl *ng build*, mit dem die Webanwendung für die Auslieferung auf einen Server vorbereitet wird: Dafür wird beispielsweise TypeScript in JavaScript übersetzt oder Dateien werden komprimiert.

2.4 TypeScript

TypeScript ist eine quelloffene, von Microsoft entwickelte Programmiersprache, welche ein *Superset* von JavaScript darstellt. Hauptmerkmal von TypeScript ist, dass es im Gegensatz zu JavaScript die Möglichkeit bietet, statische Typen zu nutzen. Als primitive Typen gibt es *number*, die immer als Fließkommazahl gespeichert werden und auch für binäre, oktale und hexadezimale Literale genutzt werden können, und *string*. Ebenso gibt es Arrays ebendieser[@35]. Damit können dann natürlich auch typisierte Klassen entwickelt werden. Außerdem ergänzt es JavaScript um weitere Funktionalitäten wie *Namespaces*, also Namensräume oder *Interfaces*. Mit einem Compiler, der mit dem Paketverwaltungssystem *npm* installiert werden kann, muss TypeScript vor der Auslieferung in JavaScript übersetzt werden[@36]. TypeScript hilft also dabei, Code zu entwickeln bei dem die einzelnen Teile wirklich zueinander passen und nicht beispielsweise ein Methodenaufruf mit falschen Parametern erfolgt. Somit können Laufzeitfehler minimiert bzw. weitestgehend vermieden werden.

Da JavaScript-Code prinzipiell auch TypeScript-Code ist, können JavaScript-Bibliotheken auch in TypeScript verwendet werden. Diese Bibliotheken sind dann allerdings zunächst “untypisiert”. Über das Projekt *Definitely Typed* (<https://definitelytyped.org/>) können für sehr viele JavaScript-Bibliotheken sogenannte *Type Definitions* bezogen und mittels *npm* installiert werden. Somit kann dann die Bibliothek “typisiert” genutzt werden. Diese muss dabei natürlich trotzdem im Projekt vorhanden sein, sie wird fortan lediglich indirekt genutzt.

2.5 Sonstiges

OpenLayers Bei OpenLayers handelt es sich um eine Library, mit der Karten in Webseiten eingebunden werden können. Dabei kann Kartenmaterial verschiedenster Quellen genutzt werden; am häufigsten wird OpenStreetMap verwendet, aber auch andere Quellen sind möglich. Die API dieser Library bietet zahlreiche Möglichkeiten, um das verwendete Kartenmaterial zu manipulieren oder zusätzliche Informationen auf der Karte einzublenden. Diese Library ist ein unter der *2-Klausel-BSD-Lizenz* veröffentlichtes Open Source-Projekt[@37].

angular-oauth2-oidc Das Paket *angular-oauth2-oidc* ermöglicht die Authentifizierung eines Benutzers über die OpenID Connect-Authentifizierungsschicht. Nachdem in einer JSON-Datei konfiguriert wurde, welcher *Identity-Provider* verwendet werden soll (sprich: zu welcher Seite er weitergeleitet wird, um sich dort zu authentifizieren), kann eingestellt werden, dass der Nutzer direkt beim initialen Seitenaufruf zum Login-Portal weitergeleitet werden soll, oder ob dies erst geschieht, wenn auf einen dedizierten Login-Button geklickt wird. Hat sich der Benutzer erfolgreich eingeloggt, kann ein vom *Identity Provider* bereitgestelltes Token zum Header von Anfragen, die einer Authentifizierung bedürfen, hinzugefügt werden. Dieses Paket wurde unter der MIT-Lizenz veröffentlicht und ist damit Open Source-Software[@38].

ngx-barcode Dieses Paket ergänzt Angular um eine Komponente, mit der Barcodes in verschiedenen Formaten dargestellt werden können. Auch dieses Paket wurde unter der MIT-Lizenz veröffentlicht[@39].

Angular Material (@angular/material) Das vom Angular-Team entwickelte Paket *@angular/material* (<https://material.angular.io>) bietet eine ganze Reihe an Komponenten für die Gestaltung des Aufbaus einer Webseite. Zu diesen Komponenten zählen beispielsweise Listen, Formularfelder oder Menüs. Dabei ist das Design aller Komponenten dem von Google entwickelten *Material Design* nachempfunden. Dieses zeichnet sich durch flache Elemente und kontrastreiche Farben aus. Mit diesem Paket ist es möglich, eine mit dem Angular-Framework entwickelte Webanwendung anspruchsvoll zu gestalten, ohne selbst (viel) CSS schreiben zu müssen. Dieses Paket ist dabei abhängig von den Paketen *@angular/animations* (für Animationen) und *@angular/cdk*, dem *Component Development Kit*. Letzteres hilft beispielsweise beim Entwickeln von Overlays oder bei der barrierefreien Gestaltung.

Angular Progressive Web App (@angular/pwa) Mit diesem Paket, kann eine Angular-App ganz einfach in eine *Progressive Web App* verwandelt werden, indem dem Projekt eine Manifest-Datei, ein *Service Worker*-Skript und eine Konfigurationsdatei hinzugefügt werden. Damit nimmt einem dieses Paket viel Arbeit ab, weil die Implementierung des Service Workers sowie dessen Registrierung und Installation im Browser durch dieses Paket übernommen werden. Damit verbleibt nur, in der Konfigurationsdatei die Ressourcen, welche im Cache gespeichert werden sollen, zu definieren.

Material Design Icons Bei *Material Design Icons* (<https://materialdesignicons.com/>) handelt es sich um ein Projekt, über das unzählige Icons im Stile des Material Designs in verschiedenen Formaten (als .png- oder .svg-Datei oder als Font) bereitgestellt werden. Bei diesen Icons handelt es sich zum Teil um von Google selbst entworfene

(und in dessen Projekten genutzte) Bilder, zum Großteil aber um Graphiken, die von Dritten eingereicht wurden; man kann also auch selbst welche vorschlagen.

Icons8 Im Gegensatz zu den Icons von *Material Design Icons* sind die bei Icons8 (<https://icons8.de>) erhältlichen Graphiken nicht (nur) monochrom, sondern mehrfarbig verfügbar. Diese sind zwar prinzipiell auch kostenlos erhältlich, dann allerdings nicht in allen Auflösungen und nur als .png-Datei. Vollumfänglicher Zugriff auf diese kann mit dem Erwerb einer Lizenz erhalten werden, außerdem erhält man so Zugriff auf weitere Funktionen.

Bilder Das für diese Webanwendung verwendete Hintergrundbild stammt von <https://www.pexels.com>.

Programmierung der Webanwendung

„Talk is cheap. Show me the code.“

— Linus Torvalds

(Finnisch-/US-amerikanischer Informatiker,
Schöpfer von Linux und Git)

In diesem Kapitel wird die Entwicklung der Web-App beschrieben. Dabei wird zunächst erläutert, welche Inhalte diese umfassen soll. Danach werden besondere Hindernisse angeführt, die zu überwinden sind. Anschließend folgt die Beschreibung der eigentlichen Programmierung, wobei erst erklärt wird, welche Vorbereitungen getroffen werden und welche grundlegenden Funktionalitäten neben den eigentlichen Inhalten implementiert werden, bevor dann die Vorstellung der verschiedenen Module schrittweise folgt.

3.1 Inhalte

Nachrichten Im ersten Modul sollen Nachrichten und Neuigkeiten der Universität und des Zentrums für Datenverarbeitung (ZDV) angezeigt werden. Dabei handelt es sich bei den Quellen um sogenannte RSS-Feeds

Campus-Karte Das zweite Modul umfasst eine Karte des Universitätscampus, auf der - nach Auswahl aus einer Liste - verschiedene Gebäude der Universität angezeigt werden sollen.

Busfahrpläne Zu den Bus- und Straßenbahnhaltestellen rund um die Universität werden im dritten Modul die Abfahrtszeiten der demnächst eintreffenden Busse und Straßenbahnen angeben. Die Fahrpläne werden hier über die API des Rhein-Main Verkehrsverbundes bezogen.

Personensuche Das vierte Modul bietet eine Suchmaske um nach Angestellten der Universität zu suchen, die im Informationssystem der Universität Mainz (UnivIS) hinterlegt sind.

Veranstaltungen In diesem Modul werden die im UnivIS angegebenen Veranstaltung in chronologischer Reihenfolge aufgelistet.

Speisepläne Nach Datum und Ausgabestelle sortierte Speisepläne zu den Menschen auf dem Universitätsgelände können in diesem Modul abgefragt werden. Die Daten hierfür stammen vom Studierendenwerk der Universität.

Wetter Das siebte Modul zeigt die Wetterdaten der Messstation auf dem Campus an.

Öffnungszeiten Hier sollen Öffnungszeiten verschiedener Einrichtungen und Büros eingesehen werden können - zunächst die der Bibliotheken sowie von Büros des ZDV. Auch diese Daten stammen vom UnivIS.

Authentifizierung Dieses nicht sichtbare Modul soll eine Authentifizierung auf Basis von *OpenID Connect* durchführen.

Bibliotheksausweis Nach erfolgreicher Authentifizierung wird in diesem Modul ein digitaler Bibliotheksausweis angezeigt.

PC-Pools Hier soll eine Liste der PC-Pools auf dem Campus - inklusive deren Ausstattung - angegeben werden. Außerdem sollen die darin stattfindenden Kurse aufgelistet werden.

3.2 Besondere Schwierigkeiten

3.2.1 Same Origin Policy (SOP) - Cross-Origin Resource Sharing (CORS)

Bei Webanwendungen wie der hier entwickelten kommt es vor, dass Daten von anderen Quellen bezogen werden sollen, als von der Domain, von der die Webseite eigentlich stammt. Das wird allerdings durch die *Same-Origin Policy (SOP)* - vom Browser und nicht vom Server - blockiert. Hierbei handelt es sich um einen Sicherheitsmechanismus, der verhindern soll, dass clientseitige Skripte Ressourcen von anderen Domains anfragen und somit an Daten gelangen, auf die diese Skripte - bzw. deren Quelle - eigentlich keinen Zugriff haben dürften. Abbildung 3.1 veranschaulicht dies: wird eine Seite - also ein HTML-Dokument - von einer Adresse geladen - in diesem Beispiel von *domain.com*, dann können mittels nachfolgender XMLHttpRequests (XHR) nur Ressourcen angefragt werden, die in Protokoll, Host und Port mit der ursprünglichen Anfrage übereinstimmen. Der Zugriff auf Ressourcen von *otherDomain.com* bleibt hingegen verwehrt - genauso werden aber auch An-

fragen an Subdomains oder an andere Ports von `domain.com`, die nicht dem der ursprünglichen Anfrage entsprechen blockiert. Insbesondere bei modernen Websei-

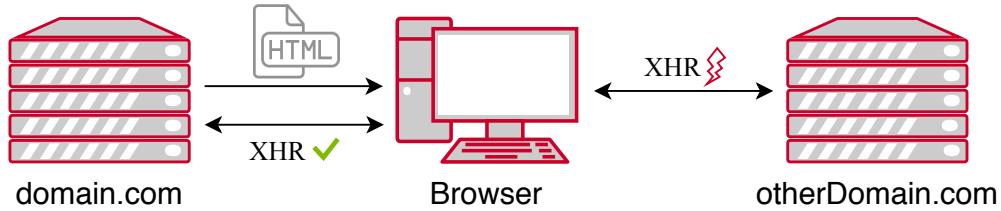


Abb. 3.1: Veranschaulichung der *Same-Origin Policy*

ten ist aber genau das gewünscht, damit Webseiten von einer Domain Zugriff auf APIs anderer Domains bekommen können. Hierfür wurde das sogenannte Cross-Origin Ressource Sharing (CORS) entwickelt. In Abbildung 3.2 ist (vereinfacht) schematisch dargestellt wie mit dieser Technik Ressourcen anderer Domains angefragt werden können. Dabei wird zunächst ein *Preflight Request* an einen Server geschickt, um herauszufinden, ob der Zugriff auf dessen Ressourcen möglich ist und welche Optionen (GET, POST, DELETE, etc.) unterstützt werden. Je nachdem, wie der Server konfiguriert ist, schickt er mit einer *Preflight Response* einen HTTP-Header `Access-Control-Allow-Origin`, über den einer, mehreren oder allen andern Domains (bzw. deren Skripten) erlaubt wird, auf `otherDomain.com` zuzugreifen. Außerdem kann über `Access-Control-Allow-Methods` eingeschränkt werden, welche Operationen möglich sind. Entsprechend dieser Informationen verwirft der Browser die Anfrage oder kann sie ausführen, um dann die eigentliche Antwort zu erhalten. Dabei muss diese Technik offensichtlich vom Browser implementiert werden[40].

Bei ein paar der Ressourcen, die für diese Arbeit verwendet werden sollten, war aber genau diese serverseitige Konfiguration nicht gegeben. Um diese dennoch nutzen zu können, wurde ein externer Proxy-Server (<https://cors-anywhere.herokuapp.com>) genutzt, welcher genau für solche Zwecke eingerichtet wurde. Um an Daten zu gelangen, deren Quelle CORS nicht unterstützt, werden die Anfragen also an diesen Proxy-Server geschickt, der diese an den eigentlichen Server weiterleitet und die dann erhaltene Antwort wieder zum Nutzer zurückschickt. Das hat - vor allem weil die Kontrolle über diesen Proxy-Server fehlt - ganz offensichtlich zwei entscheidende Nachteile: Zum einen sind Zugriffe auf Ressourcen, die einer Authentifizierung bedürfen, über einen solchen Proxy nicht ratsam (falls überhaupt möglich) und zum anderen ist man von dessen Verfügbarkeit abhängig, weil Teile der Webanwendung nicht nutzbar sind, wenn der Proxy-Server mal wieder überlastet ist. Ebenfalls ungünstig ist die Tatsache, dass es durch diesen Umweg länger dauert, bis die Daten beim Nutzer ankommen.

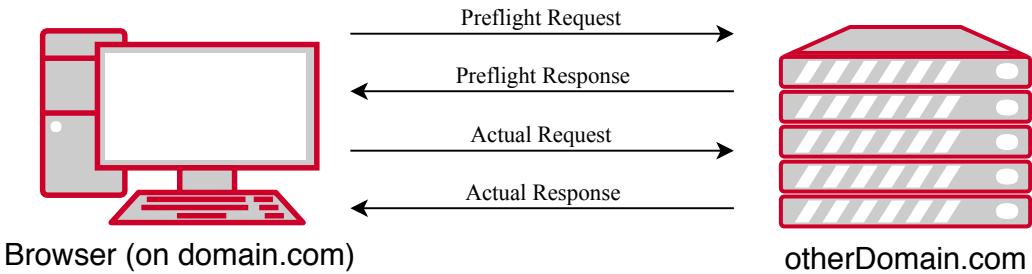


Abb. 3.2: Veranschaulichung von *Cross-Origin Ressource Sharing (CORS)*[@41]

3.2.2 JavaScript und XML

Viele der für dieses Projekt genutzten Daten lagen im XML-Format vor. Das ist insofern ungünstig, weil es mit JavaScript (bzw. TypeScript) wesentlich einfacher ist, JSON-Daten zu verarbeiten: Schon der Name verrät, dass der Aufbau solcher Daten der JavaScript-Objekt-Syntax entspricht. Mit Hilfe eines Parsers, der Teil von JavaScript ist, können mit solchen Daten automatisch JavaScript-Objekte erzeugt werden, die dann ganz einfach weiterverarbeitet werden können - in der Regel um das DOM um weitere Informationen zu ergänzen. Darauf bauen im Prinzip alle Frontend-Frameworks auf; bei Angular ist beispielsweise gar nicht erforderlich, den Parser selbst aufzurufen. Stattdessen definiert man vor dem Abschicken eines Requests über den HTTP-Client, Daten welchen Typs man erwartet und das Parsen geschieht dann automatisch (vgl. Zeile 10 aus Beispiel 2.11).

Auch wenn es zwar prinzipiell möglich ist, mittels JavaScript XML-Elemente in das DOM einzubinden, lässt sich bei den meisten Frontend-Frameworks wesentlich besser mit JavaScript-Objekten arbeiten. Also mussten alle Daten, die nicht im JSON-Format verfügbar sind, manuell aus der XML-Struktur extrapoliert und daraus JavaScript-Objekte generiert werden. Es wäre prinzipiell möglich, einen generischen Parser zu schreiben, der aus einem XML-String stur ein äquivalentes JavaScript-Objekt erzeugt (es gibt auch Libraries Dritter, die genau das tun), da aber viele Daten in unvorteilhaften Strukturen vorliegen, wurde das Parsen für jede Datenquelle individuell implementiert. Damit konnten Objekte modelliert werden, die zur Weiterverarbeitung besser geeignet sind. Anhand des Nachrichtenmoduls wird diese Vorgehensweise einmalig erklärt und für aller weiteren Fälle ausgelassen. Dort wird dann nur noch erklärt, welche Klassen oder Interfaces zur Strukturierung der Daten modelliert wurden.

3.3 Die eigentliche Programmierung

3.3.1 Vorbereitungen

Als erstes werden die *NodeJS*-Serverumgebung sowie das Paketverwaltungssystem *npm* installiert. Mit letzterem wird dann das Befehlszeileninterface von Angular (siehe Kapitel refsec:technologies:angular:cli) installiert. Mit diesem wird als nächstes über den Befehl *ng new Studierendenportal* ein neues Projekt mit dem Titel “Studierendenportal” erstellt. Dabei werden automatisch weitere Pakete installiert, welche für die Entwicklung notwendig sind - beispielsweise TypeScript (vgl. Kapitel 2.4). Ist dieser Vorgang abgeschlossen befindet sich in dem Ordner eine im Prinzip funktionsfähige Angular-Web-App bestehend aus einem *AppModule* und einem *App-Component*. Bezogen auf den Ordner, in dem das Projekt erstellt wurde, befinden sich diese im Verzeichnis */src/app/*, wo auch alle weiteren Dateien durch das CLI erstellt werden. Im Verzeichnis */src/* befinden sich zum die *Index.html*-Datei, ein globales Stylesheet, ein Icon für Lesezeichen sowie ein Verzeichnis */assets/* für andere Bilder. Sowohl im Wurzelverzeichnis als auch in */src/* werden außerdem noch verschiedene Konfigurationsdateien erstellt. Die hauptsächliche Programmierung findet aber in */src/app/* statt. Im Folgenden beziehen sich daher alle Angaben - falls nicht ausdrücklich anders angegeben - auf diesen Ordner als Wurzel. Der gesamte Quellcode kann auf der beigelegten CD eingesehen werden. Außerdem kann eine funktionierende Version auf <http://portal.web-preview.jgulab.net/> (nur erreichbar innerhalb des Universitätsnetzwerks) abgerufen werden.

3.3.2 Das Grundgerüst der Anwendung

Wie bereits in Kapitel 2.3.2 beschrieben, stellt das *AppModule* - implementiert in der Datei *app.module.ts* - den Kern der ganzen Anwendung dar. Beispiel 3.1 stellt einen Ausschnitt aus *app.module.ts* dar. Da das *Routing* in ein eigenes Modul ausgelagert wurde, werden dort auch alle Module, welche die eigentlichen Inhalte umfassen, importiert. Somit geschieht der Import dieser Module ins *Root Module* indirekt über den Import des *Router Modules*. Hierbei bedeutet “importieren“, dass das bzw. die Modul(e) zum *imports*-Array hinzugefügt werden (vgl. ab Zeile 8, 3.1). Im *AppModule* müssen also nur noch die Module importiert werden, die für zusätzliche Funktionen zuständig sind. Dazu zählen beispielsweise die Module für die Authentifizierung, für den Service Worker, aber auch grundlegende Module für die Interaktion mit dem Browser oder dem DOM.

Da an verschiedenen Stellen SVG-Icons genutzt werden sollen und es zu umständlich und zu unübersichtlich wäre, deren Quellcode an den entsprechenden Stellen direkt ins Template einzusetzen, müssen diese ebenfalls hier eingebunden werden, damit in den Templates ein Verweise auf ein solches Icon gesetzt werden kann. Während des *Build*-Prozesses wird die eigentliche Graphik in das Template eingesetzt. Bei

Code-Ausschnitt 3.1: Ausschnitt aus app.module.ts

```
1 // Importe anderer Module, Komponenten oder Services
2 /*...*/
3 @NgModule({
4   declarations: [
5     AppComponent,
6     IndexComponent
7   ],
8   imports: [
9     /*...*/
10    AppRoutingModule,
11    CoreModule,
12    MaterialModule,
13    OAuthModule.forRoot(),
14    ServiceWorkerModule.register('ngsw-worker.js', {enabled:
15      environment.production})
16  ],
17  providers: [],
18  bootstrap: [AppComponent]
19 })
20 export class AppModule {
21   constructor(iconRegistry: MatIconRegistry, sanitizer:
22     DomSanitizer) {
23     iconRegistry.addSvgIcon('home',
24       sanitizer.bypassSecurityTrustResourceUrl('assets/icons/
25         home.svg'));
26   }
27 }
```

der Verwaltung der Icons wurde sich der *Icon Registry* des Material Design-Pakets bedient, weil auch die Einbindung der Icons über dieses Paket umgesetzt wird. Ab Zeile 20 aus wird gezeigt, wie das geht, wobei auf gleiche Weise mehrere SVG-Icons registriert werden können; der Aufruf von addScgIcon kann konateniert werden. Alternativ wäre es auch möglich, alle verwendeten SVG-Icons einer Datei quasi als *Spritesheet* zusammenzufassen und diese dann zu inkludieren.

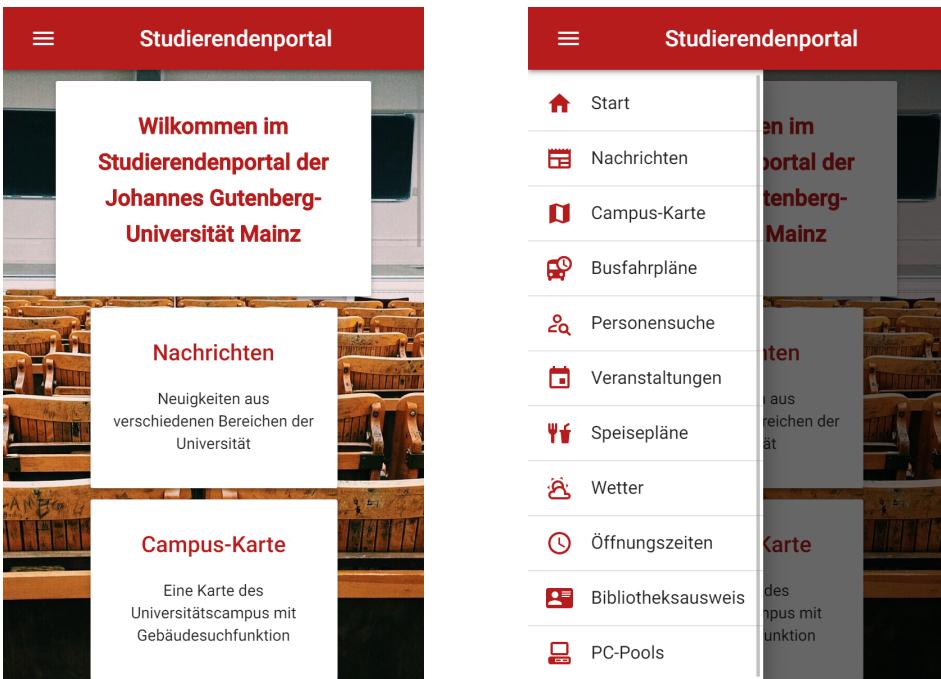
Weiterhin wichtig ist die Deklaration der Komponenten *AppComponent* und *IndexComponent* (ab Zeile 4). Das Template ersterer stellt das Grundgerüst der ganzen Web-App dar, befindet sich in der Datei *app.component.html* und wird in Ausschnitt 3.2 wiedergegeben. Dieses Template zeigt, wie mit *Angular Material* eine Seite mit einer Toolbar und einer “responsiven”, seitlichen Navigation erstellt wird und setzt sich zunächst aus drei Bausteinen zusammen: einer Kopfleiste, in der der Titel der aktuell angezeigten Komponenten stehen soll (Zeile 3), einem Menü als Seitenbereich, in dem Links zu allen Bereichen aufgelistet sind (ab Zeile 9) und einem Container für den eigentlichen Inhalt der Seite (ab Zeile 20). Das einzige Kindelement des Inhaltscontainers ist dabei das in Kapitel 2.3.8 beschriebene *router-outlet*-Element, welches für die Nutzung des Routers notwendig ist. Außerdem wird mittels Event Binding erreicht, dass das Navigationsmenü geschlossen wird, wenn ein Link aus diesem Menü angeklickt wird (Zeile 10).

Code-Ausschnitt 3.2: Ausschnitt aus app.component.html (Quelle: <http://material.angular.io/components/sidenav/examples>)

```
1 <div class="page-container">
2   <!-- toolbar with heading -->
3   <mat-toolbar color="primary" class="mat-elevation-z3">
4     <!-- Kopfbereich -->
5     <button mat-icon-button (click)="snav.toggle()" aria-label="Menü öffnen">
6       <h1 id="page-title">{{title}}</h1>
7     </mat-toolbar>
8     <mat-sidenav-container class="mat-sidenav-container" >
9       <!-- navigation menu -->
10      <mat-sidenav #snav mode="over" (click)="snav.toggle()" role="navigation">
11        <mat-nav-list>
12          <a mat-list-item routerLink="/" >
13            <mat-icon svgIcon="home" color="primary" class="nav-icon"></mat-icon>
14            Start
15          </a>
16          <!-- ... -->
17        </mat-nav-list>
18      </mat-sidenav>
19      <!-- wrapper for page content -->
20      <mat-sidenav-content role="main">
21        <router-outlet></router-outlet>
22        <!-- page content inserted from router -->
23      </mat-sidenav-content>
24    </mat-sidenav-container>
25  </div>
```

Besonders auffällig sind die vielen Tags, bei denen es sich offensichtlich nicht um Standard-HTML-Elemente handelt. Stattdessen sind es alles Komponenten aus dem Material-Design-Paket. Das Element `mat-icon` ist eine solche und weil zuvor der *Icon Registry* ein SVG-Icon mit dem Namen “home“ hinzugefügt wurde, kann wie in Zeile 12 angegeben eben jene Graphik in ein Template eingebunden werden. Bei Tags und Attributes, die mit dem Präfix “mat“ beginnen, handelt es sich um Komponenten bzw. Direktiven aus dem *Angular Material*-Paket und in den folgenden Kapiteln werden noch mehr davon vorgestellt.

Die Komponente `IndexComponent` beinhaltet die Startseite, welche im Grunde die gleichen Links wie das Navigationsmenü beinhaltet, nur sind sie als Karten dargestellt, in welchen kurz der Inhalt des Moduls beschrieben ist. Abbildung 3.3a zeigt, wie das Grundgerüst der Web-App auf einem Mobilgerät aussieht, wenn diese Startseite aufgerufen wird, und Abbildung 3.3b zeigt das geöffnete Menü, welches ein- und ausgeblendet wird, wenn man auf den Button oben links klickt.



(a) normale Ansicht

(b) mit geöffnetem Menü

Abb. 3.3: Grundgerüst der Web-App

3.3.3 Spezielle Module und Services

3.3.3.1 RoutingModule

Wie angedeutet wurde das Routing der Übersicht halber in eigenes Modul ausgelagert. Dabei wurde das *Lazy Loading* außer Kraft gesetzt (vgl. Kapitel 2.3.8), sodass alle Module auf einmal geladen werden. Es wäre eigentlich gar nicht zwingend notwendig gewesen, alle Inhalte in eigene Module zu unterteilen, der Übersichtlichkeit halber und weil bei zukünftigen Erweiterungen den Modulen eventuell weitere Komponenten hinzugefügt werden sollen, wurde es trotzdem getan.

3.3.3.2 CoreModule

Über dieses Modul werden alle in diesem Projekt implementierten Services eingebunden. Wie beim *Routing* wäre es hier ebenfalls möglich gewesen, die Services im *AppModule* direkt einzubinden und auch hier wurde dies vor allem zu Gunsten einer besseren Übersicht nicht getan. Zu finden ist dieses Modul in der Datei `/core/core.module.ts`. Es sei hier noch erwähnt, dass das Einbinden von Services auf globaler Ebene - was hier durch dieses Modul geschieht - bei Nutzung von *Lazy Loading* wichtig ist, da auf Komponenten-Ebene eingebundene Services in

nachgeladenen Modulen nicht genutzt werden können. Wird ein Service außerhalb eines Moduls nicht benötigt ist das aber unwichtig[@42].

3.3.3.3 MaterialModule

Es wurde bereits mehrfach erwähnt, dass das Material Design-Paket von Angular verwendet wird, um die Webseite zu gestalten. Dabei befindet sich (fast) jede Komponente dieses Pakets in einem eigenen Modul. Die meisten Komponenten dieser Web-App verwenden mehrere dieser Design-Komponenten und damit müssten alle Module der Design-Komponenten in den Modulen der eigentlichen Komponenten importiert werden. Daher wurde hier ebenfalls für eine bessere Übersicht ein weiteres Modul eingeführt, das alle benötigten Module des Design-Pakets importiert und damit bündelt. Folglich muss in den Modulen, deren zugehörige Komponenten diese Design-Komponenten verwenden, nur dieses eine Modul importiert werden. Dieses Modul befindet sich in `/material/material.module.ts`.

3.3.3.4 ToolbarService

Bei der Erläuterung des Templates der Wurzelkomponente in Kapitel 3.3.2 wurde erklärt, dass sich der Titel im Kopfbereich der Seite (also in der Toolbar, Ausschnitt 3.2, Zeile 5) ändern soll, wenn zwischen den einzelnen Seiten (d.h.Komponenten) gewechselt wird. Um das umzusetzen wurde der `ToolbarService (toolbar.service.ts)` implementiert. Dieser ist in Ausschnitt 3.3 wiedergegeben. In jeder Komponente wird in der jeweiligen `onInit`-Methode die Funktion `setToolbarTitle` des ToolbarService aufgerufen und der Titel der Komponente übergeben. Dieser Titel wird gleichzeitig an den TitleService, der von Angular selbst bereitgestellt wird, übergeben und somit der Seitentitel, also Beschriftung des Browser-Fensters, geändert. Im AppComponent wird ein Subscriber für den `EventEmitter` des ToolbarService implementiert, in dem der übergebene Titel als Titel der Toolbar gespeichert wird. Ausschnitt 3.4 zeigt diesen Subscriber. Weil sich durch den Subscriber die Variable `title` der AppComponent ändert, ändert sich mittels *Data Binding* auch der Titel der Toolbar (vgl. Ausschnitt 3.2, Zeile 6).

3.4 Die eigentlichen Inhalte

In dem meisten Fällen ist das Vorgehen immer das gleiche: Bei Initialisierung einer Komponente werden über einen Service Daten angefragt, die anschließend unter Umständen vom Service geparsst werden müssen, bevor sie dann in das Template der Komponente eingesetzt werden. Lediglich beim Karten-Modul und bei der

Code-Ausschnitt 3.3: Ausschnitt aus ToolBarService

```
1 @Output() fire: EventEmitter<any> = new EventEmitter<any>();
2 constructor() { }
3
4 setToolbarTitle(title: string) {
5   this.fire.emit(title);
6 }
7
8 getEmittedValue() { return this.fire; }
```

Code-Ausschnitt 3.4: "Abonnieren" des ToolbarService im AppComponent

```
1 export class AppComponent implements OnInit {
2   title = '';
3   subscription: EventListener;
4   /*...*/
5   ngOnInit() {
6     this.subscription = this.toolbarService.getEmittedValue()
7       .subscribe(title => {
8         this.title = title;
9       });
10  }
11  /*...*/
12 }
```

Personensuche ist das Vorgehen (leicht) abweichend. Zur Verarbeitung der Daten wurden mehrere Klassen und Interfaces definiert, die sich alle im Verzeichnis `models` befinden.

3.4.1 Nachrichten

Der Service dieses Moduls¹ enthält vier Funktionen, von denen die komplexeste, welche für das Parsen zuständig ist, in Ausschnitt 3.5 auszugsweise wiedergegeben wird. In dieser wird zunächst ein Feed-Objekt² erzeugt, welches einen Namen hat und eine Liste von FeedPost-Objekten³ enthält. Diese wiederum setzen sich zusammen aus einem Titel, einer Beschreibung, einem Link zum eigentlichen Artikel, einem Veröffentlichungsdatum sowie einer Reihe von Kategorien, zu denen der jeweilige Post gehört.

Bevor die eigentlichen Posts des RSS-Feeds geparsst werden, muss aus dem übergebenen XML-String ein DOM-Objekt erzeugt werden und dann wie in Zeile 4 die Liste der Posts (ausgezeichnet durch einen `item`-Tag) abgefragt werden. Anschließend wird über diese Liste iteriert, alle Eigenschaften des jeweiligen Posts abgefragt und diese im erzeugten `feedPost`-Objekt gespeichert. Wie das Abfragen der Attribute

¹/news-feed/news-feed.service.ts

²/models/feed.ts

³/models/feedPost.ts

Code-Ausschnitt 3.5: Ausschnitt aus dem NewsFeedService

```
1  parseFeedFromXmlToJson(feedAsString: string, feedName: string) {
2    const feedAsJson: Feed = new Feed(feedName);
3    const feedAsXml = new DOMParser().parseFromString(feedAsString,
4                                                    'application/xml');
5    const feedItems = (feedAsXml.getElementsByTagName('item'));
6    for (let index = 0; index < feedItems.length; index++) {
7      const feedItem: FeedPost = new FeedPost();
8      const itemAsXml = feedItems.item(index);
9      const categories = itemAsXml.getElementsByTagName('category');
10     feedItem.title = itemAsXml.getElementsByTagName('title')[0]
11       .firstChild.nodeValue;
12     /*...*/
13     feedAsJson.items.push(feedItem);
14   }
15   this.sortItems(feedAsJson);
16 }
```

Code-Ausschnitt 3.6: Abfragen der Feeds im NewsFeedComponent

```
1  getFeeds() {
2    this.feedSources = this.feedService.getFeedSources();
3    if ((this.feeds = this.feedService.feedsAsJSON).length === 0) {
4      this.feedSources.forEach(src =>
5        this.feedService.getNewsFromFeed(src.url).subscribe(feed =>
6          {
7            this.feedService.parseFeedFromXmlToJson(feed, src.name);
8          });
9      );
10 }
```

funktioniert, ist in Zeile 10 dargestellt. Weil mit `getElementsByTagName` immer eine Liste vom Typ `NodeList` zurückgegeben wird und ein Post genau einen Titel hat, wird ebenjener über den Index '0' erreicht. Um den eigentlichen Inhalte `item`-Elements zu erhalten, ist der Aufruf `.firstChild.nodeValue` notwendig, was am Aufbau eines solchen `Node`-Elements liegt.

Mit den Kategorien des Post wird analog in einer weiteren Schleife (hier nicht gezeigt) verfahren und diese Kategorien dann dem Post hinzugefügt. Als nächstes werden die Einträge des Feeds chronologisch sortiert, bevor dieser in der Liste der Feeds gespeichert wird.

Hier nicht gezeigt sind die Funktionen zum Sortieren, zum Abfragen des Feeds vom Server sowie zum Abfragen der Liste der Feed-Quellen. Diese Liste befindet sich in einer eigenen Datei⁴ und enthält eine Reihe von Objekten, die sich aus einem Namen sowie der URL des Feeds zusammensetzen. Auf diese Weise können sehr

⁴/news-feed/RssFeeds.json

einfach weitere RSS-Feeds eingebunden werden. Das Abfragen der Daten durch den HTTP-Client unterscheidet sich hier insofern von dem Aufruf in Beispiel 2.11, als keine Klasse, Instanzen derer man erwartet, angeben wird, sondern dem get-Aufruf die Option “{`responseType: 'text'`}“ mitgegeben wird. Damit wird deklariert, dass kein(e) JSON-Objekt(e) sondern nur einfacher Text - also im Prinzip ein String - erwartet wird. Außerdem wird im Aufruf der eigentlichen URL der Ressource die URL des Proxy-Servers vorangestellt. Anhand des Ausschnitts 3.12 in Kapitel 3.4.3 wird dies genauer erklärt.

Bei der Initialisierung der `NewsFeedComponent`⁵ werden zwei Funktionen aufgerufen: In einer wird der Titel der Seite sowie der Toolbar (vgl 3.3.3.4) geändert und in der anderen das Abfragen der Feeds durchgeführt. In letzterer - wiedergegeben in Ausschnitt 3.6 - werden zunächst die Quellen abgefragt und danach geprüft, ob im Service bereits die Feeds als JSON gespeichert sind - also ob die Liste derer leer ist oder nicht (Zeile 3). Allerdings wird hier kein Objekt “kopiert“, sondern eine Referenz gesetzt, sodass, wenn beim Parsen im Service die Liste der Feeds aktualisiert wird (vgl. Ausschnitt 3.5, Zeile 5), dies auch automatisch mit der Liste in der Komponente geschieht. Beim Aufruf der Methode zum Übersetzen in Zeile 6 findet daher keine Rückgabe statt.

Weil durch Navigieren zwischen Seiten über das Menü die Komponenten mitsamt ihrer Daten immer “zerstört“ und neu “konstruiert“ werden, müssten jedes mal, wenn eine Komponente erzeugt wird, die Daten neu vom Server angefragt und anschließend geparsst werden. Wenn später der Service Worker hinzugefügt wird, können zwar die XML-Daten aus dem Cache bezogen werden, um etwas Zeit zu sparen, trotzdem müssen sie dann noch in JSON-Objekte übersetzt werden. Da der Service (bzw. alle Services, das gleiche Prinzip gilt auch für die anderen Module) global existiert, gehen die Daten - einmal übersetzt und im Service gespeichert - nicht verloren, wenn zu einer anderen Komponente und zurück navigiert wird.

Das Code-Beispiel 3.7 zeigt einen Ausschnitt aus dem Template dieser Komponente⁶. In einem Registerkartenlayout (`mat-tab`) werden die Posts als Liste von Karten (`mat-card`) angezeigt, wobei im Kopfbereich der Titel sowie das Veröffentlichungsdatum stehen, im Inhaltsbereich die Beschreibung (also eine Zusammenfassung) eingefügt wird und im Fußbereich die Kategorien aufgelistet sind. Klickt man eine solche Karte an, wird der Link des Beitrags geöffnet, wobei dies über eine Funktion - also mittels *Event Binding* - geschieht (Zeile 3), in der geprüft wird, ob ein Mobilgerät verwendet wird und in diesem Fall eine Bestätigung angefordert wird, damit beim Scrollen durch die Meldungen nicht unbeabsichtigt ein Link geöffnet wird.

⁵/news-feed/news-feed-component/news-feed.component.ts

⁶/news-feed/news-feed-component/news-feed.component.html

Code-Ausschnitt 3.7: Ausschnitt des NewsFeed-Templates

```
1  <mat-tab *ngFor="let feed of feeds" label="{{feed.name}}">
2    <div class="output">
3      <mat-card *ngFor="let item of feed.items" (click)="
4        openLinkForItem(item)" class="mat-elevation-z3">
5        <mat-card-header>
6          <mat-card-title>
7            <h3>{{ item.title }}</h3>
8          <mat-card-subtitle>{{item.pubDate | date: 'dd.MM.yyyy' }}</mat-card-subtitle>
9        </mat-card-header>
10       <mat-card-content [innerHTML]="item.description"></mat-card>
11         -content>
12       <mat-card-footer>
13         <!--list of -->
14       </mat-card-footer>
15     </mat-card>
16   </div>
17 </mat-tab>
```

Abbildung 3.4a zeigt, wie der Feed der als Karten dargestellten Meldungen auf einem Smartphone aussieht. Zunächst einmal sieht man, dass der Titel der Toolbar am oberen Bildrand geändert wurde. Darunter sieht man die Tabs (Registerkarten), mit denen zwischen den Quellen gewechselt werden kann und danach sind die Posts des jeweiligen Feeds aufgelistet.

3.4.2 Campus-Karte

Damit in diesem Modul die Gebäude auf der Karte angezeigt werden können, muss bei Initialisierung der Komponente zunächst ein Liste ebendieser bezogen werden. Da diesen Daten im JSON-Format vorliegen, sieht die `getBuildingsList`-Methode des `MapService`⁷ im Prinzip genauso aus wie in Beispiel 2.11, nur das ein Array von `Building`-Objekten erwartet wird. Ein solches `Building`-Objekt⁸ enthält: einen Namen, einen „Nicknamen“, eine Adresse, und ein `Coordinate`-Objekt⁹, welches wiederum aus Längen- und Breitengrad besteht. Dabei könnte man die Daten im Service speichern, sobald der Service Worker hinzugefügt wird, speichert dieser die Ressource ohnehin im Cache, sodass dies an dieser Stelle nicht notwendig ist.

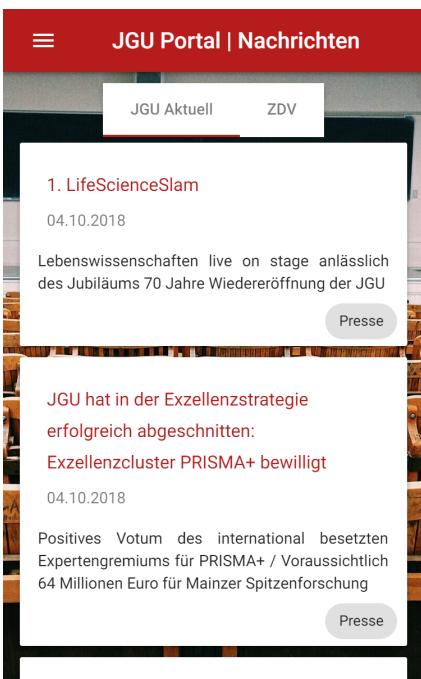
Um die Suchfunktion mit Auto-Vervollständigung durchführen zu können, müssen in `MapComponent`¹⁰ zwei Dinge geschehen: Eine Kopie der Gebäudeliste muss angelegt werden und in dieser Liste müssen diejenigen Gebäude hinterlegt werden, deren Namen mit der aktuellen Eingabe übereinstimmen. Die zwei Funktionen in

⁷/map/map.service.ts

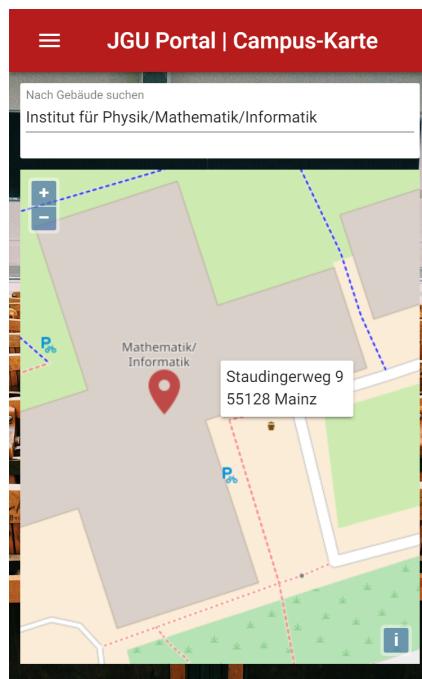
⁸/models/building.ts

⁹/models/coordinate.ts

¹⁰/map/map-component/map.component.ts



(a) Das Nachrichtenmodul



(b) Das Kartenmodul

Abb. 3.4: Das Nachrichten - und das Kartenmodul

Code-Ausschnitt 3.8: Auszug aus MapComponent

```

1  private getBuildings() {
2    this.mapService.getBuildingsList()
3      .subscribe(res => {
4        this.buildings = res;
5        this.filteredBuildings = this.ctrl.valueChanges
6          .pipe(
7            startWith(''),
8            map(buildingName => this.filterBuildings(buildingName))
9          );
10     });
11   }
12
13  private filterBuildings(name: string): Building[] {
14    const filterValue = name.toLowerCase();
15    return this.buildings.filter(building => {
16      return building.name.toLowerCase().startsWith(filterValue)
17      || (building.nickname && building.nickname.toLowerCase()
18        .startsWith(filterValue));
19    });
}

```

Ausschnitt 3.8 setzen dies um. Hierfür wird nach Empfang der Daten ein Observable (`FilteredBuildings`) initialisiert (Zeile 5), welches ein Array vom Typ `Building` „kapselt“. Über ein `FormControl`-Objekt (`ctrl`) wird diesem Observable jedes Mal durch die Pfeilfunktion in Zeile 8 eine neue Liste an Gebäuden zugewiesen, wenn eine Eingabe getätigt wird. Genauer: bei `ctrl.valueChanges` handelt es sich ebenfalls um ein Observable, dessen Zustand - also der Wert - sich ändert, wenn ein Eintrag aus der Liste der Gebäude ausgewählt wird. Der Aufruf von `startWith` ist wichtig, damit initial alle Gebäude als Drop-Down-Liste angezeigt werden. In der `filterBuildings`-Methode (Zeile 13) werden dann diejenigen Gebäude herausgesucht, deren Name oder Nickname mit dem übergebenen String beginnt.

Das Anzeigen der Karte wird in der Methode `ngAfterViewInit` implementiert, wobei diese Methode über das Interface `AfterViewInit` bereitgestellt wird und erst aufgerufen wird, wenn das Template der Komponente auch vollständig initialisiert wurde. Warum das so sein muss, lässt sich an Ausschnitt 3.9. In Zeile 9 wird das Attribut `target` definiert, wobei eine ID - in diesem Fall „map“ - angegeben wird, welche ein DOM-Element referenziert, als dessen Kind ein `Canvas`-Element erzeugt wird, auf welchem wiederum die Karte erstellt wird. Würde diese Initialisierung bereits durch die `onInit`-Methode ausgelöst, kann es sein, dass das Template noch nicht vollständig ins DOM eingebunden wurde, sodass dort noch gar kein Element mit der ID „map“ vorhanden ist - die Initialisierung würde also scheitern.

Neben dem `target`-Attribut werden noch weitere Eigenschaften des Map-Objektes festgelegt: unter `layers` werden die einzelnen Schichten der Karte definiert, wobei das `TileLayer` das eigentliche Kartenmaterial enthält, welches in diesem Fall von OpenstreetMap (OSM) stammt, und mit dem `VectorLayer` Vektordaten oder - wie in diesem Fall - Bilder bzw. Icons auf der Karte angezeigt werden können. Über das Attribut `view` können initiale Position und Zoom festgelegt werden - in diesem Fall wird (in etwa) das Zentrum des Universitäts-Campus angezeigt und so nah „herangezoomt“, dass dessen Darstellung ungefähr den ganzen Bildschirm füllt. Hat man ein Gebäude aus der Vorschlagsliste ausgewählt, geschehen in der Funktion `centerMapAtBuilding` (hier nicht gezeigt) im Wesentlichen vier Dinge:

- Die Zoom-Stufe wird erhöht.
- Die Karte wird um das ausgewählte Gebäude zentriert.
- Das Gebäude wird mit einem Icon markiert.
- Ein Textfeld, welches die Adresse des Gebäudes enthält, wird neben dem Marker angezeigt.

Code-Ausschnitt 3.9: Initialisieren der Karte

```
1  ngAfterViewInit() {
2      this.myMap = new Map({
3          layers: [
4              new TileLayer({
5                  source: new OSM // Base Layer with tiles
6              }),
7              new VectorLayer({ source: this.markerSource })
8          ],
9          target: 'map',
10         view: new View({
11             center: fromLonLat([8.2411, 49.9923], 'EPSG:3857'),
12             zoom: 16
13         })
14     );
15     this.popupOverlay = new Overlay({
16         element: this.popup.nativeElement,
17         positioning: 'bottom-left',
18         stopEvent: false,
19         position: this.myMap.getView().getCenter()
20     );
21     /*...*/
22     this.myMap.addOverlay(this.popupOverlay as ol.Overlay);
23 }
```

Um das Textfeld anzuzeigen wird, wie ab Zeile 15(3.9) dargestellt, ein Overlay erzeugt und dieses dem Karten-Objekt hinzugefügt (ab Zeile 15). Hierbei wird mit `this.popup.nativeElement` ein DOM-Element referenziert, welches als Kindelement das eigentliche Popup hat. Das zugehörige Template¹¹ ist in Code-Beispiel 3.10 wiedergegeben, wo das Element für die Karte sowie für das Popup-Overlay samt Adresskarte ab Zeile 12 definiert werden. Während der Marker solange angezeigt wird, bis man ein anderes Gebäude ausgewählt oder zu einer anderen Seite navigiert, wird das Textfeld ausgeblendet, wenn die Zoomstufe zu niedrig ist, da dieses anderenfalls den Marker überlagern würde. Dies geschieht, indem dem Map-Objekt ein *EventListener* zugewiesen wird, welcher auf das 'scroll'-Event reagiert und das Attribut `showAddress` des MapComponent in Abhängigkeit Zommstufe auf `true` oder `false` setzt. Mittels *Attribute Binding* wird in Zeile 14 erreicht, dass - je nachdem, ob `showAddress` 'wahr' oder 'falsch' ist - die *Opacity* des Elements auf '0' oder '1' gesetzt wird, wobei der Übergang mit CSS¹² animiert wird. Die `*ngIf`-Direktive ist notwendig, damit die Adresskarte nicht angezeigt wird wenn (noch) kein Gebäude ausgewählt wurde.

Vorher werden das Eingabefeld sowie die Liste der Gebäude angegeben. Auch hier wird zweimal *Attribute Binding* verwendet: Zum einen um das `formControl`-Objekt `ctrl` zuzuweisen und zum anderen, um das Eingabefeld mit der Auto vervollständigung zu verknüpfen (Zeile 3). Anschließend werden alle Gebäude als "Option"

¹¹/map/map-component/map.component.html

¹²/map/map-component/map.component.scss

Code-Ausschnitt 3.10: Template des MapComponent

```
1 <mat-form-field class="selectForm">
2   <input matInput type="text" id="searchField" name="searchField"
3     [FormControl]="ctrl" [matAutocomplete]="auto">
4 </mat-form-field>
5 <mat-autocomplete #auto="matAutocomplete">
6   <mat-option *ngFor="let building of filteredBuildings | async"
7     [value]="building.name"
8     (onSelectionChange)="centerMapAtBuilding(building,
9       $event)">
10    {{ building.name }}</mat-option>
11 </mat-autocomplete>
12 <div id="map">
13   <div #popup>
14     <mat-card id="address" *ngIf="focusedBuilding !== undefined"
15       [style.opacity]="showAddress? '1': '0'">
16       {{ focusedBuilding.defaultAddress }}</mat-card>
17   </div >
18 </div>
```

gelistet. Hierbei wird die `async`-Pipe genutzt weil es sich bei `filteredBuildings` um ein *Observable* handelt; dessen Wert sich kann also jederzeit ändern und wenn das geschieht, muss die Liste neu erstellt werden.

Bei jeder Option wird hier der Name (Zeile 8) und - falls vorhanden - der *Nickname* (Zeile 9) des Gebäudes angezeigt. Außerdem wird mittels *Data-Binding* jeder Option der Name des jeweiligen Gebäudes als `value` zugeordnet, sodass dieser in das Eingabefeld übertragen wird, wenn man die Option auswählt. Wird ein solches Element ausgewählt, wird mittels *Event Binding* (Zeile 7) die Methode zum Zentrieren etc. aufgerufen. Dabei wird neben dem ausgewählten Gebäude auch das Event übergeben. Das ist notwendig, weil bei Veränderung der Auswahl das “`selectionChange`“-Event sowohl durch das “Abwählen“ des alten Gebäudes, als auch durch Auswählen des neuen Gebäudes ausgelöst wird - also zweimal. Durch Übergabe des Events ist es möglich, zu prüfen, welches der beiden Ereignisse tatsächlich eingetreten ist. Wurde kein Gebäude ausgewählt, kann die Karte auch nicht zentriert werden und der Funktionsaufruf wird beendet.

3.4.3 Busfahrpläne

Die Abfahrtszeiten für dieses Modul werden über die API des Rhein-Main Verkehrsverbundes bezogen und können als JSON angefragt werden. Für dieses Modul wurden die in Abbildung 3.5 dargestellten Klassen modelliert. Das Attribut `name` der

Code-Ausschnitt 3.11: Ausschnitt aus BusStopComponent

```

1  private initBusStops() {
2      this.busStops = this.busStopService.getBusStops();
3      this.busStops.forEach(busStop => {
4          this.busStopService.getDepartureBoardForStation(busStop.id)
5              .subscribe(departureBoard => {
6                  busStop.departureBoard = departureBoard;
7              });
8      });
9  }

```

Departure¹³ ¹⁴-Klasse bezieht sich dabei auf den Namen der Bus- bzw. Straßenbahnlinie und direction gibt den Namen der Endhaltestelle an. Die Attribute rtTime und rtDate in derselben Klasse beziehen sich auf die derzeit erwartete Verspätung. Während die “Abfahrtstafeln“ (Departure) über die API bezogen werden müssen, sind die Bushaltestellen (BusStop¹⁵¹⁶) in einer Datei lokal gespeichert¹⁷. Diese Datei wird über den BusStopService¹⁸ importiert, welcher wiederum eine Methode enthält, damit diese Liste der Haltestellen in der Komponente¹⁹ abgefragt werden kann. In dieser Komponente ist die in Ausschnitt 3.11 angegebene Funktion implementiert.

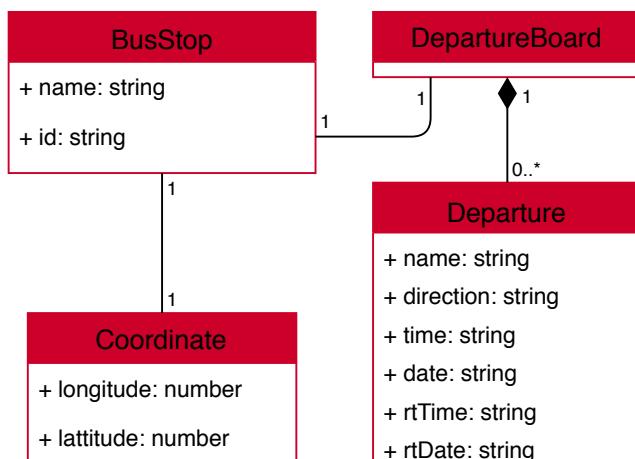


Abb. 3.5: UML-Diagramm der Klassen des Busfahrplan-Moduls

In dieser wird zunächst die Liste der Bus- und Straßenbahnhaltestellen bezogen (Zeile 2) und dann für jede dieser Haltestellen die Abfahrtszeiten abgefragt, sodass diese bei der jeweiligen Haltestelle gespeichert werden können.

Das eigentliche Abfragen ist hier etwas komplexer, da die URL für die Anfrage aus mehreren Parametern zusammen gesetzt werden muss, wie in Ausschnitt 3.12

¹³/models/departureBoard.ts

¹⁴/models/departure.ts

¹⁵/models/busStop.ts

¹⁶/models/coordinate.ts

¹⁷/bus-schedule/busStops.json

¹⁸/bus-schedule/bus-schedule.service.ts

¹⁹/bus-schedule/bus-schedule-component/bus-schedule.component.ts

Code-Ausschnitt 3.12: Abfragen der Abfahrtstafeln im BusScheduleService

```
1 const proxy = 'https://cors-anywhere.herokuapp.com/';
2 const apiKey = 'accessId=26594e9d-0888-496c-a664-5b0b888f3882';
3 const format = '&format=json';
4 const maxLength = '&maxJourneys=10';
5 const url = 'https://www.rmv.de/hapi/departureBoard?' + apiKey +
    format + maxLength;
6
7 getDepartureBoardForStation(stationID: number): Observable<
    DepartureBoard> {
8     return this.http.get<DepartureBoard>(proxy + url + `&id=${{
    stationID}`);
9 }
```

dargestellt. Zunächst wird der apiKey-angegeben, ohne welchen ein Zugriff auf die RMV-API gar nicht möglich ist. Durch den Parameter format wird erreicht, dass die Daten als JSON und nicht im XML-Format übertragen werden, was der Standard wäre. Weiterhin wird mit maxLength festgelegt, dass höchstens die nächsten 10 Abfahrten übertragen werden. Aus diesen Parametern wird dann die eigentliche URL zusammengesetzt. In der eigentlichen Anfrage (Zeile 6) wird dann noch die URL des Proxy-Servers vorangestellt, sowie die ID der Bushaltestelle, deren Abfahrtstafel bezogen werden soll, angehängt.

Dabei könnten die Daten - also die Abfahrtstafeln theoretisch im Service gespeichert werden, da sich die Abfahrtszeiten (oder Verspätungen) aber fast im Sekundentakt ändern, wird hier darauf verzichtet. Somit liegen immer die aktuellsten Informationen vor, wenn von dieser Komponente zu einer anderen und zurück navigiert wird. Weiterhin wäre es ebenso möglich gewesen, mittels setInterval die Daten regelmäßig auch dann zu aktualisieren, wenn der Nutzer auf dieser Seite verbleibt. Da so jedoch relativ schnell das Penum der maximal erlaubten Anfragen pro Stunde bzw. pro Tag überschritten würde, wenn entsprechend viele Studierende dieses Angebot nutzen. Dementsprechend wurde auch hierauf verzichtet. Außerdem wurde aus demselben Grund davon abgesehen diese Daten über den Service Worker im Cache zu speichern.

Abbildung 3.6a zeigt die Ansicht der Abfahrtszeiten. Zu jeder Haltestelle gibt es ein Akkordeon-Element in dem die nächstes an dieser Station ankommenden Busse und Straßenbahnen inklusive deren Informationen tabellarisch aufgelistet sind. Kommen in nächster Zeit keine Busse oder Straßenbahnen an, wird “keine Abfahrten in nächster Zeit“ angezeigt.

Wie eine solche Tabelle mit *Angular Material* generiert werden kann, ist in dem Auszug 3.13 dargestellt²⁰. Hierfür wird zunächst die Tabelle mittels *Data-Binding*

²⁰/map/map-component/map.component.html

(a) Ansicht der Busfahrpläne

Zeit	Aktuell	Linie	Richtung
13:43	13:48	Bus 68	Budenheim Bahnhof
13:46	13:47	Tram 53	Mainz-Lerchenberg Hindemithstraße
13:47	13:47	Bus 6	Wiesbaden Nordfriedhof
13:47	13:47	Bus 54	Ginsheim-Gustavsburg- Ginsheim Friedr.-Ebert- Platz

(b) mit geöffnetem Menü

Abb. 3.6: Das Abfahrtszeiten- und das Personensuchmodul

mit einem Datensatz verknüpft, der dargestellt werden soll. Außerdem wird für die jeweilige Station geprüft, ob Abfahrten vorhanden sind, da anderenfalls ein hier nicht gezeigtes Template (referenziert mit noDepartures) angezeigt wird, welches lediglich “keine Abfahrten in nächster Zeit“ enthält (Zeile 1). Anschließend werden für jede Spalte Container definiert, in denen deren Überschrift und eine Liste der Werte, über die iteriert werden soll, angegeben wird. Außerdem wird angeben, welchen Inhalt die Zeilen dieser Spalte haben sollen (Zeile 5). In diesem Fall wird hier die Spalte der geplanten Abfahrtszeiten definiert. Da diese Information getrennt in Datum und Uhrzeit vorliegt, müssen diese Angaben konkateniert werden um eine ISO 8601-konforme Angabe zu erhalten, der dann durch die Pipe formatiert werden kann.

Mit Hilfe dieser Container werden dann die eigentlichen Zeilen der Tabelle erzeugt: In Zeile 10 wird die Kopfzeile generiert und in Zeile 11 die Inhaltszeilen. Dabei wird mit displayedColumns jeweils ein String-Array übergeben, in dem die Namen der Spalten die angezeigt werden sollen, aufgelistet sind. Diese Namen beziehen sich auf die zuvor definierten Container, sodass die Reihenfolge der Spalten von der Reihenfolge der Spaltennamen im Array und nicht von der der Container abhängig ist. Indem Einträge aus dem Array gelöscht oder diesem hinzugefügt werden, können Spalten ein- oder ausgeblendet werden. Dementsprechend müssen auch nicht alle Container referenziert werden; nicht genutzte Container sind dann nicht Teil des DOMs.

Code-Ausschnitt 3.13: Tabelle der Abfahrtszeiten

```

1  <table mat-table *ngIf="station.departureBoard.Departure !==
2      undefined; else noDepartures"
3      [dataSource]="station.departureBoard.Departure"
4          class="mat-elevation-z3">
5      <ng-container matColumnDef="Abfahrt (geplant)">
6          <th mat-header-cell *matHeaderCellDef>Zeit</th>
7          <td mat-cell *matCellDef="let departure">
8              {{ departure.date + 'T' + departure.time | date:'HH:mm'}}</td>
9      </ng-container>
10     <!-- ... -->
11     <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
12     <tr mat-row *matRowDef="let row; columns: displayedColumns;">
13         </tr>
14     </table>

```

3.4.4 Personensuche

Der Service²¹ dieses Moduls beinhaltet drei Methoden, welche auszugsweise in Beispiel 3.14 wiedergegeben sind. Der Aufruf des HTTP-Client ist demjenigen aus dem vorherigen Kapitel recht ähnlich. Nach deren Aufruf wird ein Liste von Personen zurückgegeben, deren Nachname mit dem übergebenen Substring beginnt. Allerdings auch hier die Daten wieder in XML-Form vor, sodass die Funktion `parsePersonsFromXmlToJson` (Zeile 10) notwendig ist, welche - wie am Namen unschwer erkennbar - aus der erhaltenen Liste der Personen als XML-String ein Array von Personen als JSON-Objekt erzeugt. Dabei werden die in Abbildung 3.7 abgebildeten Klassen `Person`²² und `Contact`²³ verwendet. Hierfür wird zunächst eine leere Liste erstellt und überprüft, ob die Antwort des Servers überhaupt Personen enthält - also ob Personen gefunden wurden. Ist das nicht der Fall, so wird einfach die leere Liste zurückgegeben. Andernfalls werden die Daten der Personen geparsst und die Liste aller gefundenen Personen als JSON zurückgegeben. Im Wesentlichen funktioniert diese Ausgabe wie bereits in Kapitel 3.4.1 erklärt wurde, weshalb hier auf einer erneute Erklärung des Vorgangs verzichtet werden soll.

Zum Parsen und Modellieren der Daten (sowohl bei diesem, als auch bei allen anderen Modulen) sei hier noch erwähnt, dass in der Regel für einen Attribut der Typ `string` gewählt wurde, auch wenn ein anderer im ersten Moment intuitiver gewesen wäre; im Falle von IDs beispielsweise handelt sich normalerweise um Zahlenwerte sodass man entsprechende Variablen auch. Es wurde nur dann davon abgewichen, wenn ein Vorteil daraus gezogen werden kann. Am Ende ist es unerheblich, ob Zahlen oder Strings in ein Template eingesetzt werden, daher ist es unnötig, Werte, die als Text vorliegen, in Zahlen umzuwandeln, wenn sie nicht explizit für Berech-

²¹/person-search/person-search.service.ts

²²/models/person.ts

²³/models/contact.ts

Code-Ausschnitt 3.14: Auszug aus PersonSearchService

```
1  findPersons(name: string): Observable<string> {
2      return this.http.get(proxy + url + `&fullname=${name}` , {
3          responseType: 'text'});
4
5  parsePersonsFromXmlToJson(response: string): Person[] {
6      const foundPersons: Person[] = [];
7      /*...*/
8      return foundPersons;
9  }
10
11 getPerson(id: string): Observable<string> {
12     const ref = id.replace('Person.', '').split('.').join('/');
13     return this.http.get(proxy + personUrl + ref, {responseType:
14         'text'});
15 }
```

nungen oder sonstige Operationen benötigt werden. Als Gegenbeispiel seien hier Telefonnummern genannt, die nicht weiterverarbeitet werden, und daher als “string“ verbleiben können.

Die verbleibende dritte Methode (getPerson) ist für dieses Modul nur mittelbar nötig; Im nächsten Modul werden Veranstaltungen aufgelistet, zu denen auch immer eine Kontaktperson angegeben wird. Der Übersichtlichkeit halber wurde darauf verzichtet, deren Kontaktdaten dort ebenfalls direkt anzuzeigen. Stattdessen ist ein Link hinterlegt, welcher zu diesem Modul weiterleitet, sodass hier die entsprechenden Informationen aufgelistet werden. Dieser Person ist dabei eine ID zugeordnet, in der Punkte durch Schrägstriche ersetzt sowie der Substring “Person.“ entfernt werden müssen, bevor dann über eine andere URL als diejenige, über welche die eigentliche Suche funktioniert, die Daten der Person bezogen werden können.

Daraus ergibt sich, dass die `ngOnInit`-Methode der Komponente²⁴ etwas anders gestaltet sein muss (siehe Ausschnitt 3.15): Neben dem obligatorischen, hier nicht gezeigten Methodenaufruf zum Ändern des Titels der Seite und der Toolbar wird geprüft, ob in der URL der Parameter “id“ gesetzt ist und in diesem Fall die Methode `getPerson` aufgerufen. Diese sieht im Grunde genauso aus, wie der Teil der Methode `searchPerson` ab Zeile 16, mit dem Unterschied, dass die Variable `isSearching` nicht verändert wird.

Bei der Initialisierung der Komponente wird zudem die Größe des Bildschirms (d.h. eigentlich die des Fensters) geprüft, weil für kleine Anzeigen ein anderes Layout gewählt wurde, also für große Displays. Während auf großen Geräten die Daten in tabellarischer Form angezeigt werden, musste auf Grund der hohen Anzahl an Spalten auf kleinen Geräten ein anderer Weg gewählt werden. Dort wird für jede

²⁴/person-search/person-search-component/person-search.component.ts

Code-Ausschnitt 3.15: Ausschnitt aus PersonSearchComponent

```
1  ngOnInit() {
2      const id = this.route.snapshot.paramMap.get('id');
3      if (id) {
4          this.getPerson(id);
5      }
6      this.mobile = window.screen.width <= 768;
7  }
8
9  searchPerson() {
10     if (this.searchName === undefined || this.searchName === '')
11         || this.lastSearch === this.searchName) { return; }
12     this.isSearching = true;
13     this.lastSearch = this.searchName;
14     this.personSearchService.findPersons(escape(this.searchName))
15         .subscribe(response => {
16             this.foundPersons =
17                 this.personSearchService.parsePersonsFromXmlToJson(
18                     response);
19         this.isSearching = false;
19     });
19 }
```

Person ein Akkordeon-Element angezeigt, in welchem deren Kontaktdaten als Liste dargestellt sind.

Klickt man auf den Button zum Suchen oder betätigt die Enter-Taste, wird mittels *Event Binding* die Methode `searchPerson` aufgerufen. In dieser wird zunächst geprüft, ob die Eingabe undefiniert oder leer ist, sowie ob die Eingabe unverändert ist - also ob erneut nach dem gleichen Namen gesucht wird. In all diesen Fällen wird die Suche abgebrochen. Als nächstes wird `isSearching` als `true` gesetzt, was zur Folge hat, dass ein Template angezeigt wird, welches wiederum eine Ladeanimation anzeigt. Danach wird der zuletzt gesuchte Name neu gesetzt, bevor schlussendlich die eigentliche Suche über den `PersonSearchService` durchgeführt. Wichtig ist, dass vor dem Aufruf der Suche noch durch die Methode `escape` Sonderzeichen und Umlaute ersetzt werden. Wurde diese Suche ausgeführt, werden die erhaltenen Daten durch den Service übersetzt und die Liste der gefundenen Personen (`foundPersons`) neu gesetzt. Zudem wird die Ladeanimation wieder ausgeblendet indem `isSearching` auf `false` gesetzt wird. Außerdem wird ein Hinweisfeld angezeigt, wenn keine Ergebnisse gefunden werden konnten.

Das Template²⁵ dieser Komponente ist auf Grund der zwei unterschiedlichen Layouts relativ sperrig und wird daher hier nicht eingebunden. Die wesentlichen Bestandteile wurden aber bereits beschrieben und während der Aufbau von Tabellen bereits im letzten Kapitel erklärt wurde, wird im nächsten erläutert wie die für das “mobile“

²⁵/person-search/person-search-component/person-search.component.html

Layout verwendeten Akkordeon-Elemente genutzt werden. Abbildung 3.6b zeigt, wie diese Komponente nach erfolgreicher Suche aussieht.

3.4.5 Veranstaltungen

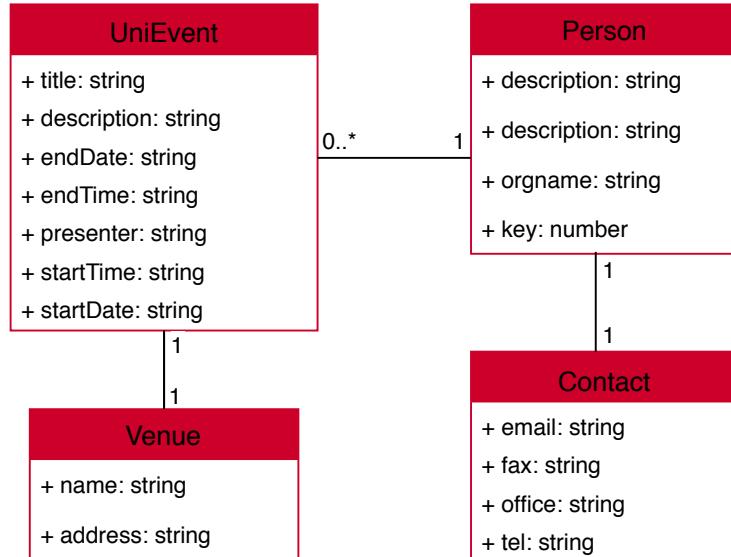


Abb. 3.7: UML-Diagramm der Klassen für die Personensuche und das Veranstaltungsmodul

Die Vorgehensweise in diesem Modul unterscheidet sich nur unwesentlich von der in den vorhergehenden Modulen; beim Initialisieren der Komponente wird eine Reihe von Veranstaltungen geladen und weil diese auch in XML-Form vorliegen, müssen diese in JSON-Objekte übersetzt werden. Neben den bereits im vorhergehenden Kapitel beschriebenen Klassen `Person` und `Contact` wurden dafür die ebenfalls in Abbildung 3.7 abgebildeten Klassen `UniEvent`²⁶ und `Venue`²⁷ verwendet.

In diesem Modul wurde sich zu Nutze gemacht, dass der `CalendarService`²⁸ global existiert; wurden die Daten geparsst, werden sie im Service gespeichert, sodass beim Initialisieren der Komponente zunächst geprüft werden kann, ob die Liste der Veranstaltungen bereits als JSON-Array vorliegt. Wie das umgesetzt wurde, zeigt Ausschnitt 3.16 aus dem `CalendarComponent`²⁹. Nachdem auch hier wieder der Titel der Seite und der Toolbar geändert wurde, wird die Liste der im Service gespeicherten Veranstaltungen abgefragt (`getEvents()` gibt ein Array vom Typ `UniEvent` zurück) und geprüft, ob diese leer ist und nur dann die Daten vom Server bezogen (Zeile 4), wobei diese dann natürlich anschließend geparsst werden müssen.

Das Übersetzten der Daten funktioniert hier im Wesentlichen wie bereits in den vorhergehenden Modulen. Eine Schwierigkeit war an dieser Stelle war, dass in der

²⁶/models/uniEvent.ts

²⁷/models/venue.ts

²⁸/calendar/calendar.service.ts

²⁹/calendar/calendar-component/calendar.component.ts

Code-Ausschnitt 3.16: Initialisierung des CalendarComponent

```
1  ngOnInit() {
2    this.setTitle();
3    if ((this.events = this.calendarService.getEvents()).length ===
4        0) {
4      this.calendarService.getEventsFromServer()
5        .subscribe(events => {
6          this.events =
7            this.calendarService.parseEventsFromXmlToJson(events);
8        });
9  }
```

Code-Ausschnitt 3.17: Ausschnitt aus CalendarService

```
1  parseEventsFromXmlToJson(response: string): UniEvent[] {
2    /*...*/
3    this.parsePersonsFromXmlToJson(personsAsXml);
4    for (let index = 0; index < eventsAsXml.length; index++) {
5      const eventAsXml = eventsAsXml[index];
6      const event = new UniEvent();
7      const contactKey = (eventAsXml.getElementsByTagName('contact'
8        )[0].firstChild as Element).getAttribute('key');
9      event.person = this.persons.get(contactKey);
10     /*...*/
11     this.events.push(event);
12   }
13   this.sortEvents(this.events);
14   return this.events;
15 }
16 private parsePersonsFromXmlToJson(personsAsXml: NodeList<
17   Element>) {
18   for (let index = 0; index < personsAsXml.length; index++) {
19     /*...*/
20     this.persons.set(key, person);
21   }
}
```

Anwort vom Server Kontaktperson (Person) und Veranstaltungsort (Venue) nicht in die Veranstaltung (UniEvent) “eingebettet“ sind - sowie es beispielsweise bei den Kontaktdaten der Personen der Fall ist. Stattdessen sind sie jeweils einzeln gelistet. Dies macht aus Sicht der serverseitigen Verwaltung ja auch durchaus Sinn, schließlich sind “Veranstaltung“, “Ort“ und “Person“ drei unabhängige Entitäten, die auch individuell verwaltet werden.

Für die weitere Verarbeitung wurden der Veranstaltung die zugehörige Ansprechperson und sowie der Veranstaltungsort als “Kindobjekte“ zugewiesen. Da die Event-Objekte logischerweise die Kontaktperson sowie den Veranstaltungsort über deren ID referenzieren, wurde dies wie in den zwei Funktionen, welche in Ausschnitt 3.17 wiedergegeben werden. In der Funktion zum “Übersetzten“ der Daten werden zunächst die Personen und die Räume (hier nicht gezeigt) verarbeitet, wobei dies jeweils in

Code-Ausschnitt 3.18: Ausschnitt des Templates der

```
1 <mat-accordion>
2   <mat-expansion-panel *ngFor="let event of events">
3     <mat-expansion-panel-header>
4       <mat-panel-title [ngClass]="panelOpenState? 'expanded' : 'collapsed' ">
5         <span class="date">{{ event.startDate | date: 'dd.MM' }}</span> {{ event.title }}
6       </mat-panel-title>
7     </mat-expansion-panel-header>
8     <mat-list>
9       <!-- ... -->
10      <mat-list-item>
11        <mat-icon svgIcon="person" color="primary"></mat-icon>
12        <a routerLink="/Personensuche/{{ event.person.key }}">
13          Kontakt
14        </a>
15      </mat-list-item>
16    </mat-list>
17  </mat-expansion-panel>
18 </mat-accordion>
```

weiteren, äquivalenten Methoden geschieht. In beiden Fällen wird über die Liste der Elemente iteriert, diese geparsst und anschließend in einer Map gespeichert, wobei die Person bzw. der Raum der zugehörigen ID zugewiesen wird (Zeile 20). Da in den Events ebenjene IDs hinterlegt sind (Zeile 8), können über diese dem erzeugten *UniEvent*-Objekt das entsprechende Person - bzw Venue-Objekt zugewiesen werden (Zeile 9). Danach wird das *UniEvent*-Objekt einer Liste hinzugefügt und diese vor der Rückgabe chronologisch sortiert.

Wie im vorherigen Kapitel angedeutet, wurde für das Layout dieser Komponente ein Akkordeonelement verwendet. Ein Ausschnitt des Templates³⁰ ist in Beispiel 3.18 angegeben. Ein Element, das auf- und zuklappt, wenn man es anklickt, wird eigentlich schon durch den *mat-expansion-panel*-Tag definiert. Umgibt man dieses mit dem *mat-accordion*-Tag, so wird zwischen einem aufgeklappten Element und den benachbarten ein *Padding* eingefügt. In einem solchen Panel wird dann ein Kopfbereich mit einem Titel angeben, gefolgt vom Inhalt, der nur im aufgeklappten Zustand sichtbar ist. In diesem Fall handelt es sich um eine Liste der das Event beschreibenden Attribute welche unter anderem auch die Kontaktperson enthält. Im vorhergehenden Kapitel wurde bereits darauf hingewiesen, dass deren Daten hier nicht angezeigt werden. Stattdessen wird ein Link eingefügt, der zur Komponente der Personensuche führt, wobei die ID der Person an den Pfad angehängt wird (Zeile 12). Das Ergebnis sieht dann so aus wie in Abbildung 3.8a.

³⁰/calendar/calendar-component/calendar.component.html

(a) Ansicht der Veranstaltungen

(b) Die Speisepläne der Menschen

Abb. 3.8: Das Veranstaltungs- und das Mensamodul

3.4.6 Speisepläne

Auch für dieses Modul liegen die Daten im XML-Format vor und müssen dementsprechend geparsst werden, allerdings sind die Daten hier nicht wirklich modelliert: Empfangen wird eine Liste von Speisen, denen wiederum eine Reihe von Attributen zugeordnet sind - beispielsweise Preis, Inhaltsstoffe oder Ausgabetheke. Zur besseren Verarbeitung sowie Darstellung wurden neben der Klasse Meal³¹ noch die Klassen Canteen³² und Counter³³ entwickelt - alle dargestellt in Abbildung ???. Dabei sind in den empfangenen Daten den Speisen mehr Attribute zugewiesen, als in der Meal-Klasse angegeben: Hier werden nur die Eigenschaften in einen Meal-Objekt gespeichert, die auch tatsächlich angezeigt werden - eine Artikelnummer taucht beispielsweise gar nicht auf.

Beide Methoden der Klasse Canteen sind in Auszug 3.19 angegeben. Mit der einen wird ein Gericht der entsprechenden Ausgabe der Kantine zugeordnet und anschließend eine Liste aller Tage, an denen diese Kantine geöffnet ist (also Gerichte erhältlich sind), erstellt. Die andere gibt alle an dem angegeben Tag verfügbaren Speisen - gruppiert nach Ausgabe zurück. Hierfür wird eine Liste für die Rückgabe erstellt, anschließend über die in dieser Mensa enthaltenen Ausgabetheken iteriert und für alle generell dort ausgegebenen Gerichte geprüft, ob sie an dem übergebe-

³¹/models/meal.ts

³²/models/canteen.ts

³³/models/counter.ts

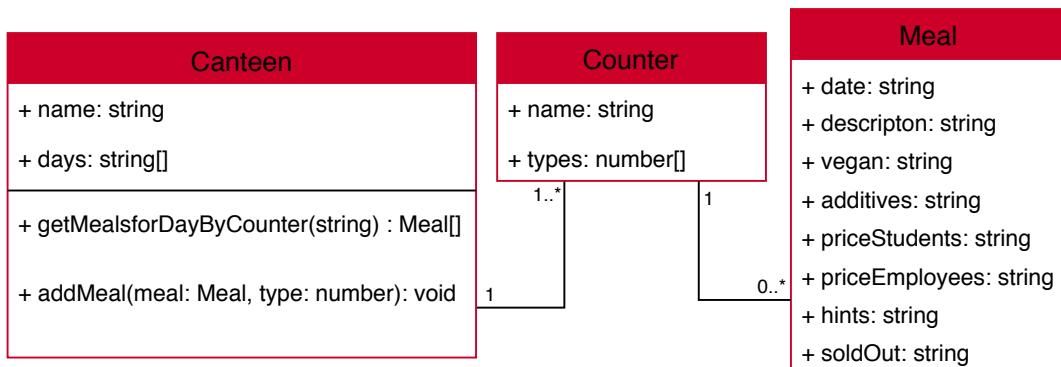


Abb. 3.9: UML-Diagramm der Klassen für das Mensamodul

Code-Ausschnitt 3.19: Auszug aus der Klasse Canteen

```

1  addMeal(meal: Meal) {
2      this.counters.forEach(counter => {
3          if (counter.types.indexOf(+meal.type) >= 0) {
4              counter.meals.push(meal);
5          }
6      });
7      if (this.days.indexOf(meal.date) < 0) {
8          this.days.push(meal.date);
9      }
10 }
11 getMealsForDayByCounter(day: string): Counter[] {
12     const result: Counter[] = [];
13     this.counters.forEach(counter => {
14         const temp: Counter = new Counter(counter.name);
15         counter.meals.forEach(meal => {
16             if (meal.date === day) {
17                 temp.meals.push(meal);
18             }
19         });
20         result.push(temp);
21     });
22     return result;
23 }

```

nen Tag erhältlich sind. Ist das der Fall, wird die jeweilige Speise zu einem neuen Counter-Objekt hinzugefügt. Wurde dies für jede Speise der Theke geprüft, wird diese der Liste für die Rückgabe hinzugefügt. Wurde so für alle Ausgaben verfahren, wird die Liste zurückgegeben.

Weiterhin wurden in einer .json-Datei³⁴ alle Menschen auf dem Campus aufgelistet, wobei zu diesen die darin befindlichen Ausgabeketten angegeben sind, welchen wiederum die Typen der an diesen Ausgaben erhältlichen Gerichte zugeordnet sind.

³⁴/canteen/canteens.json

Code-Ausschnitt 3.20: Ausschnitt aus CanteenComponent

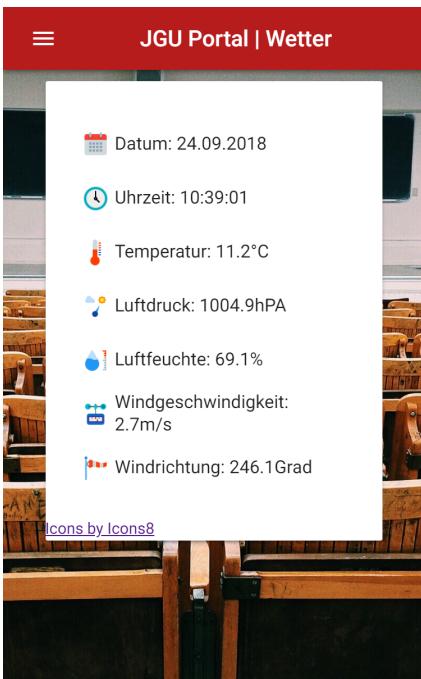
```
1  setActiveCanteen(event: MatTabChangeEvent) {
2      this.selectedCanteen = this.canteens[event.index];
3      if (this.selectedDay !== undefined) {
4          this.counters =
5              this.selectedCanteen.getMealsForDayByCounter(
6                  this.selectedDay);
7      }
8  }
9  handleSelect(event: MatOptionSelectionChange) {
10     if (!event.isUserInput) {
11         return;
12     }
13     if (this.selectedCanteen === undefined) {
14         this.selectedCanteen = this.canteens[0];
15     }
16     this.counters = this.selectedCanteen.getMealsForDayByCounter(
17         event.source.value);
18 }
```

Mit Hilfe dieser drei Klassen können die Daten dann weiterverarbeitet werden, wobei das Parsen im CanteenService³⁵ auch hier wieder im Grunde so funktioniert, wie in den bisher beschriebenen Fällen. Dazu nur noch ein paar Anmerkungen: Es werden logischerweise nur die Daten derjenigen Speisen weiterverarbeitet, die in einer Mensa auf dem Campus ausgegeben werden. Ebenso werden Daten von Speisen verworfen, die an vergangenen Tagen ausgegeben wurden - dies wird in einer eigenen Methode geprüft. Wurden die Daten eines Gerichts aus den XML-Daten ausgelesen, wird dieses - auch wieder in einer eigenen Funktion - in einer switch-Anweisung der entsprechenden Mensa zugeordnet, was über die bereits beschrieben addMeal-Methode der jeweiligen Kantine geschieht.

Die ngOnInit-Methode des CanteenComponent³⁶ sieht im Grunde genauso aus, wie die des vorhergehenden Moduls: Auch hier wird zunächst geprüft, ob die Daten bereits im Service als JSON vorliegen und nur, falls das nicht der Fall ist, die Daten vom Server bezogen. Außerdem befinden sich in dieser Komponente die zwei in Ausschnitt 3.20 wiedergegebenen Funktionen. Auch hier wurde ein Registerkartenlayout gewählt, um zwischen den einzelnen Kantinen zu wechseln. Wenn dies geschieht, wird die für dieses Ereignis als *EventListener* definierte Methode setActiveCanteen ausgeführt. In dieser wird zum einen die aktive Mensa neu gesetzt und zum anderen die Liste der Ausgabeketten der neu gewählten Mensa abgefragt, wenn bereits ein Tag aus der Drop-Down-Liste ausgewählt wurde. Das bedeutet, wenn ein Tag ausgewählt wurde und man die Mensa wechselt, werden dort die am gewählten Tag verfügbaren Gerichte (gruppiert nach zugehöriger Ausgabe) aufgelistet. Ist kein Tag ausgewählt worden, wird dort logischerweise nichts angezeigt.

³⁵ canteen/canteen.service.ts

³⁶ /canteen/canteen-component/canteen.component.ts



(a) Die Wetterdaten



(b) Öffnungszeiten von ZDV und Bibliotheken

Abb. 3.10: Wetterdaten und Öffnungszeiten verschiedener Einrichtungen

Die zweite Methode wird ausgeführt, wenn ein Datum aus aus der Liste ausgewählt wurde. Zunächst wird dort geprüft, ob dieses Event vom Nutzer ausgelöst wurde und anderenfalls verworfen. Da beim erstmaligen Initialisieren noch keine Mensa (aktiv) ausgewählt wurde, wird standardmäßig die erste in der Liste angezeigt und als aktiv gesetzt. Anschließend werden die für den gewählten Tag verfügbaren Gerichte der “aktiven” Mensa - gruppiert nach Ausgabetheke - abgefragt.

Im Template³⁷ werden die Theken in den Registerkarten der Kantinen als Akkordeon-Elemente dargestellt, wobei innerhalb dieser Akkordeon-Elemente die ausgegebenen Speisen als Karten aufgelistet sind. Abbildung 3.8b zeigt dieses Layout.

3.4.7 Wetter

Abgesehen davon, dass auch die Wetterdaten im XML-Format vorlagen und daher geparsst werden mussten, ist dieses Modul recht unkompliziert: Beim Initialisieren der Komponente³⁸ wird geprüft, ob die Wetterdaten bereits im WeatherService³⁹ gespeichert sind. Ist das der Fall, werden diese genutzt, ansonsten werden sie vom Server bezogen, geparsst, im Service gespeichert und dann ins Template eingesetzt. Außerdem werden natürlich auch hier die Titel der Seite und der Toolbar geändert. Abbildung 3.10a zeigt, wie dieses Modul aussieht.

³⁷ /canteen/canteen-component/canten.component.html

³⁸ /weather/weather-component/weather.component.html

³⁹ /weather/weather.service.ts

3.4.8 Öffnungszeiten

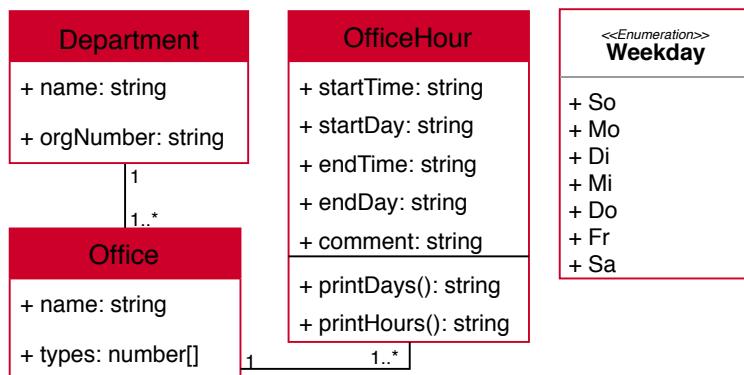


Abb. 3.11: UML-Diagramm der Klassen für die Öffnungszeiten

Da auch in diesem Modul die Daten über das UnivIS bezogen werden, sind sie ebenfalls in XML-Form und müssen geparsst werden. Für die weitere Verwendung wurden die in Abbildung 3.11 dargestellten Klassen implementiert. Das *Enum* Weekday wird dabei von keiner der anderen Klassen explizit verwendet, sondern dient nur dazu, zu einem Wochentag als Zahl den entsprechenden Wochentag als Kürzel zu erhalten. Ist beispielsweise in der Variablen day als Tag ‘1’ angegeben, kann das Kürzel ‘Mo’ durch Weekday[day] erhalten werden.

Die Liste der Department-Objekte⁴⁰ liegt lokal als JSON-Datei vor⁴¹, weil deren IDs benötigt werden, um die Öffnungszeiten (*officeHour*⁴²) der zugehörigen “Büros”⁴³ abfragen zu können. Momentan sind hier nur Daten für das ZDV sowie für die Bibliotheken hinterlegt, weil für weitere Einrichtungen (noch) keine Daten im UnivIS vorhanden sind. Sollten noch weitere Öffnungszeiten eingepflegt werden, können sie durch Erweitern dieser Datei recht einfach hinzugefügt werden. Beim Parsen der Daten ist hier darauf zu achten, dass neben den regulären Öffnungszeiten auch Ausnahmen angegeben sind - beispielsweise für die vorlesungsfreie Zeit. Abgesehen davon funktioniert das Übersetzen der Daten auch hier wieder wie zuvor, wobei sich des beschriebenen *Enum* bedient wurde, um aus den Wochentagen als Zahl das entsprechende Kürzel zu erhalten.

Um das Template etwas übersichtlicher zu halten, wurden in der Klasse *officeHour* die zwei in Ausschnitt 3.21 wiedergegebenen Methoden implementiert. Mit der ersten werden der erste und der letzte Tag, für den diese Öffnungszeit gültig ist, als String formatiert. Ist diese Öffnungszeit nur für einen Tag gültig, wird nur dieser ausgegeben. In der zweiten Methode geschieht analog das gleiche für die

⁴⁰/models/department.ts

⁴¹/office-hours/Departments.json

⁴²/models/officeHour.ts

⁴³/models/office.ts

Code-Ausschnitt 3.21: Ausschnitt aus der Klasse OfficeHour

```
1 let res = '';
2 if (officeHour.endDay !== undefined) {
3     res = officeHour.startDay + ' - ' + officeHour.endDay;
4 } else {
5     res += officeHour.startDay;
6 }
7 return res;
8 }
9
10 printHours(officeHour: OfficeHour): string {
11     return this.startTime + ' - ' + this.endTime + ' Uhr';
12 }
```

Code-Ausschnitt 3.22: Konfiguration für die Authentifizierung

```
1 import { AuthConfig } from 'angular-oauth2-oidc';
2
3 export const authConfig: AuthConfig = {
4     issuer: 'https://openid.uni-mainz.de',
5     redirectUri: window.location.origin + '/signin-oicd',
6     clientId: 'jgu.net_portal_website',
7     scope: 'openid',
8 };
```

entsprechenden Öffnungszeiten. In der Komponente passiert außer dem Abfragen der Daten sowie dem obligatorischen Ändern der Titel weiter nichts.

Für das Layout werden auch hier Registerkarten verwendet, die ja bereits vorgestellt wurden. In den Registerkarten sind dann für alle “Büros“ der jeweiligen Einrichtung Karten aufgelistet, in welchen deren Öffnungszeiten aufgelistet sind. In Abbildung 3.10b ist dargestellt wie dieses Modul aussieht.

3.4.9 Authentifizierung

Die Implementierung der Authentifizierung gestaltet sich dank des Pakets *angular-oauth2-oidc* recht unkompliziert. Nachdem dieses mit *npm* zugewiesen wurde, muss das enthaltene Modul zunächst im *AppModule* importiert werden (vgl Zeile 13, 3.1). Als nächstes muss im Wurzelverzeichnis eine Konfigurationsdatei⁴⁴ erstellt werden. In dieser muss - wie in Ausschnitt 3.22 wiedergegeben - neben dem *Identity-Provider*, der *clientId* und dem *scope* eine URL angegeben werden, zu der man nach erfolgreicher Authentifizierung weitergeleitet wird. Zuletzt muss das *AppComponent* um die in Ausschnitt 3.23 angegebene Methode erweitert werden.

In dieser Funktion wird zunächst die eben beschriebene Konfiguration geladen und der Authentifizierungsservice damit initialisiert. Nachdem diesem auch ein neuer

⁴⁴auth.config.ts

Code-Ausschnitt 3.23: Auslösen der Authentifizierung in AppComponent

```
1  private configureWithNewConfigApi() {
2      this.oauthService.configure(authConfig);
3      this.oauthService.tokenValidationHandler = new
4          JwksValidationHandler();
5  }
```

Code-Ausschnitt 3.24: Anfrage mit Header am Beispiel des Bibliotheksausweises

```
1  getLibraryId(): Observable<BibId> {
2      const headers = new HttpHeaders({
3          'Authorization': 'Bearer ' + this.oauthService.getAccessToken
4              ()
5      });
6      return this.http.get<BibId>(url, { headers: headers });
7  }
```

tokenValidationHandler initialisiert wurde, wird der eigentliche Authentifizierungsvorgang angestoßen. Wichtig ist noch, dass diese Methode im Konstruktor des AppComponent aufgerufen wird. Diese Konfiguration hat zur Folge, dass man direkt nach Seitenaufruf zum *Identity Provider* weitergeleitet wird. Es ist natürlich auch möglich dies über einen dedizierten Login-Button zu lösen, wobei dafür der Aufruf von loadDiscoveryDocumentAndLogin() aus der eben beschriebenen Methode entfernt und an entsprechender Stelle eingefügt werden muss.

Wie mit diesem Paket Anfragen getätigt werden können, die einer Authentifizierung bedürfen, wird im nächsten Kapitel erklärt.

3.4.10 Bibliotheksausweis

Da für den Bibliotheksausweis eine Authentifizierung notwendig ist, muss bei der Anfrage ein *Header* mitgeschickt werden. Wie dies funktioniert, ist in Ausschnitt 3.24 dargestellt. Soll eine ID angefragt werden, wird einer Header erzeugt, in welchen das vom *Identity Provider* erhältene *AccessToken* eingesetzt wird. Der Header kann dann dem *Request* - wie in Zeile 5 dargestellt - mitgegeben werden. Zusätzlich wird angegeben, dass ein Objekt der Klasse BibId⁴⁵ erwartet wird.

Hat man auf diese Weise eine ID erhalten, soll mit dieser ein Strichcode erzeugt werden. Mit dem hierfür verwendeten Paket *ngx-barcode* ist das so einfach, wie in Ausschnitt 3.25 dargestellt. Der Komponente *ngx-barcode* müssen lediglich der darzustellende Wert sowie dessen Format mittels *Attribute Binding* zugewiesen werden. Optional kann angegeben werden, dass der dargestellte Wert ebenfalls unter dem

⁴⁵/models/bibId.ts

Code-Ausschnitt 3.25: Template für den Bibliotheksausweis

```
1 <mat-card *ngIf="libraryId !== ''; else loading">
2   <mat-card-content>
3     
4     <ngx-barcode *ngIf="libraryId !== undefined; else noId"
5       [bc-value]="libraryId" [bc-format]="format" [bc-
       display-value]="true">
6   </ngx-barcode>
7   <ng-template #noId>
8     <div>Kein Bibliotheksausweis hinterlegt.</div>
9   </ng-template>
10  </mat-card-content>
11 </mat-card>
12 <ng-template #loading><mat-spinner></mat-spinner></ng-template>
```

Strichcode angezeigt werden soll. Ist dieser Wert undefiniert, weil kein Bibliotheksausweis hinterlegt ist, wird ein entsprechender Hinweis angezeigt. Außerdem wird eine Ladeanimation angezeigt, solange es sich libraryId noch um einen leeren String handelt, mit welchem dieses Attribut initialisiert wird. Wurde eine Antwort empfangen, enthält diese Variable entweder die ID, welche dann angezeigt wird, oder sie wird auf undefined gesetzt. In beiden Fällen wird dann zumindest die Grafik angezeigt.

3.4.11 PC-Pools

Die Daten für dieses Modul sind zweigeteilt: In einer lokalen JSON-Datei⁴⁶ sind die PC-Pools mit deren Ausstattung aufgelistet und die Kurse werden über eine API bezogen - ebenfalls als JSON. Die Daten sind in den Klassen, welche in Abbildung 3.13 abgebildet sind, modelliert. Strenggenommen ist für die Details der Räume keine eigene Klasse notwendig. Für die weitere Verarbeitung ist diese Kapselung jedoch von Vorteil. Warum das so ist lässt sich am Ausschnitt 3.26 aus dem PcPoolsService⁴⁷ erläutern.

In diesem ist zunächst die Methode aufgeführt, über welche die Daten von der API bezogen werden. Hierbei fällt auf, dass die Daten nicht als Array vom Type Course⁴⁸ empfangen werden. Stattdessen sind sie in Objekte gekapselt, in welchen der Name des zugehörigen PC-Pools hinterlegt ist. Der Einfachheit halber werden diese Objekte ebenfalls als ComputePool⁴⁹ bezeichnet - serverseitig gibt es ja keine Typisierung. Dadurch gibt es aber sowohl die lokale gespeicherte Liste der Räume mit computePoolDetails⁵⁰ und die von der API erhaltene Raumliste ohne Details.

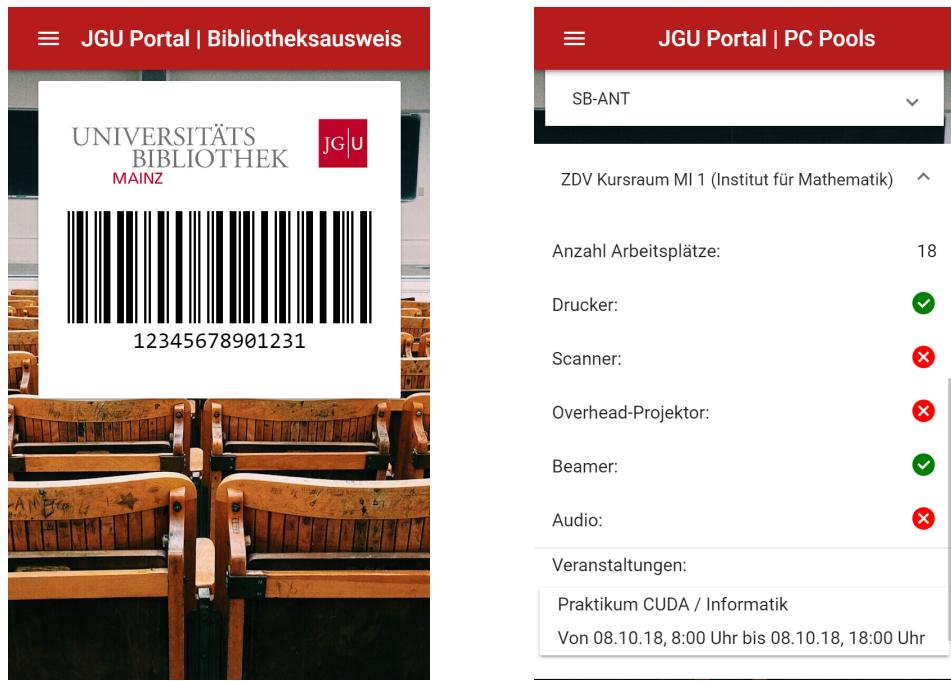
⁴⁶/pc-pools/computePools.json

⁴⁷/pc-pools/pc-pools.service.ts

⁴⁸/models/course.ts

⁴⁹/models/computePool.ts

⁵⁰/models/computePoolDetails.ts



(a)Exemplarischer Bibliotheksausweises

(b)Ansicht der PC-Pools

Abb. 3.12: Das Bibliotheksausweis- und das PC-Pool-Modul

Die “Verschmelzung“ beider Listen wird ausgeführt, indem vor der Weitergabe der Liste an das `PcPoolsComponent`⁵¹ ein *Mapping* in einer *Pipe* stattfindet (Zeile 5). In der durch das Mapping aufgerufenen Methode wird für jedes erhaltene `ComputePool`-Objekt das lokale gespeicherte Objekt mit demselben Raumnamen gesucht und die Kurse übertragen. Anschließend wird die, um die Veranstaltungen ergänzte, lokale Liste zurückgegeben. Abbildung 3.12b zeigt, wie die erhaltenen Daten dann angezeigt werden.

3.5 Umwandlung in eine Progessive Web App

Nachdem alle Inhalte eingebaut wurden, bleibt nur noch, die bis hierher entwickelte Web-App in eine Progressive Web App zu verwandeln. Mit Verwendung des Pakets `@angular/pwa` muss im Grunde lediglich der Befehl `ng add @angular/pwa` des Angular-CLI ausgeführt und dieses Paket somit installiert werden. Hierbei geschehen im Wesentlichen drei Dinge:

1. Im Wurzelverzeichnis des Projekts wird die Datei `ngsw-config.json` erstellt.
2. Im Verzeichnis `/src/` wird die Datei `manifest.json` erstellt.

⁵¹/pc-pools/pc-pools-component/pc-pools.component.ts

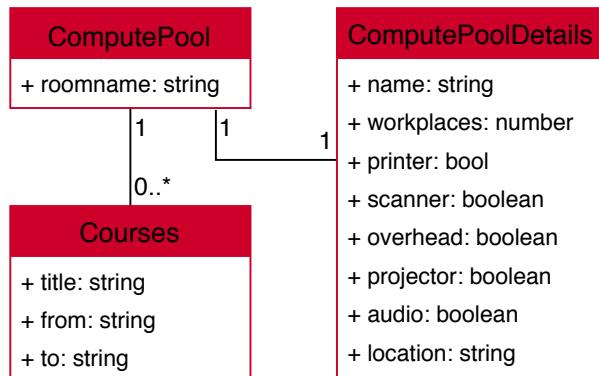


Abb. 3.13: UML-Diagramm der Klassen für das PC-Pool-Modul

Code-Ausschnitt 3.26: Ausschnitt aus PcPoolsService

```

1  getComputePools(): Observable<ComputePool[]> {
2      return this.http.get<ComputePool[]>(proxy + url)
3      .pipe(
4          map(computePools => this.addDetailsToComputePools(
5              computePools))
6      );
7  }
8  private addDetailsToComputePools(computePools: ComputePool[]): void {
9      this.pcPools.forEach(pool => {
10         computePools.forEach(computePool => {
11             if (pool.roomName === computePool.roomName) {
12                 pool.events = computePool.events;
13             }
14         });
15     return this.pcPools;
16 }
  
```

3. Im AppModule wird das ServiceWorker-Modul importiert sowie ein entsprechendes Skript registriert (vgl. Ausschnitt 3.1, Zeile 14).

Damit erhält man - nach Abschluss des Build-Prozesses mittels `ng build` - im Grunde schon eine den Spezifikationen entsprechende *Progressive Web App* - vorausgesetzt, sie wird via HTTPS ausgeliefert. Allerdings sollten die automatisch im Verzeichnis `/src/assets/icons` erzeugten Icons ersetzt, die Eigenschaften der Web-App in `manifest.json` angepasst, und in `ngsw-config.json` die im Cache zu speichernden Ressourcen definiert werden. Im Falle dieses Projekts wurde die Manifest-Datei - wie in Ausschnitt 3.27 dargestellt - angepasst. Dabei werden als Haupt- bzw. Hintergrundfarbe die Rot- und Grau-Töne des JGU-Corporate-Design gewählt und die Icons durch das JGU-Logo ersetzt - die vorgegebenen Pfade bleiben unverändert. Außerdem wurde ein kurzer und ein langer Titel gesetzt, sowie angegeben, dass die Seite als eigenständige Anwendung ausgeführt werden soll, wenn der Nutzer dies wünscht. Die Parameter `scope` und `start_url` bleiben unverändert, weil es sich hier um eine Single-Page Application handelt, sodass logischerweise alle Unterseiten

Code-Ausschnitt 3.27: Ausschnitt aus `manifest.json`

```
1  {
2    "name": "Studierendenportal JGU Mainz",
3    "short_name": "JGU Portal",
4    "theme_color": "#b71c1c",
5    "background_color": "#fafafa",
6    "display": "standalone",
7    "scope": "/",
8    "start_url": "/",
9    "icons": [
10      /* */
11    ]
12 }
```

Teil der Anwendung sind und beim Öffnen der Anwendung immer zum Wurzelpfad navigiert werden soll.

In der Konfigurationsdatei für den Service Worker bleiben die vordefinierten Parameter für statischen Assets - also für Bilder, Stylesheets und Skripte sowie für das HTML-Dokument - unverändert. Sollten weitere solcher Dateien in einem anderen Ordner hinzugefügt werden, so müssen keine vollständigen Dateipfade angegeben werden. Es reicht, ein Verzeichnis hinzuzufügen sowie mit einem oder zwei Sternchen ("*") zu definieren, ob nur der Inhalt des Verzeichnisses selbst oder auch alle weiteren Unterverzeichnisse gecachet werden soll(en). Beim Build-Prozess werden dann automatisch für alle so definierten Dateien die Pfade generiert und hinzugefügt.

Zuletzt können noch `dataGroups[@43]` definiert werden. Dabei wird für eine Liste von URLs angegeben, nach welchen Kriterien diese im Cache gespeichert bzw. vom Service Worker verarbeitet werden sollen. In diesem Beispiel wird mit `maxSize` angegeben, dass von den über diese beiden URLs bezogenen Ressourcen nur zwei Einträge gespeichert werden sollen, die außerdem höchstens 24 Stunden lang gespeichert werden. Weiterhin wird definiert, dass zunächst versucht werden soll, eine aktuelle Version der Daten von der entsprechenden Quelle zu beziehen. Erst wenn nach 30 Sekunden - dem definierten "Timeout" - keine Antwort erhalten wurde soll auf eine im Cache gespeichert Version zurückgegriffen werden. Alternativ kann als Strategie auch "performance" gewählt werden, wodurch primär versucht wird, auf gespeicherte Datensätze zurückzugreifen. Auf gleiche Weise wird für alle anderen URLs verfahren, wobei natürlich auch anderer Optionen gewählt wurden.

Code-Ausschnitt 3.28: Exemplarische Konfiguration des Service Worker

```
1 "dataGroups": [
2   {
3     "name": "news",
4     "urls": [
5       "https://www.uni-mainz.de/32.php",
6       "https://www.zdv.uni-mainz.de/feed/",
7     ],
8     "cacheConfig": {
9       "maxSize": 2,
10      "maxAge": "24h",
11      "strategy": "freshness",
12      "timeout": "30s"
13    }
14  }, /* . . . */
15 ]
```

Zusammenfassung

„A program is never less than 90% complete and never more than 95% complete“

— Terry Baker

Dieses Kapitel fasst abschließend die Inhalte der Arbeit zusammen, gibt eine Übersicht darüber, welche Ziele erreicht wurden und welche nicht und gibt dann Ausblick darüber, welche Verbesserungen und Überarbeitungen möglich, nötig oder empfehlenswert sind.

4.1 Rückblick

In dieser Arbeit wurde mit dem Angular-Framework eine Single-Page Application (SPA) entwickelt, welche dann in eine Progressive Web App (PWA) „umgewandelt“ wurde, sodass diese Web-App auf kompatiblen Geräten bzw. in kompatiblen Browzern (fast) genauso wie eine native Anwendung installiert und genutzt werden kann. Dabei wurde zunächst erläutert, welche Idee hinter PWAs steckt und was genau eine solche ausmacht. Anschließend wurden Alternativen zu Angular vorgestellt sowie Gemeinsamkeiten und Unterschiede herausgearbeitet. Basierend darauf wurde dann begründet, warum sich Angular von den vorgestellten Libraries bzw. Frameworks am besten für dieses Projekt geeignet hat und außerdem dieses Framework ausführlich beschrieben. Vor der Beschreibung der eigentlichen Programmierung wurden besondere Hindernisse angeführt, die es bei einem solchen Projekt - nicht nur beim Einsatz von Angular, sondern auch von äquivalenten Alternativen - zu überwinden gilt.

Im Wesentlichen wurden fast alle gewünschten Inhalte, die so auch in der “Uni Mainz“-App zu finden sind, in die Web-App eingebaut. Es fehlt lediglich die Statusübersicht über die Dienste des ZDV (E-Mail, Remote-Desktop, Druckdienste,...). Außerdem funktioniert das Anzeigen der Kontaktdaten der Ansprechperson einer Veranstaltung nicht, wie gewollt. Geplant war, dass einer jeden Veranstaltung ein Link zugeordnet wird, über den man zum Personensuche-Modul gelangt, in welchem dann die entsprechenden Informationen der Kontaktperson angezeigt werden sollen. Da für die Abfrage der Daten einer Person über ihre ID aber ein key benötigt wird, der

sich in regelmäßigen Abständen ändert, ist dieser Ansatz so nicht praktikabel. Eine Lösung könnte darin bestehen, die bei der Auflistung der Veranstaltungen erhaltenen Informationen über die Kontaktperson an das Personensuche-Modul weiterzuleiten, statt über die ID eine erneute Suche anzustoßen.

Ansonsten sind es Details, die an der ein- oder anderen Stelle ergänzt werden könnten. So wäre es beispielsweise möglich, im Mensamodul Inhaltsstoffe und Allergene der Speisen anzuführen.

Das Umwandeln der Web-App in eine *Progressive Web App* hingegen - was ja eine Hauptanforderung an dieses Projekt darstellte - war durch das Paket `@angular/pwa` einfach und unkompliziert. Die größte Schwierigkeit bestand hauptsächlich darin, die in XML-Form vorliegenden Daten in JSON-Form zu "übersetzen".

4.2 Ausblick

Zu den Inhalten und Funktionen, die noch umgesetzt werden könnten, gehört beispielsweise, dass die Bus- und Straßenbahnhaltestellen nicht auf der Karte markiert werden. Hierauf wurde zunächst verzichtet, weil in dem verwendeten Kartenmaterial bereits Haltestellen eingezeichnet sind. Dennoch wäre es möglich, bei jeder Haltestelle im Busfahrplan-Modul einen Link zu hinterlegen, der zum Kartenmodul weiterleitet, wo dann die Haltestelle auf der Karte markiert würde. Das Busfahrplan-Modul könnte außerdem dahingehend erweitert werden, dass es möglich ist, zu einer Bus- oder Straßenbahnlinie auch den Fahrverlauf abzufragen.

Weiterhin wäre es denkbar, die Meldungen im Nachrichten-Modul nach Kategorien zu sortieren bzw. dem Nutzer eine Möglichkeit zu bieten, selbst zu entscheiden, welche Kategorien angezeigt werden sollen.

Ebenfalls könnte beispielsweise im Bibliotheksausweis-Modul ein dedizierter Login-Button platziert werden, damit man nicht direkt bei Seitenaufruf weitergeleitet wird. Auf die aktuell umgesetzte Weise ist die Web-App offline quasi nicht möglich, da ohne Internetverbindung logischerweise auch keine externe Authentifizierung möglich ist.

Ein weiterer verbesserungsfähiger Punkt ist das Design bzw. das Layout mancher Module. Es wurde zwar stets versucht, ein für alle Bildschirmgrößen praktikables Layout zu finden, auf Grund der unterschiedlich gearteten Daten war dies aber teilweise nicht ganz einfach. Bei den Veranstaltungen beispielsweise kann die Beschreibung einen einzigen Satz umfassen - aber auch einen ganzen Absatz. Hier ein Layout zu finden, dass beiden Szenarien gerecht wird und nicht in dem einen Fall

zu viel und im anderen zu wenig Platz beansprucht - und zwar auf großen, wie auf kleinen Displays - ist mitunter nicht ganz einfach. Das liegt aber genauso wenig an Angular selbst, wie es bei den nicht ganz optimal organisierten Stylesheets der Fall ist.

Obwohl Angular “testgetriebenes Programmieren“ unterstützt bzw. ermöglicht, wurde hierauf zugegebenermaßen verzichtet; es könnten dementsprechend noch ein paar Testfälle implementiert werden um sicherzustellen, dass alle Randfälle bedacht wurden. Genauso wäre an ein paar Stellen besseres “Error-Handling“ - also das Behandeln von *Exceptions* - empfehlenswert. Zwar wurde vor allem beim Parsen der XML-Daten darauf geachtet, (möglichst) alle Spezialfälle abzudecken, es ist aber durchaus möglich, dass hier Fehler durch nicht bedachte Szenarien entstehen.

Literaturverzeichnis

[40]MONSUR HOSSAIN. *CORS in Action - Creating and consuming cross-origin APIs.* Manning Publications, 2014 (siehe S. 35).

Webseiten

- [@1]*Visual Studio-Tools für Xamarin.* 2018. URL: <https://visualstudio.microsoft.com/de/xamarin/> (besucht am 3. Okt. 2018) (siehe S. 6).
- [@2]*Build amazing mobile apps powered by open web tech.* 2018. URL: <https://phonegap.com/> (besucht am 3. Okt. 2018) (siehe S. 6).
- [@3]Alex Russell. *What, Exactly, Makes Something A Progressive Web App?* 2016. URL: <https://infrequently.org/2016/09/what-exactly-makes-something-a-progressive-web-app/> (besucht am 25. Aug. 2018) (siehe S. 6).
- [@4]Paul Kinlan Matt Gaunt. *The Web App Manifest.* URL: <https://developers.google.com/web/fundamentals/web-app-manifest/> (besucht am 25. Aug. 2018) (siehe S. 7).
- [@5]Matt Gaunt. *Service Workers: an Introduction.* URL: <https://developers.google.com/web/fundamentals/primers/service-workers/> (besucht am 25. Aug. 2018) (siehe S. 8).
- [@6]Jake Archibald. *Introducing Background Sync.* 2015. URL: <https://developers.google.com/web/updates/2015/12/background-sync> (besucht am 25. Aug. 2018) (siehe S. 8).
- [@7]Muriel Santoni. *Progressive Web Apps : Feature compatibility based on the browser.* 2018. URL: <https://www.goodbarber.com/blog/progressive-web-apps-feature-compatibility-based-on-the-browser-a883/> (besucht am 25. Aug. 2018) (siehe S. 9).
- [@8]Andreas Domin. *Library vs. Framework: Das sind die Unterschiede.* 2018. URL: <https://t3n.de/news/library-vs-framework-unterschiede-1022753/> (besucht am 30. Sep. 2018) (siehe S. 9).
- [@9]Jens Neuhaus. *Angular vs. React vs. Vue: A 2017 comparison.* 2017. URL: <https://medium.com/unicorn-supplies/angular-vs-react-vs-vue-a-2017-comparison-c5c52d620176> (besucht am 30. Sep. 2018) (siehe S. 9).

- [@10]Stephen Fluin. *Branding Guidelines for Angular and AngularJS*. 2017. URL: <https:////blog.angularjs.org/2017/01/branding-guidelines-for-angular-and.html> (besucht am 6. Sep. 2018) (siehe S. 10).
- [@11]Stephen Fluin. *Version 6 of Angular Now Available*. 2018. URL: <https://blog.angular.io/version-6-of-angular-now-available-cc56b0efa7a4> (besucht am 7. Sep. 2018) (siehe S. 10).
- [@12]*Introducing JSX*. 2018. URL: <https://reactjs.org/docs/introducing-jsx.html> (besucht am 29. Sep. 2018) (siehe S. 11).
- [@13]Dan Abramov. *Create Apps with No Configuration*. 2016. URL: <https://reactjs.org/blog/2016/07/22/create-apps-with-no-configuration.html> (besucht am 29. Sep. 2018) (siehe S. 11).
- [@14]*Add React to a Website*. 2018. URL: <https://reactjs.org/docs/add-react-to-a-website.html> (besucht am 29. Sep. 2018) (siehe S. 12).
- [@15]*Virtual DOM and Internals*. 2018. URL: <https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom> (besucht am 25. Sep. 2018) (siehe S. 12).
- [@16]*Components and Props*. 2018. URL: <https://reactjs.org/docs/components-and-props.html#rendering-a-component> (besucht am 29. Sep. 2018) (siehe S. 12).
- [@17]Evan Yu. *FIRST WEEK OF LAUNCHING VUE.JS*. 2014. URL: <http://blog.evanyou.me/2014/02/11/first-week-of-launching-an-oss-project/> (besucht am 29. Sep. 2018) (siehe S. 13).
- [@18]*Components Basics*. 2018. URL: <https://vuejs.org/v2/guide/components.html> (besucht am 29. Sep. 2018) (siehe S. 13).
- [@19]*Single File Components*. 2018. URL: <https://vuejs.org/v2/guide/single-file-components.html> (besucht am 29. Sep. 2018) (siehe S. 14).
- [@20]*Introduction to components*. 2018. URL: <https://angular.io/guide/architecture-components> (besucht am 7. Sep. 2018) (siehe S. 17).
- [@21]*Angular - Displaying Data*. 2018. URL: <https://angular.io/guide/displaying-data> (besucht am 7. Sep. 2018) (siehe S. 19).
- [@22]*Introduction to modules*. 2018. URL: <https://angular.io/guide/architecture-modules> (besucht am 7. Sep. 2018) (siehe S. 21).
- [@23]*Introduction to services and dependency injection*. 2018. URL: <https://angular.io/guide/architecture-services> (besucht am 7. Sep. 2018) (siehe S. 21).
- [@24]Chinmaya Champatiray. *Dependency Inversion Principle, IoC Container, and Dependency Injection*. 2014. URL: <https://www.codeproject.com/Articles/465173/Dependency-Inversion-Principle-IoC-Container-Depen> (besucht am 3. Okt. 2018) (siehe S. 21).
- [@25]*Structural Directives*. 2018. URL: <https://angular.io/guide/structural-directives> (besucht am 7. Sep. 2018) (siehe S. 22).
- [@26]*Attribute Directives*. 2018. URL: <https://angular.io/guide/attribute-directives> (besucht am 7. Sep. 2018) (siehe S. 22).
- [@27]*Pipes*. 2018. URL: <https://angular.io/guide/pipes> (besucht am 7. Sep. 2018) (siehe S. 22).

- [@28] *Data binding*. 2018. URL: <https://angular.io/guide/architecture-components#data-binding> (besucht am 30. Sep. 2018) (siehe S. 24).
- [@29] *The RxJS library*. 2018. URL: <https://angular.io/guide/rx-library> (besucht am 29. Sep. 2018) (siehe S. 24).
- [@30] *Observables in Angular*. 2018. URL: <https://angular.io/guide/observables-in-angular> (besucht am 29. Sep. 2018) (siehe S. 24).
- [@31] *Observables compared to other techniques*. 2018. URL: <https://angular.io/guide/comparing-observables#observables-compared-to-events-api> (besucht am 29. Sep. 2018) (siehe S. 25).
- [@32] Mike Wasson. *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET*. 2013. URL: <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx> (besucht am 30. Sep. 2018) (siehe S. 26).
- [@33] *Add the Router Outlet*. 2018. URL: <https://angular.io/guide/router#add-the-router-outlet> (besucht am 30. Sep. 2018) (siehe S. 27).
- [@34] *Architecture overview*. 2018. URL: <https://angular.io/guide/architecture#whats-next> (besucht am 30. Sep. 2018) (siehe S. 28).
- [@35] *Basic Types*. 2018. URL: <https://www.typescriptlang.org/docs/handbook/basic-types.html> (besucht am 7. Sep. 2018) (siehe S. 30).
- [@36] *TypeScript*. 2018. URL: <https://www.typescriptlang.org/> (besucht am 7. Sep. 2018) (siehe S. 30).
- [@37] *OpenLayers*. 2018. URL: <https://openlayers.org/> (besucht am 7. Sep. 2018) (siehe S. 30).
- [@38] Manfred Steyer. *angular-oauth2-oidc*. 2018. URL: <https://www.npmjs.com/package/angular-oauth2-oidc> (besucht am 7. Sep. 2018) (siehe S. 31).
- [@39] Bryon Williams. *ngx-barcode*. 2018. URL: <https://www.npmjs.com/package/ngx-barcode> (besucht am 7. Sep. 2018) (siehe S. 31).
- [@41] *Cross-Origin Resource Sharing (CORS)*. 2018. URL: <https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/cors.html> (besucht am 3. Okt. 2018) (siehe S. 36).
- [@42] *Providing services in modules vs. components*. 2018. URL: <https://angular.io/guide/providers#providing-services-in-modules-vs-components> (besucht am 6. Okt. 2018) (siehe S. 41).
- [@43] *Service worker configuration*. 2018. URL: <https://angular.io/guide/service-worker-config> (besucht am 8. Okt. 2018) (siehe S. 69).

Abkürzungsverzeichnis

HTML Hypertext Markup Language

XML Extensive Markup Language

CSS Cascading Stylesheets

PWA Progressive Web App

JSON JavaScript Object Notation

SPA Single-Page Application

CLI Command-Line Interface

JSX JavaScript XML

HTTP Hypertext Transfer Protocol

API Application Programming Interface

DOM Document Object Model

FQDN Fully Qualified Domain Name

SOP Same-Origin Policy

CORS Cross-Origin Ressource Sharing

URL Uniform Ressource Locator

Abbildungsverzeichnis

2.1	Flussrichtung der Daten bei <i>Data-Binding</i>	24
2.2	Vergleich der Funktionsweise von herkömmlichen Seiten mit <i>Single-Page Applications</i>	26
2.3	Übersicht über die Architektur des Angular-Frameworks	28
3.1	Veranschaulichung der <i>Same-Origin Policy</i>	35
3.2	Veranschaulichung von <i>Cross-Origin Ressource Sharing (CORS)</i>	36
3.3	Grundgerüst der Web-App	40
3.4	Das Nachrichten - und das Kartenmodul	46
3.5	UML-Diagramm der Klassen des Busfahrplan-Moduls	50
3.6	Das Abfahrtszeiten- und das Personensuchmodul	52
3.7	UML-Diagramm der Klassen für die Personensuche und das Veranstaltungsmodul	56
3.8	Das Veranstaltungs- und das Mensamodul	59
3.9	UML-Diagramm der Klassen für das Mensamoduls	60
3.10	Wetterdaten und Öffnungszeiten verschiedener Einrichtungen	62
3.11	UML-Diagramm der Klassen für die Öffnungszeiten	63
3.12	Das Bibliotheksausweis- und das PC-Pool-Modul	67
3.13	UML-Diagramm der Klassen für das PC-Pool-Modul	68

Colophon

This thesis was typeset with $\text{\LaTeX}2\epsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at <http://cleanthesis.der-ric.de/>.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, 15. Oktober 2018

Marcel Bastian

