FH Vorarlberg
University of Applied Sciences

TBD

Bachelorarbeit II
zur Erlangung des akademischen Grades

Bachelor of Science in Engineering (BSc)

Fachhochschule Vorarlberg
Informatik – Software and Information Engineering

Betreut von
Prof. (FH) Dipl. Inform. Thomas Feilhauer

Vorgelegt von
Dominic Luidold
Dornbirn, 20. Mai 2021

Widmung

TODO

„TODO“
TODO

Kurzreferat

TODO

TODO

Abstract

TODO

TODO

Geschlechtergerechte Sprache

Der Verfasser der vorliegenden Arbeit bekennt sich zu einer geschlechtergerechten Sprachverwendung.

Um diese Arbeit sowohl geschlechtergerecht als auch -inklusive zu formulieren, wird explizit auf fix männlich oder weiblich zugeordnete Personengruppen, das sogenannte Binnen-I oder andere Schreibweisen verzichtet. Stattdessen wird auf die Schreibweise mit einem Doppelpunkt (beispielsweise „Anwender:innen“, „Entwickler:innen“ etc.) gesetzt, die alle Personengruppen einschließt und dazu beiträgt, eine Bewusstheit für bestehende, diskriminierende Sprachgewohnheiten gegenüber Frauen sowie queeren Mitmenschen zu schaffen beziehungsweise zu stärken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CSS Cascading Style Sheets

DOM Document Object Model

JSON JavaScript Object Notation

LTS Long Term Support

MPA Multi-page Application

PWA Progressive Web App

SSR Server-side rendering

SPA Single-page Application

UI User Interface

UX User Experience

1. Einleitung

Diese Bachelorarbeit verfolgt das Ziel, einen Einblick in die Single-page Application (SPA) Frameworks *Angular*[1] und *Vaadin*[2] zu geben und deren Gemeinsamkeiten, Unterschiede sowie Vor- und Nachteile zu beleuchten.

Um ein grundlegendes Verständnis über die Thematik von SPAs zu erlangen, wird zu Beginn der Arbeit auf das Konzept einer SPA eingegangen und die zugrundeliegende Herangehensweise mit der einer klassischen Multi-page Application (MPA) verglichen. Im weiteren Verlauf werden die unterschiedlichen Ansätze von Angular und Vaadin genauer betrachtet und eine tatsächliche Umsetzung der zuvor erläuterten Technologien mittels zweier Demo-Applikationen getestet. Am Ende dieser Arbeit wir darauf eingegangen, ob sich - anhand unterschiedlicher Kriterien und Anwendungsfälle - eine Empfehlung für eines der beiden SPA Frameworks aussprechen lässt.

1.1. Motivation

In den letzten Jahren lässt sich beobachten, dass Webapplikationen, Apps und Anwendungen allgemein verstärkt mittels des SPA-Ansatzes umgesetzt werden und somit auf eine deutlich unterschiedlichere Herangehensweise - im Gegensatz zu klassischeren MPAs - setzen. (Ismail 2019) Für die Umsetzung einer solchen Applikation stehen eine Vielzahl von Frameworks zur Verfügung, die darüber hinaus weitere Features bieten und Entwickler:innen bei der Umsetzung unterstützen.

[1] Angular (https://angular.io)
[2] Vaadin (https://vaadin.com)

Die richtige Wahl des Frameworks, der jeweiligen Technologien und der im Hintergrund agierenden Strukturen spielen eine wesentliche Rolle bei der Planung und Umsetzung eines neuen Projektes. Welches Framework sich besser eignet, lässt sich oftmals nicht auf den ersten (oder sogar zweiten) Blick feststellen. Diese Arbeit befasst sich daher genauer mit dem Konzept von SPAs und vergleicht zwei darauf aufbauende Frameworks, die mit deutlich unterschiedlichen Technologie-Stacks arbeiten und zu vergleichbaren Lösungen führen.

1.2. Problemstellung

Die in Abschnitt 1.1 angesprochene Vielzahl an SPA-Frameworks bietet grundlegend den Vorteil, dass eine große Auswahlmöglichkeit und eine gewisse Konkurrenz untereinander zu einem hohen Qualitätsstandard führt. Zudem wird dadurch sichergestellt, dass es für jedes Projekt - unabhängig von den jeweiligen Anforderungen und etwaigen Eigenheiten - eine Möglichkeit gibt, dieses mit einem der verfügbaren Frameworks umzusetzen. Auf der anderen Seite führt die stetig wachsende Anzahl an Möglichkeiten - vor allem von solchen, die auf JavaScript basieren - jedoch dazu, dass sich meist nur schwer beurteilen lässt, welches Framework und welche zugrundeliegende Technologie sich für die Umsetzung einer Applikation bestmöglich eignet.

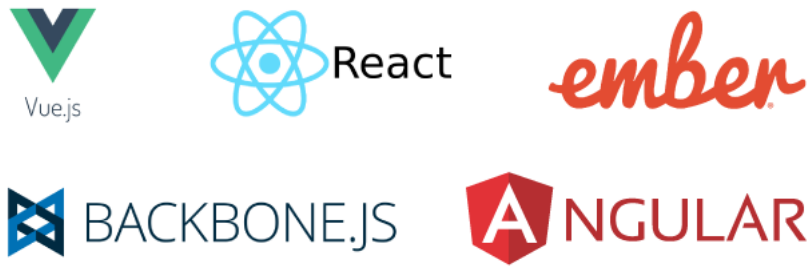

Abbildung 1.1.: Liste möglicher JavaScript-Frameworks zur Umsetzung von SPAs (Quelle: A. 2020)

Um eine geeignete Wahl eines Frameworks treffen zu können, sollten vorab Kriterien und Anforderungen definiert werden, die schlussendlich erfüllt wer-

den müssen. Neben grundlegenden Funktionalitäten, die in den meisten Fällen von einer Vielzahl der Frameworks abgedeckt werden können, stellen die projektspezifischen Eigenheiten und vor allem die Auswahl der zugrundeliegenden Technologien (beispielsweise *JavaScript* oder auch *Java*) eine der wichtigsten Herausforderungen dar. Diese Entscheidung muss gut überlegt und abgewogen werden, da diese im weiteren Verlauf weitreichende Folgen bei der Umsetzung einer (Web-)Applikation zur Folge hat und sich ein Wechsel nach gestarteter Entwicklung nur unter großem Aufwand umsetzen lässt.

Die in Abbildung 1.1 auf Seite 2 dargestellten Frameworks zeigen eine Auswahl an Frameworks auf, die auf *JavaScript* aufbauen beziehungsweise basieren und somit primär auf dem Client, dem Browser, eingesetzt werden können. SPAs lassen sich jedoch nicht nur Frontend-seitig und vollständig mit JavaScript entwickeln (bei denen ein Großteil der Logik auf einem externen Server abläuft), sondern können auch gänzlich mittels auf Java basierenden Frameworks umgesetzt werden. Bei diesen Frameworks, zu denen Vaadin gehört, lässt sich sowohl die Logik als auch das User Interface (UI) in einem einheitlichen Projekt kombinieren und entwickeln.

Da die unterschiedlichen Ansätze und Technologie-Stacks, sowohl hinter dem auf Java basierenden Vaadin als auch auf dem auf TypeScript aufbauenden Angular, gewisse Vor- und Nachteile sowie Tücken mit sich bringen, fällt die Wahl auf eines der beiden SPA-fähigen Frameworks auf den ersten Blick nicht leicht. Hinzu kommt die Frage, welches der Frameworks weiterführende Funktionalitäten bietet, um mit geringem Aufwand beispielsweise eine Progressive Web App (PWA) umzusetzen oder anwendungsspezifische Daten lokal sowie extern persistieren zu können.

1.3. Zielsetzung

Die in den Abschnitten 1.1 und 1.2 angeführten Punkte haben aufgezeigt, dass die große Anzahl an Frameworks, mit denen SPAs umgesetzt werden können, zwar sehr positiv einzuschätzen ist, die damit verbundenen Probleme bei der Auswahl des richtigen Frameworks werden dadurch jedoch verstärkt. Aufgrund der unterschiedlichen zugrundeliegenden Technologien und einhergehenden Her-

angehensweisen ist eine bedachte Wahl wichtig.

Diese Arbeit verfolgt daher das Ziel, das JavaScript/TypeScript Framework *Angular* dem auf Java basierenden Framework *Vaadin* gegenüberzustellen und zu vergleichen. Das Ziel ist es, mittels Literatur belegter Vergleiche einen allgemeinen Überblick über SPAs zu geben, diese klassischen Ansätzen gegenüberzustellen und zwei Demo-Applikationen zu entwickeln. Diese Webanwendungen werden dann herangezogen, um anhand von vorab definierten Kriterien feststellen zu können, ob und in wie weit Empfehlungen für eines der beiden Frameworks ausgesprochen werden können.

Um den Fokus dieser Arbeit genauer zu definieren und einzuschränken, wird die Planung, Umsetzung sowie abschließende Beurteilung der Applikationen anhand der ausgearbeiteten Kriterien auf folgende Punkte beschränkt:

- Möglichkeit zur einfachen Umsetzung einer Progressive Web App (PWA)
- Möglichkeit der Wiederverwendbarkeit von Komponenten, gegebenenfalls mittels *Web Components*
- Möglichkeit Daten lokal (Browser) sowie extern (Server) zu persistieren

2. Stand der Technik

Das folgende Kapitel gibt einen Überblick über die Funktionsweise einer SPA und vergleicht das Konzept von SPAs mit klassischen MPAs. Im Anschluss wird im Detail auf die Funktionsweise von Vaadin und Angular beziehungsweise deren unterschiedlichen Ansätze in Hinblick auf die Entwicklung der UI mittels JavaScript und Java eingegangen. Im weiteren Verlauf werden die damit verbundenen Vor- und Nachteile genauer beleuchtet.

2.1. Konzept einer SPA

Eine klassische MPA basiert auf dem Konzept, dass bei jedem Aufruf eines neuen View beziehungsweise einer HTML-Seite eine Anfrage an den Server gestellt wird. Dieser verarbeitet die Anfrage und retourniert das jeweils neu zusammengestellte Resultat der Präsentations- sowie der darunterliegenden Schichten an den Client. Eine SPA ist hingegen eine Webanwendung, bei der die Präsentationsschicht und die damit verbundene Logik vom Server entkoppelt und vollständig in den Client, sprich den Browser, ausgelagert wird. (Scott 2015, Seite 5ff.)

Die Abbildung 2.1 auf Seite 6 stellt den Aufbau solch einer SPA schematisch dar und verdeutlicht, dass die Darstellung der mittels AJAX und XHR angeforderten Daten komplett vom Client übernommen wird. Serverseitig wird lediglich ein *Controller* benötigt, welcher die übermittelten Daten in Objekte umwandeln kann, die für die Logik entsprechend verständlich sind.

Diese Herangehensweise führt dazu, dass beim Aufrufen einer (Unter-)Seite mittels clientseitigem Routing keine komplett neue HTML-Seite geladen werden muss, sondern lediglich Teile des User Interface - sogennante *Views* -

ausgetauscht werden. Hierfür wird das entsprechende Document Object Model (DOM) mittels JavaScript dynamisch ausgetauscht und die benötigten Daten werden bei Bedarf asynchron mittels AJAX und XHR vom Server geladen. (Scott 2015, Seite 7)

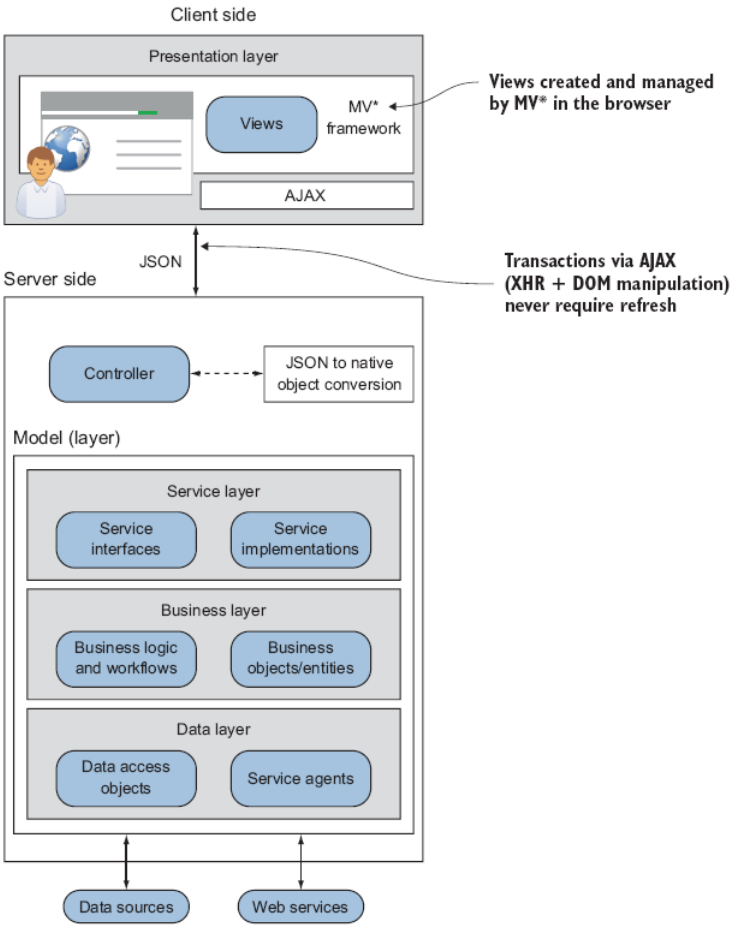

Abbildung 2.1.: Aufbau einer SPA
(Quelle: Scott 2015, Seite 6)

2.1.1. Server-side rendering

Neben der reinen Übertragung von Daten mittels JSON (oder anderweitigen Datenformaten) kann bei SPAs alternativ beziehungsweise erweiternd auch auf Server-side rendering (SSR) gesetzt werden. Bei diesem Ansatz werden Aus-

schnitte von HTML bereits auf dem Server vorbereitet und zusammen mit weiterführenden Daten an den Client geschickt. Dieser kann somit einen Teil der Antwort ohne weitere Aufbereitung darstellen, während die restlichen Daten mittels DOM-Manipulation in die View eingebettet werden. (Scott 2015, Seite 7)

2.1.2. Bestandteile einer SPA

Der zugrundeliegende Aufbau einer SPA - und der Bestandteil der Applikation, der lediglich einmal geladen wird - ist die sogenannte *Shell*. Die Shell ist eine einzelne HTML-Datei, welche vom Browser vollständig geladen wird und in den meisten Fällen lediglich minimale Strukturen (ein Navigationsmenü, statische Inhalte etc.) sowie einen leeren `DIV` Tag enthält, wie Abbildung 2.2 auf Seite 6 zeigt. Genutzt wird diese als Ausgangspunkt für alle weiteren Views, die unabhängig von der Shell agieren und dynamisch geladen werden. (Scott 2015, Seite 8)

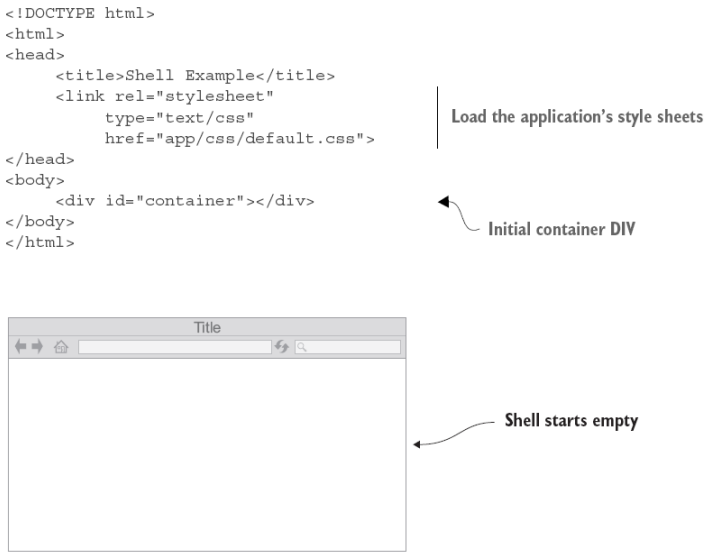

Abbildung 2.2.: SPA Shell
(Quelle: Scott 2015, Seite 8)

Voneinander getrennt sichtbare Bereiche der Anwendung können im weiteren Verlauf ebenfalls mit `DIV` Tags abgegrenzt werden und sind dem in der Shell

definierten `DIV` Container untergeordnet. Dies ermöglicht sowohl eine logische als auch inhaltliche Gruppierung und das gezielte Austauschen bestimmter Bereiche, sogenannter *Regions*. (Scott 2015, Seite 9)

Die einzeln dargestellten Views, welche dynamisch ausgetauscht werden können, stellen keine vollständigen HTML-Seiten dar, sondern bilden lediglich gezielt definierte Ausschnitte. Diese Ausschnitte werden bei jedem Navigationsvorgang innerhalb der Website durch das entsprechend eingesetzte Framework ausgetauscht und erfordern kein erneutes Laden der Website. (Scott 2015, Seite 10f.)

2.2. Vorteile einer SPA gegenüber einer MPA

Der Einsatz und die Entwicklung einer SPA bietet sowohl Vorteile für Entwickler:innen als auch Anwender:innen gegenüber der Nutzung einer klassischen MPA.

Der bereits mehrfach angesprochene Vorgang, lediglich bestimmte Teile beziehungsweise Views der Webanwendung auszutauschen, erhöht die Benutzbarkeit sowie die User Experience (UX) laut Mikowski und Powell deutlich. Da keine komplett neue (Unter-)Seite geladen werden muss, entfällt das Aufscheinen einer - abhängig von der Internet- und Servergeschwindigkeit - kurz sichtbaren, weißen Übergangsseite während des Ladeprozesses. Dem:Der Benutzer:in kann stattdessen beispielsweise ein dynamisch dargestellter Fortschrittsbalken dargestellt werden, der sich bei vorhandener Ladezeit laufend aktualisiert. (Mikowski und Powell 2013, Seite 20)

Scott hebt hervor, dass die Aufteilung in eine entkoppelte Präsentationsschicht auf dem Client dazu führt, dass diese unabhängig von der Logik auf dem Server gewartet und aktualisiert werden kann. Während bei klassischen MPAs stellenweise HTML, JavaScript etc. mit serverseitigem Code (beispielsweise *PHP*, *JavaServer Pages*, ...) vermischt werden, kann bei SPAs zudem eine gewisse Differenzierung und Abtrennung von HTML, Cascading Style Sheets (CSS) und JavaScript im Frontend erzielt werden, was die Wartbarkeit ebenfalls erhöht. (Scott 2015, Seite 13)

Sowohl Mikowski und Powell als auch Scott gehen des Weiteren darauf ein, dass die Datenübertragung und Verarbeitung bei einer SPA effizienter und schneller stattfinden kann, als bei einer MPA. Die Programmlogik zur Darstellung und dynamischen Entscheidungsfindung befindet sich beim Client (weshalb dieser Operationen schnell durchführen kann), während der Server lediglich Validierung, Authentifizierung und Datenspeicherung durchführt. (Mikowski und Powell 2013, Seite 20) Zudem sind Transaktionen zwischen Client und Server nach dem Initialen Aufruf der Applikation schneller, da lediglich Daten in einem vorab definierten Datenformat asynchron übertragen und keine kompletten HTML-Seiten samt JavaScript und CSS ausgetauscht werden müssen. (Scott 2015, Seite 13)

2.3. Funktionsweise von Vaadin

Sowohl Angular als auch Vaadin sind beides Frameworks, die das Entwickeln von SPAs unterstützen und ermöglichen. Der gewählte Ansatz, die zugrundeliegenden Technologien und die jeweiligen Herangehensweisen unterscheiden sich stellenweise jedoch deutlich voneinander.

Der nachfolgende Abschnitt befasst sich daher im Detail mit der Funktionsweise und den Konzepten von Vaadin sowohl aus dem Blickwinkel von Entwickler:innen als auch Anwender:innen. Wo sinnvoll, wird ein direkter Vergleich zu Angular gezogen und darauf eingegangen, wie die beiden Frameworks Probleme unterschiedlich handhaben und lösen.

2.3.1. Ansätze von Vaadin

Vaadin ist ein Framework beziehungsweise eine Plattform, mit der Webapplikationen sowohl komplett in Java, vollständig mit TypeScript und HTML als auch in einer Kombination beider entwickelt und umgesetzt werden können. Die beiden Ansätze sind dabei in zwei verschiedene Frameworks aufgeteilt, um - je nach Einsatzzweck - diese entsprechend einzusetzen:

- **Vaadin Flow** ist ein Framework, mit dem Webapplikationen in Java umgesetzt werden können. Technologien wie beispielsweise HTML oder

JavaScript werden bei der Entwicklung der UI nicht benötigt, der gesamte Programmcode basiert auf einer einheitlichen Programmiersprache. Die gesamte Anwendung selbst läuft auf einem Server, während das Framework den Applikationszustand sowie die Client-Server-Kommunikation übernimmt. (Vaadin Ltd 2021m)

- **Vaadin Fusion** ist ein Framework, bei dem TypeScript für das Frontend auf dem Client und Java für das Backend auf dem Server eingesetzt wird. Mit Fusion können clientseitig reaktive Webapplikationen entwickelt werden, die typsichere Java Endpunkte aufrufen. Vorgefertigte Komponenten erleichtern hierbei das Entwickeln der UI, während eigene Elemente mittels voller Kontrolle über das DOM umgesetzt werden können. (Vaadin Ltd 2021n)

Diese Arbeit befasst sich nachfolgend primär mit *Vaadin Flow*, um eine alternative Herangehensweise aufzuzeigen, wie eine SPA komplett auf dem Server und mittels Java entwickelt werden kann. Durch diesen Fokus kann im weiteren Verlauf ein tiefergehenderer Vergleich der Funktionalitäten und Konzepte von Vaadin und Angular bewerkstelligt werden, der bei spezifischen Punkten konkret auf die Unterschiede der beiden Technologien eingeht.

2.3.2. Frontend und Backend in Java

Vaadin selbst schreibt, dass sich das Arbeiten mit HTML, CSS und JavaScript für reine Java-Entwickler sowohl als herausfordernd als auch als zeitintensiv darstellt. (Vaadin Ltd 2021i, Framework - Introduction - Overview)

Vaadin Flow bietet daher die Möglichkeit, mit einer vollständig in Java geschriebenen Applikation - die auf einem Server ausfgeführt wird - jeweils die Anwendungslogik sowie das User Interface umzusetzen. Die Vielzahl von sogenannten *Components*, welche Vaadin von Haus aus mitliefert und einzelnen Bestandteilen wie beispielsweise einem Button oder ähnlichem entspricht, unterstützen Entwickler:innen dabei, bei Bedarf ohne HTML oder JavaScript auszukommen. Die Components steuern dabei das zugrundeliegende JavaScript im Hintergrund über die Framework-eigene *Java API* beziehungsweise die *Java*

Component API. Der Quellcode 1 auf Seite 11 stellt einen stark vereinfachten Ausschnitt von Components dar, die so im Client bereits entsprechend dargestellt werden. (Vaadin Ltd 2021i, Framework - Introduction - Overview)

```
1  // Create an HTML element
2  Div layout = new Div();
3
4  // Use TextField for standard text input
5  TextField textField = new TextField("Your name");
6
7  // Button click listeners can be defined as lambda expressions
8  Button button = new Button("Say hello",
9            e -> Notification.show("Hello!"));
10
11 // Add the web components to the HTML element
12 layout.add(textField, button);
```

Quellcode 1: Beispiel einer einfachen UI mittels der *Java API* (Quelle: Vaadin Ltd 2021d)

Während mit Vaadin Flow hauptsächlich in Java, und somit auf dem Server, entwickelt wird, bietet das Framework dennoch die Möglichkeit, auf Browser APIs, spezifische Web Components sowie auf das DOM zuzugreifen.

Als Vorteil stellt sich zudem heraus, dass die Anbindung des Frontends an ein Backend in den meisten Fällen mit Vaadin Flow bereits komplett vorhanden ist und nicht separat mit auf REST basierender Kommunikation umgesetzt werden muss. Dadurch kann die UI und die dafür benötigten Daten direkt über Java verknüpft und auch aktualisiert werden. Hierbei unterstützen mitgelieferte *Events* und *Event Listener* die Entwicklung und Benutzbarkeit, indem Änderungen auf dem Client automatisch auch auf dem Server - und umgekehrt - abgebildet werden. (Vaadin Ltd 2021c)

2.3.3. Kommunikation zwischen Client und Server

Bei einer SPA spielt die Kommunikation des Clients (dem JavaScript/TypeScript Frontend) und dem Server (dem Backend) eine sehr wichtige Rolle. Im Gegensatz zu MPAs, bei denen die Daten bereits auf dem Server verarbeitet, eingebettet und als Ganzes übertragen werden, werden bei SPAs diese erst bei

Bedarf und im Nachhinein geladen. Die Art und Weise, wie dieser Vorgang umgesetzt wird, unterscheidet sich bei Vaadin Flow und Angular jedoch deutlich voneinander.

2.3.3.1. Automatische Kommunikation von Vaadin

Da sowohl die Persitenzschicht, die Applikationslogik als auch das User Interface bei Vaadin Flow mittels Java umgesetzt werden können, ergibt sich der Vorteil, dass die Kommunikation zwischen Client und Server vom Framework selbst übernommen wird und hierbei unter anderem auf das bereits angesprochene *Two-way data binding* setzt. Die Nutzung von Vaadin-eigenen Components bietet des Weiteren die Möglichkeit, das DOM im Webbrowser selbst zu steuern, während eine Repräsentation desselben DOM serverseitig in Java gehalten wird. Änderungen, die auf dem Client durchgeführt werden, werden automatisch synchronisiert. (Vaadin Ltd 2021i, Framework - Introduction - Core Concepts)

Artur Signell, CTO der Vaadin Ltd., erklärt währenddessen in einem Foren-Beitrag vom Juni 2018, dass genauere Informationen zur tatsächlich Umsetzung der automatisch vorgenommen Kommunikation nicht nach außen kommuniziert werden, da die Umsetzung davon als rein internes Implementierungsdetail gehandhabt wird. (Signell 2018)

Mittels den Entwicklertools, die in gängigen Webbrowsern zur Verfügung stehen, kann sich - abseits der nicht vorhandenen Dokumentation - jedoch ein kurzes Bild davon gemacht werden, wie die Kommunikation grundsätzlich funktioniert. Die Abbildung 2.3 auf Seite 13 zeigt die vom Client an den Server geschickte Anfrage, die durch das Befüllen eines Vaadin `TextField` (einem HTML - `INPUT` Element) ausgelöst wird. Für die Kommunikation wird JSON eingesetzt, welches die in das `INPUT` Element eingefüllten Daten mit gängiger HTTP-Kommunikation an den Server schickt, auf die schlussendlich mittels Java-typischer Notation zugegriffen werden kann.

Abbildung 2.3.: Automatisch erzeugte Kommunikation von Vaadin Flow (Quelle: eigene Abbildung)

2.3.3.2. Anpassbare Kommunikation bei Angular

Im Vergleich zur automatischen Kommunikation, die Vaadin Flow von Haus aus bietet, unterstützt Angular Entwickler:innen zwar beim Datenaustausch mit einem Server, die konkrete Umsetzung muss jedoch deutlich eigenständiger programmiert werden.

Der von Angular entwickelte `HttpClient` Service - der über das `@angular/common/http` Package bezogen werden kann - stellt eine entsprechende API zur Verfügung, mit der typisierte `Response` Objekte angefordert werden können, eine vereinheitlichte Fehlerbehandlung ermöglicht wird sowie das Abfangen und Bearbeiten von `Request` und `Response` Objekten erlaubt. (Google LLC 2021b)

Der Einsatz der angesprochenen API kann bei Angular grundsätzlich im gesamten Projekt erfolgen, Wilken empfiehlt jedoch, die Nutzung von der tatsächlichen Logik zu abstrahieren und dafür einen eigenständigen Service zu erstellen. Der `HttpClient`, welcher zur Kommunikation mit einem Server (und gegebenenfalls in einem eigenständigen Service) verwendet wird, unterstützt HTTP Request-Methoden wie beispielsweise `GET`, `POST`, `PUT` und `DELETE`, die bei einem entsprechenden Aufruf ein `Observable` als Antwort liefern. Mit die-

sem kann in weiterer Folge in der Applikation gearbeitet und auf die Daten zugegriffen werden. (Wilken 2018, Seite 142-144.)

Der Quellcode 2 auf Seite 14 zeigt einen exemplarischen Service, der mittels `HTTP GET` eine Anfrage an den Server beziehungsweise einen API Endpunkt stellt und ein Array des Typs `Sample` als Antwort zurückliefert.

```
[...]

@Injectable()
export class SampleService {
    constructor(private http: HttpClient) {}

    getSampleData(): Observable<Sample[]> {
        return this.http.get<Sample[]>(/* URL to server or API endpoint */);
    }
}

[...]
```

Quellcode 2: Exemplarische Nutzung des `HttpClient` in einem Service

Der Quellcode 3 auf Seite 14 zeigt in Folge die Nutzung der gerade eben demonstrierten Funktion, bei der die Daten mittels `subscribe()` abgefragt werden können und entsprechend zugewiesen werden, sobald diese zur Verfügung stehen.

```
[...]

private loadSampleData() {
    this.sampleService.getSampleData().subscribe(data => {this.data = data});
}

[...]
```

Quellcode 3: Beispielhafte Nutzung der `getSampleData` Funktion

Verglichen mit der im Abschnitt 2.3.3.1 auf Seite 12 beschriebenen Herangehensweise von Vaadin unterscheidet sich Angular deutlich. Wilken merkt an, dass die Nutzung des `HttpClient` der am häufigsten verwendete Ansatz bei Angular darstellt. Um eine Kommunikation mit einem Server herzustellen, können

laut ihm jedoch auch, beziehungsweise zusätzlich, diverse andere Protokolle und Technologien genutzt werden. (Wilken 2018)

2.3.4. Vaadin Components und deren Grundlage

Vaadin Components sind von Vaadin entwickelte „UI-Module“, die zur einfachen Entwicklung von gängigen User Interface Bestandteilen genutzt werden können. Diese sind bereits im Vaadin Framework enthalten, können im Zuge der Nutzung der Web Components Technologien jedoch unabhängig von Vaadin selbst und somit in allen gängigen Applikationen sowie Webbrowsern genutzt werden. (Vaadin Ltd 2021l, Seite 2ff.)

Die Abbildung 2.4 auf Seite 15 zeigt hierbei eine Auswahl von Vaadin Components, die häufig in Webapplikationen genutzt werden und entsprechend für das Framework umgesetzt wurden.

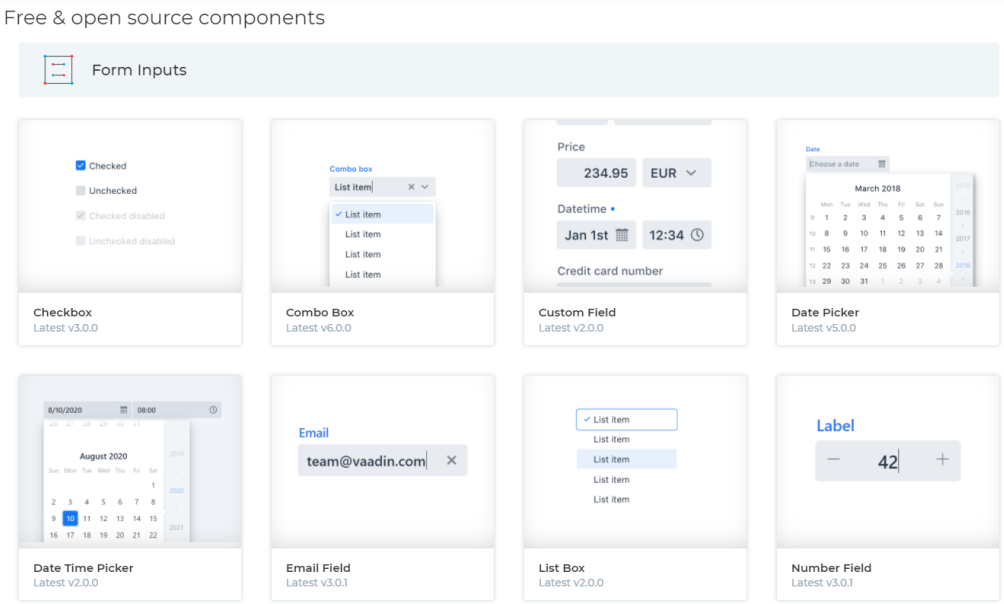

Abbildung 2.4.: Auswahl von Open Source Vaadin Components (Quelle: Vaadin Ltd 2021k)

Die Technologien, auf die sich Vaadin bei der Implementierung und Umsetzung der Components stützt, bauen auf diversen Konzepten auf, zu denen unter anderem *Custom elements*, *Shadow DOM* sowie *HTML templates* ge-

hören. Diese werden dazu genutzt, um mittels JavaScript spezifische Klassen für die gewünschten, neuen Web Components zu erstellen, diese entsprechend zu registrieren und mittels Template-Funktionalität die neu erstellten Objekte schlussendlich zu definieren. (Mozilla Contributors 2021b)

Der Quellcode 4 auf Seite 16 zeigt beispielhaft auf, wie der grundlegende Aufbau eines eigens umgesetzten Web Components aussieht. Vaadin nutzt diesen Aufbau sowie die zugrundeliegende Herangehensweise für die Umsetzung der vorhergehend aufgezeigten *Vaadin Components*. (vgl. Vaadin Ltd 2021h) Aufgrund des Fokus dieser Arbeit wird im weiteren Verlauf jedoch nicht näher auf die zugrundeliegende Struktur von Web Components eingegangen.

```
1  customElements.define('word-count', WordCount, { extends: 'p' });
2
3  [...]
4
5  class WordCount extends HTMLParagraphElement {
6    constructor() {
7      // Always call super first in constructor
8      super();
9
10     // Element functionality written in here
11
12     ...
13   }
14 }
```

Quellcode 4: Beispiel für den grundlegenden Aufbau eines eigenen Web Components (Quelle: Mozilla Contributors 2021a)

Die Nutzung der Vaadin Components lässt sich sowohl eigenständig mit anderen Frameworks als auch direkt in Vaadin selbst mittels der sogenannten *Java API* umsetzen. Jede Komponente besitzt beim Einsatz von Java entsprechende Objekt-Typen und dazugehörige Attribute, bei der Verwendung von HTML stehen Tags mit ähnlicher Funktionalität zur Verfügung. (Vaadin Ltd 2021k)

Quellcode 5 auf Seite 17 ermöglicht einen Einblick, wie ein *Text Field* Component mittels Vaadins Java API genutzt werden kann. Die deklarierten und initialisierten `TextField` Objekte können, in Java typischer Weise, an beliebiger Stelle genutzt werden. Weiterführende beziehungsweise benötigte Funktio-

nalitäten können im Anschluss über das Zuweisen und Setzen von spezifischen Attributen erzielt werden, die das Verhalten des Textfeldes beeinflussen und vorgeben.

```
1  TextField labelField = new TextField();
2  labelField.setLabel("Label");
3
4  TextField placeholderField = new TextField();
5  placeholderField.setPlaceholder("Placeholder");
6
7  TextField valueField = new TextField();
8  valueField.setValue("Value");
9
10 add(labelField, placeholderField, valueField);
```

Quellcode 5: Mögliche Umsetzungen des Vaadin *Text Field* (Quelle: Vaadin Ltd 2021j, Basic text field)

Abbildung 2.5 auf Seite 17 bezieht sich maßgeblich auf Quellcode 5, da die dargestellten Textfelder jene sind, die mittels der aufgezeigten `TextField` Objekte erzeugt wurden und mittels gesetzter Attribute unterschiedliche Ausgangspunkte liefern.

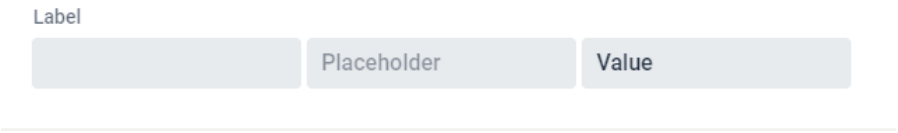

Abbildung 2.5.: Erzeugter Output von Quellcode 5 (Quelle: Vaadin Ltd 2021j, Basic text field)

2.3.5. Routing und Navigation

Wilken schreibt, dass die meisten Webapplikationen die Funktionalität benötigen, während der Nutzung zwischen verschiedenen Seiten und Unterseiten navigieren zu können. MPAs laden beim Aufruf einer Seite die hinterlegten HTML, CSS und JavaScript Dateien von einem Webserver und nutzen dafür die angegebene URL. SPAs hingegen laden Daten asynchron und bei Bedarf und

wechseln lediglich sogenannte Views und keine kompletten Seiten. Die Navigation innerhalb einer SPA muss entsprechend anderweitig vorgenommen werden, wobei hier sowohl JavaScript an sich als auch das verwendete Framework eine wichtige Rolle spielt. (Wilken 2018, Seite 159f.)

2.3.5.1. Routing bei Vaadin

Vaadin bietet die Möglichkeit, mittels der `@Route` Annotation alle Components (sowohl Vaadin Components als auch eigene) über eine spezifische URL ansprechbar zu machen und diese somit aufrufen zu können. Quellcode 6 auf Seite 18 stellt beispielhaft dar, wie eine selbst entwickelte Komponente im Browser über die URL `http://example.com/some/path` angesteuert werden kann. (Vaadin Ltd 2021e, Using the @Route Annotation)

```
1 @Route("some/path")
2 public class SomePathComponent extends Div {
3     public SomePathComponent() {
4         setText("Hello @Route!");
5     }
6 }
```

Quellcode 6: Beispielhafte Nutzung der `@Route` Annotation
(Quelle: Vaadin Ltd 2021e, Using the @Route Annotation)

Mittels verschiedener Events, die während der Nutzung der Webapplikation und der darin auftretenden Navigation ausgelöst werden, entsteht ein sogenannter *Navigation Lifecycle*. Dieser kann dafür genutzt werden, um zusätzliche Funktionalitäten anzubieten. UI Komponenten implementieren dafür entsprechende `Listener`, welche auf zugehörigen `Observer` Interfaces aufbauen. Mit dem Navigation Lifecycle und einem `BeforeLeaveEvent` kann beispielsweise eine Abfrage im User Interface erstellt werden, die Nutzer:innen bestätigen lässt, dass das Verlassen der Website womöglich zum Verlust von Daten führt. (Vaadin Ltd 2021b)

URL Parameter (sowohl solche, die Teil der URL selbst sind als auch Query Parameter) können mittels dem `HasUrlParameter` Interface abgefragt und im weiteren Verlauf genutzt werden. Wie Quellcode 7 auf Seite 19 zeigt, kann über die `setParameter` Methode auf Teile der URL zugegriffen werden, die mittels Generics direkt in der Methode verankert sind. (Vaadin Ltd 2021g) Query Parameter lassen sich währenddessen mit dem `BeforeEvent` ebenfalls in derselben Methode abfragen. (Vaadin Ltd 2021f)

```
1  [...]
2
3  public class Sample extends Div implements HasUrlParameter<String> {
4
5      @Override
6      public void setParameter(BeforeEvent event, String param) {
7          // "param" equals a String thats part of the url
8
9          [...]
10
11         // "queryParams" contains all query parameters
12         Location location = event.getLocation();
13         QueryParameters queryParams = location.getQueryParameters();
14     }
15 }
```

Quellcode 7: Zugriff auf URL Parameter bei Vaadin

Um die Navigation zwischen verschiedenen Routen zu ermöglichen, bietet Vaadin - neben dem händischen Abändern der URL im Browser - verschiedene Ansätze:

- Mittels der `RouterLink` Komponente können Links erstellt werden, welche die angegebene Route als Ziel haben (`menu.add(new RouterLink("Home", HomeView.class));`). (Vaadin Ltd 2021a, Using the RouterLink Component)
- Gewöhnliche HTML Anker Tags können genutzt werden, um zwischen verschiedenen Routen zu navigieren (`<a href="company"[...]`). Diese verursachen jedoch ein Neuladen der gesamten Seite. Wenn der zusätzliche Ladeprozess der Seite verhindert werden soll, kann das Vaadin-eigene

`router-link` Attribut zusätzlich angehängt werden. (Vaadin Ltd 2021a, Using Standard href Links)

- Serverseitig kann mittels `UI.navigate(/* Route name */)` der Wechsel zu einer neuen Route ausgelöst werden. (Vaadin Ltd 2021a, Server-side Navigation)

2.3.5.2. Routing bei Angular

Um bei Angular eine vergleichbare Routing und Navigation Funktionalität zu erzielen, wird das `@angular/router` Package benötigt, welches in das Projekt eingebunden werden muss. Anders als bei Vaadin können bei Angular Routen an zentraler Stelle definiert werden, bei denen angegeben wird, welche TypeScript Komponente für die Abhandlung zuständig ist, wie Quellcode 8 auf Seite 20 zeigt. (Google LLC 2021a)

```
1 const routes: Routes = [
2   { path: 'first-component', component: FirstComponent },
3   { path: 'second-component', component: SecondComponent },
4 ];
```

Quellcode 8: Definition von Routen bei Angular
(Quelle: Google LLC 2021a)

Vom Prinzip her ähnlich bietet Angular ebenfalls die Möglichkeit, auf die in der URL enthaltenen Parameter zuzugreifen und in der Präsentations- sowie Applikationslogik zu verwenden. Hierfür wird das `ActivatedRoute` Objekt dem Konstruktor einer Komponente übergeben, welches sowohl den Zugriff auf die Query Parameter (`ActivatedRoute#queryParams`) als auch Parameter, die Bestandteil der URL sind (`ActivatedRoute#paramMap`), ermöglicht. Zudem stehen allgemeine Informationen zur Route selbst zur Verfügung. (Google LLC 2021a)

Links innerhalb der UI werden - vergleichbar mit Vaadin - ebenso mit HTML Anker Tags und einem eigens dafür vorgesehenen Attribut erzeugt, beinhalten im Zuge der entsprechenden Darstellungsoptionen jedoch noch weitere Funktionalitäten. Quellcode 9 auf Seite 21 stellt diesen Ansatz dar und beinhaltet

zudem in Zeile 19 das benötigte `<router-outlet>` Element, welches die ausgewählten und geladenen Routen im Browser darstellt. (Google LLC 2021a)

```
1  <h1>Angular Router App</h1>
2  <!-- This nav gives you links to click, which tells the router which route
3  to use (defined in the routes constant in  AppRoutingModule) -->
4  <nav>
5    <ul>
6      <li>
7          <a routerLink="/first-component" routerLinkActive="active">
8              First Component
9          </a>
10     </li>
11     <li>
12         <a routerLink="/second-component" routerLinkActive="active">
13             Second Component
14         </a>
15     </li>
16   </ul>
17 </nav>
18 <!-- The routed views render in the <router-outlet>-->
19 <router-outlet></router-outlet>
```

Quellcode 9: Verlinken von Routen innerhalb von Angular
(Quelle: Google LLC 2021a)

3. Methodik und Vorgehensweise

Die in Kapitel 2 ab Seite 5 behandelten Punkte befassen sich mit den grundlegenden Konzepten einer SPA und den spezifischen Funktionalitäten sowie Herangehensweisen, auf denen Vaadin basiert. Wo sinnvoll, wurde zudem ein Vergleich zu Angular gezogen.

In den folgenden Kapiteln fokussiert sich diese Arbeit nun auf die Planung, Umsetzung und Beurteilung von zwei Demo-Applikationen, die auf denselben User Stories basieren, jedoch sowohl mit Vaadin als auch Angular umgesetzt werden. Ziel ist es, diese Webapplikationen entlang vorab definierter Kriterien zu entwickeln, um schlussendlich eine Bewertung durchführen zu können.

3.1. Kriterien

Die Kriterien, anhand derer sich die Umsetzung und Beurteilung richten werden, wurden bereits in der Einleitung im Abschnitt 1.3 ab Seite 3 kurz aufgegriffen. Da diese jedoch noch nicht im Detail behandelt wurden, folgt in den nächsten Absätzen eine entsprechend genauere Darstellung.

Umsetzung als Progressive Web App

Die Demo-Applikationen sollen in mehrerer Hinsicht als moderne Webapplikation genutzt werden können. Dazu gehört, neben der standardmäßigen Nutzung mittels gängiger Webbrowser auf dem Desktop, die Nutzung als PWA, die vor allem für Smartphones optimiert und ausgelegt ist und sich ähnlich wie eine App verhält.

Bei der Umsetzung und Bewertung wird daher darauf eingegangen, in wie weit sich die Webanwendungen mit den von Vaadin und Angular zur Verfügung gestellten Mitteln als PWAs umsetzen lassen und welche Funktionalitäten schlussendlich zur Verfügung stehen.

Einsatz und Entwicklung von wiederverwendbaren Komponenten

In Abschnitt 2.3.4 wurde bereits auf die Vaadin eigenen *Vaadin Components* eingegangen. Da die Wiederverwendbarkeit in der Entwicklung eine große Rolle spielt, und auch auf UI-Komponenten zutrifft, sollen mehrfach verwendete Bestandteile der Demo-Applikationen wiederverwendbar, und wo sinnvoll, mit Web Components umgesetzt werden.

Bei der Umsetzung und Bewertung wird daher darauf eingegangen, in welchem Ausmaß die Frameworks die Umsetzung dieses Vorhabens - zusätzlich zu den gegebenen Browserstandards - unterstützen und ob sie den Entwicklungsprozess dabei erleichtern.

Lokale und serverseitige Datenanbindung/-haltung

SPAs, beziehungsweise Anwendungen allgemein, leben von den Daten, die sie verarbeiten und mit denen Nutzer:innen interagieren können. Gerade im Zuge der Umsetzung der Demo-Applikationen als PWA stellt sich die Frage, wie Angular und Vaadin eine Datenanbindung und Datenhaltung lokal (sprich auf dem jeweiligen Endgerät) sowie auf einem Server ermöglichen.

Bei der Umsetzung und Bewertung wird daher darauf eingegangen, welche Möglichkeiten sich bieten, bei den entwickelten Anwendungen Daten sowohl lokal zu speichern (um beispielsweise eine Offline-Fähigkeit zu ermöglichen) und wie man diese an eine externe Datenbank auf einem Server anbinden kann.

3.2. Vorgehensweise

Damit die Demo-Applikationen die eben genannten Kriterien beide gleichermaßen beinhalten und umsetzen, und somit bewertet werden können, werden diese in User Stories eingebettet. Die User Stories orientieren sich dabei jedoch an

keinem vollständigen Applikations-Konzept, sondern zeigen verschiedene Anwendungsfälle auf, die in einem realen Umfeld ein Bewertungskriterium für die Auswahl eines Frameworks spielen können.

3.2.1. User Stories

Nachfolgend sind zwei User Stories aufgelistet, die im Detail erklären, welche Funktionalitäten die Vaadin und Angular Demo-Applikation beinhalten sollen und die genannten Kriterien einfließen lassen.

User Story 1: Echtzeit Eingangskontrolle

Beschreibung: Als Veranstalter:in einer Organisation möchte ich kommende und gehende Besucher:innen bei allen Ein-/Ausgängen in Echtzeit erfassen, sodass diese den Veranstaltungsort nur in kontrollierter Weise betreten können.

Akzeptanzkriterien:

- Besucher:innen können aus einer vorhandenen Teilnehmer:innenliste ausgewählt werden, um festzulegen, ob sie sich innerhalb oder außerhalb vom Veranstaltungsort befinden.
- Wird der Status eine:r Besucher:in aktualisiert, wird diese Änderung automatisch auf allen Instanzen sichtbar, ohne dass man diese manuell neu laden muss. Dies entspricht entspricht der Multi-Nutzer:innen-Fähigkeit.
- Die Anwendung wird sowohl auf Computern als auch im mobilen Einsatz auf einem gängigen Smartphone unterstützt.
- Wenn keine Internetverbindung vorhanden ist, ist zumindest der zuletzt verfügbare Stand der Besucher:innenliste aufrufbar.

User Story 2: Fotoverwaltung

Beschreibung: Als Nutzer:in möchte ich meine Fotos hochladen können, sodass diese an einem zentralen Ort gespeichert sind und man jederzeit darauf zugreifen kann.

Akzeptanzkriterien:

- Nutzer:innen können Fotos von ihrem Computer oder Smartphone auswählen und hochladen, damit diese in einer zentralen Galerie angesehen werden können.
- Wenn Fotos von einem anderen Client hochgeladen wurden, kann man diese mit dem aktuellen Gerät ebenso ansehen, ohne dieser erneut hochzuladen oder synchronisieren zu müssen.
- Wenn keine Internetverbindung vorhanden ist, werden Fotos lokal gespeichert und können hochgeladen werden, sobald wieder eine Verbindung zur Verfügung steht.

4. Umsetzung

Damit das Ziel dieser Arbeit - einen Vergleich zwischen den beiden SPA Frameworks Vaadin und Angular herzustellen - erreicht werden kann, wurden zwei Demo-Applikationen nach den in Kapitel 3 ab Seite 22 aufgezeigten Methoden und Herangehensweisen entwickelt.

Bevor auf die erzielten Ergebnisse eingegangen wird, folgt in diesem Kapitel eine kurze Erläuterung dazu, wie die zwei Webanwendungen gestartet beziehungsweise verwendet werden können und wie deren Entwicklung und Umsetzung abgelaufen ist. Zudem wird kurz auf einige (technische) Hürden eingegangen, die währenddessen aufgetreten sind.

Der Quellcode der beiden Anwendungen ist der physischen Abgabe als Kopie beigelegt und zusätzlich jederzeit auf GitHub abrufbar: `https://github.com/DominicLuidold/Bachelor-Thesis-II`

4.1. Vaadin Demo-Applikation

Als erste der beiden Applikationen wurde die Vaadin Demo umgesetzt und *User Story 1: Echtzeit Eingangskontrolle* sowie *User Story 2: Fotoverwaltung* dafür abgeschlossen. Die Entwicklung wurde bewusst aufgeteilt, sodass währenddessen ein Fokus auf das jeweilige Framework und die entsprechenden Technologien gelegt werden konnte und kein kontinuierliches Umdenken erforderlich war.

Zur Umsetzung der Vaadin Demo-Applikation kommt die Version `19.0.2` zum Einsatz, die zum Stand des Entwicklungsbeginns Anfang März 2021 die aktuellste Version darstellt. In Kapitel 2 ab Seite 5 wurden stellenweise Auszüge aus der Dokumentation zu Vaadin `18.x` verwendet, diese behalten jedoch weiterhin ihre Gültigkeit und haben sich nur marginal verändert.

4.1.1. Starten und Verwenden der Vaadin Demo

Vaadin Flow unterstützt sowohl für die aktuelle Long Term Support (LTS) Version `14.x` als auch für Version `19.x` den Einsatz von Java `8`, `11` und `16`. Im Zuge der Umsetzung der Vaadin Demo wird auf Java `11` gesetzt, weshalb diese Version auf dem ausführenden System installiert sein muss.

Zusätzlich zu einer vorhandenen Java Umgebung wird *Apache Maven*[1] benötigt, welches alle projektspezifischen Abhängigkeiten lädt und verwaltet. Abgesehen von einem gängigen Webbrowser müssen keine zusätzlichen Tools/etc. heruntergeladen werden.

Die Vaadin Webanwendung kann verhältnismäßig einfach gestartet werden und unterstützt dabei zwei verschiedene Modi (für beide muss man vorab in das entsprechende Anwendungsverzeichnis navigieren):

- **Development Mode:** Im Anwendungsverzeichnis `vaadin-demo-app/` muss lediglich der Befehl `mvn` in einem Terminal eingegeben werden. Daraufhin werden alle benötigten Abhängigkeiten automatisch geladen und nach kurzer Zeit ist die Anwendung unter `http://localhost:8080` aufrufbar.

- **Production Mode:** Die Vorgehensweise ist ähnlich zum Development Mode, statt dem direkten Ausführen der Anwendung im jeweiligen Terminal wird nach der Eingabe von `mvn clean package -Pproduction` eine `.jar` Datei im Ordner `target/` erstellt. Diese kann im weiteren Verlauf dafür genutzt werden, die Webanwendung auf einem beliebigen System auszuführen und bereitzustellen.

4.1.2. Entwickeln mit Vaadin

Das Entwickeln einer Vaadin Flow Applikation benötigt nur wenige technische Voraussetzungen, die in Abschnitt 4.1.1 auf Seite 27 bereits aufgezeigt wurden. Vaadin selbst erleichtert den Start zudem mittels einer eigenen Website[2], auf

[1] Apache Maven (https://maven.apache.org)
[2] Get Started | Vaadin (https://vaadin.com/start)

der verschiedene *Hello World*-Beispiele heruntergeladen werden können. Bei Bedarf können zudem erste Konfigurationsschritte im Browser getätigt und das daraus resultierende Projekt als Start für die Entwicklung genutzt werden.

Die für diese Arbeit entwickelte Vaadin Demo-Applikation setzt neben den bereits genannten Komponenten auch auf Spring Boot, welches beim Erstellen einer neuen Vaadin Flow Anwendung standardmäßig mitgeliefert wird. Spring Boot hat sich während der Umsetzung als wichtiges Hilfsmittel erwiesen, da entsprechend zur Verfügung stehende Annotations und eine einfache Integration mit dem ORM-Framework *Hibernate* die Entwicklung beschleunigt und den benötigten Konfigurationsaufwand deutlich minimiert haben. Die *Live Reload*[3] Funktionalität hat es zudem ermöglicht, Änderungen am Java Quellcode vorzunehmen, während der Applikationsserver gestartet ist. Diese Änderungen werden durch den optimierten „Restart“ deutlich schneller im Browser dargestellt, verglichen mit dem händischen Stoppen und erneutem Starten der Anwendung.

Das Arbeiten mit der von Vaadin veröffentlichten Dokumentation selbst hat in den meisten Fällen gut funktioniert. Verglichen mit anderen Frameworks oder Tools hat sich jedoch bereits früh bemerkbar gemacht, dass Vaadin deutlich weniger beziehungsweise eingeschränkter eingesetzt wird. Neben der offiziellen Dokumentation und einigen hilfreichen Foreneinträgen bei Vaadin selbst, finden sich nur wenige Lösungsansätze auf den gängigen Entwickler-Plattformen oder bei der Nutzung einer der gängigen Suchmaschinen. Dieser Umstand hat sich vor allem dann als Nachteil herausgestellt, wenn die verfügbaren Ansätze entweder unvollständig oder nicht sehr detailreich waren und das Verständnis für die entsprechende Herangehensweise von Vaadin noch nicht gänzlich vorhanden ist.

Abseits der angebrachten Punkte stellt sich das Arbeiten mit Vaadin Flow vor allem für Entwickler:innen, die bereits mit Java gearbeitet haben, jedoch als gewohnt dar und funktioniert gut. HTML oder JavaScript Kenntnisse werden keine benötigt und die *Vaadin Components* beziehungsweise eigene Komponenten lassen sich meist mit wenigen Zeilen CSS an die jeweiligen Bedürfnisse anpassen.

[3]Live Reload (https://vaadin.com/docs/latest/guide/live-reload)

4.2. Angular Demo-Applikation

Nach der abgeschlossenen Entwicklung der Vaadin Demo wurde der Fokus auf die Entwicklung der Angular Demo und dem dazugehörigen *Node.js*[4] Backend gelegt, die ebenfalls *User Story 1: Echtzeit Eingangskontrolle* sowie *User Story 2: Fotoverwaltung* beinhalten.

Um die Angular Webanwendung verwenden zu können, bedarf es - im Vergleich zur Vaadin Umsetzung - zwei verschiedener und weitestgehend unabhängiger Projekte beziehungsweise Applikationen. Auf der einen Seite die Applikation, die im Client dargestellt und angewandt werden kann und auf der anderen Seite jene, die auf einem Server läuft und Daten verarbeitet, bereitstellt sowie zentral persistiert.

Zur Umsetzung der Angular Demo-Applikation kommen Angular `11.2` für die Frontend-Entwicklung und Node.js `14.16` mit Express `4.17` im Backend zum Einsatz. Sowohl das Frontend als auch das Backend werden mittels TypeScript (in der Version `4.x`) entwickelt und für den schlussendlichen Betrieb zu JavaScript kompiliert. Die verwendeten Versionen entsprechen zum Stand des Entwicklungsbeginns Ende März 2021 jeweils den aktuellsten Versionen beziehungsweise bei Node.js der aktuellen LTS Version.

4.2.1. Starten und Verwenden der Angular Demo

Da die Angular Demo in zwei separate Projekte aufgeteilt ist, müssen diese auch jeweils eigenständig gestartet werden. Als Voraussetzung dafür wird in beiden Fällen lediglich eine vorhandene Node.js Installation benötigt, welche *npm* mitliefert und damit die Verwaltung der projektspezifischen Abhängigkeiten übernimmt.

Im Regelfall sollte zuerst das Backend gestartet werden, bevor im Anschluss die Angular Webanwendung selbst in Betrieb genommen wird. Um das Backend zu starten, müssen folgende Schritte durchgeführt werden:

[4] Node.js (https://nodejs.org)

- **Abhängigkeiten installieren:** Im Verzeichnis `angular-backend/` müssen die benötigten Abhängigkeiten mittels `npm install` installiert werden.

- **Backend starten:** Anschließend muss, ebenfalls im Anwendungsverzeichnis, der Node.js Server mit dem Ausführen des Befehls `npm start` gestartet werden. Nach kurzer Zeit steht das Backend unter `http://localhost:8181` zur Verfügung.

Die Angular Demo kann mit etwas mehr Aufwand, aber dennoch verhältnismäßig einfach, gestartet werden und unterstützt dabei wieder zwei verschiedenen Modi (hierbei ist zu beachten, dass die PWA Funktionalität lediglich im *Production Mode* zur Verfügung steht):

- **Development Mode:** Im Anwendungsverzeichnis `angular-demo-app/` müssen zuerst die Abhängigkeiten mittels `npm install` installiert werden. Anschließend kann die Angular Demo über das Ausführen der Befehle `npm install -g @angular/cli` und `ng serve -o` gestartet werden und ist nach kurzer Zeit unter `http://localhost:4200` aufrufbar.

- **Production Mode:** Nach dem Laden der Abhängigkeiten (siehe Development Mode) muss die Anwendung mit dem Befehl `ng build --prod` für den Production Mode vorbereitet werden. Sobald der Vorgang abgeschlossen ist, stehen die fertigen Projektdateien im Anwendungsverzeichnis im Ordner `dist/angular-demo-app/` zur Verfügung.
  Diese Dateien können jedoch nicht direkt aufgerufen werden, sondern müssen dem Browser von einem Webserver zur Verfügung gestellt werden. Mit `npm install -g http-server` kann ein kleiner HTTP-Server installiert werden, der anschließend über ein Terminal mit den Befehlen `cd dist/angular-demo-app/` und `http-server -o -p 8282` gestartet werden kann. Die Anwendung steht nach kurzer Zeit unter `http://localhost:8282` zum Aufruf bereit und enthält, im Vergleich zum Development Mode, die PWA Funktionalität.

Zu beachten ist, dass die Angular Demo standardmäßig nur auf dem Gerät funktionsfähig ist, auf dem auch das Node.js Backend gestartet wurde. Um

das zu ändern, muss im Verzeichnis `angular-demo-app/src/environments/` in der Datei `environment.ts` die IP Adresse individuell angepasst werden. Im *Development Mode* werden die Änderungen automatisch übernommen, im *Production Mode* müssen die oben angeführten Schritte erneut vorgenommen werden.

4.2.2. Entwickeln mit Angular und Node.js

Ähnlich wie bei der Entwicklung mit Java und Vaadin, erfordert das Arbeiten mit den (Web-)Technologien - auf denen sowohl die Angular Demo-Applikation und das dazugehörige Backend basieren - nur wenige technische Voraussetzungen (siehe dazu Abschnitt 4.2.1 ab Seite 29).

Während es für einen Node.js Server unzählige Anleitungen und Beispiele im Internet gibt, unterstützt Angular den Entwicklungsstart und auch die weiterführende Umsetzung mit der *Angular CLI*[5]. Sowohl das Erstellen eines *Hello World* Projekts als auch das Generieren von gängigen Komponenten lässt sich damit einfach handhaben und wurde während der Entwicklung aktiv eingesetzt.

Im Vergleich zur Vaadin Demo hat sich der Programmieraufwand hierbei leicht erhöht, da neben der eigentlichen Anwendung selbst auch ein dazugehöriges Backend entwickelt werden musste. Da beide Anwendungen auf TypeScript, und somit JavaScript, basieren und die Herangehensweise ähnlich ist, hat dieser Umstand die Entwicklung jedoch nicht spürbar negativ beeinflusst. Als wichtiger Punkt kommt hinzu, dass die zur Verfügung stehende Dokumentation ausreichend ist und vor allem der große Vorteil besteht, dass sowohl Angular als auch Node.js mit einem *Express*[6] Server sehr weit verbreitet sind. Entsprechende Suchen in den gängigen Suchmaschinen führen zu unzähligen Tutorials, Antworten auf gängige Fragen und Beispielen, die bei der Entwicklung extrem hilfreich sein können.

Sofern technische „Hürden“ während der Entwicklung aufgetreten sind, hatten diese primär mit der Umsetzung der PWA Funktionalität zu tun. Die von

[5] Angular CLI (https://angular.io/cli)
[6] Express (https://expressjs.com)

Angular bereitgestellte PWA Implementierung lässt sich nur in dem in Abschnitt 4.2.1 ab Seite 29 dargelegten *Production Mode* testen. Vorgenommene Änderungen können daher nicht wie gewohnt direkt begutachtet werden, sondern müssen zuerst den immer gleichen Prozess durchlaufen. Zudem haben die Browser die Website des öfteren gecached, sobald die PWA genutzt oder installiert wurde. Dadurch war nicht immer direkt ersichtlich, ob nun die aktuelle Version dargestellt wird und ob diese bereits aktualisiert wurde.

Der Umstieg auf einen Mix zwischen HTML, CSS und JavaScript/TypeScript hat sich bei der Umsetzung am deutlichsten bemerkbar gemacht und ist auch einer der offensichtlichen Punkte, in denen sich Vaadin und Angular unterscheiden. Sowohl der deutlich größere Projektumfang als auch die veränderten Programmierparadigmen erfordern eine gewisse Einarbeitungszeit, was sich während der Entwicklung an machen Stellen als Vorteil, an anderen Stellen aber auch als Nachteil erwiesen hat. Für Entwickler:innen, die bisher noch wenig mit diesen Technologien gearbeitet haben, ist der Einstig jedoch gut möglich und wahrscheinlich etwas leichter, verglichen mit Vaadin und dem objektorientierten Ansatz von Java.

5. Ergebnisse

In den vorhergehenden Kapiteln wurde auf den aktuellen *Stand der Technik* eingegangen, die *Methodik und Vorgehensweise* der zu dieser Arbeit gehörenden Entwicklung beleuchtet und anschließend auf die *Umsetzung* eingegangen. Neben den Erfahrungen, die sich primär auf die Entwicklung selbst beziehen und in Kapitel 4 ab Seite 26 bereits beschrieben wurden, spielen vor allem aber die tatsächlich erzielten Ergebnisse der Vaadin und Angular Applikationen eine entscheidende Rolle.

In den nachfolgenden Abschnitten wird daher detailliert darauf eingegangen, wie die schlussendlichen Resultate ausgefallen sind und wie diese technisch umgesetzt wurden. Wo sinnvoll, wird zusätzlich die Anwendung aus der Sicht von Nutzer:innen beleuchtet und die Unterschiede sowie Ähnlichkeiten der beiden Demo-Applikationen aufgegriffen.

5.1. Die SPAs im „äußerlichen“ Vergleich

Die Demo-Applikationen wurden so entwickelt, dass sowohl die Vaadin als auch die Angular Umsetzung jeweils dieselben Funktionalitäten (soweit möglich) aufweisen und den Nutzer:innen anbieten. Das zugrundeliegende Aussehen der Anwendungen war von dieser Anforderung jedoch ausgenommen, weshalb zwei stellenweise sehr unterschiedliche Resultate entstanden sind.

Bei der Entwicklung der Angular Webanwendung kam die von Angular selbst entwickelte *Material UI component library*[1] zum Einsatz. Diese stellt diverse *Components* - wie beispielsweise Tabellen, Buttons, „Snackbars“ etc. - zur Verfügung, die anschließend in der Anwendung verwendet und beliebig angepasst

[1] Angular Material (https://material.angular.io)

werden können. Die Abbildung 5.1 auf Seite 34 zeigt die eben genannten Components in der Angular Demo im Einsatz. Der Stil von Angular Material ist dabei am Material Design von Google[2] angelehnt und ähnelt dem Aussehen des Betriebssystems Android sowie dafür verfügbaren Apps. Die Verwendung der verschiedenen Komponenten gestaltet sich als relativ einfach, bezogen auf das Anpassen des jeweiligen Aussehens und Verhaltens bei unterschiedlichen Geräte- und Displaygrößen.

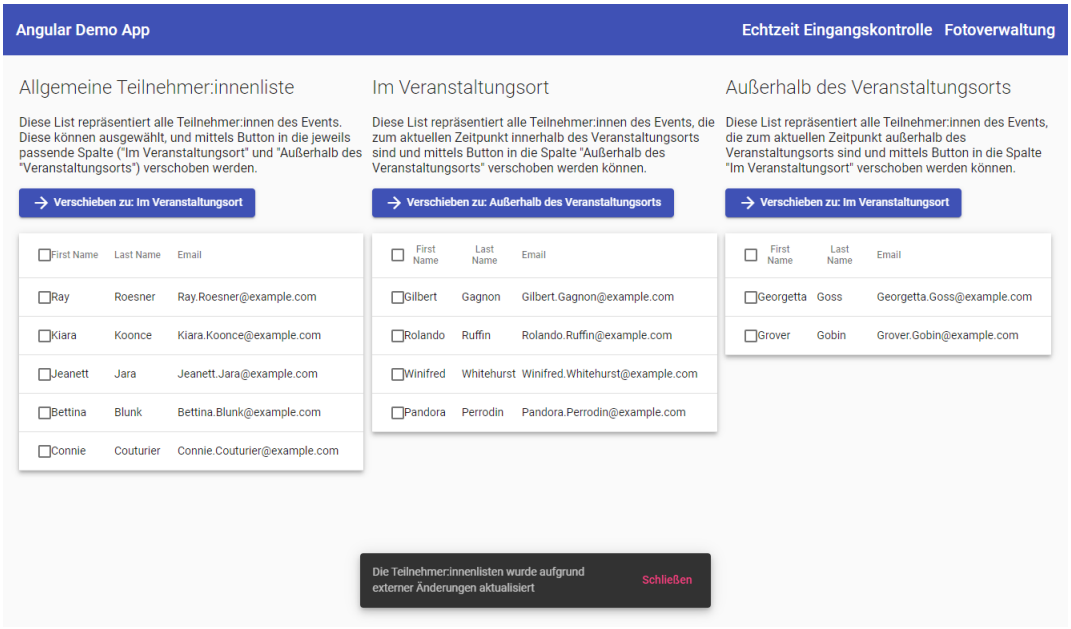

Abbildung 5.1.: Funktionalität *Echtzeit Eingangskontrolle*, umgesetzt mit Angular

Die Vaadin Applikation baut währenddessen auf den in Abschnitt 2.3.4 ab Seite 15 beschriebenen *Vaadin Components* auf, welche beim sogenannten „Styling“ vergleichbar mit den Komponenten von Angular sind. Der von Vaadin standardmäßig zur Verfügung gestellte Stil verfolgt dabei ein deutlich anderes Design-Konzept, wie die Abbildung 5.2 auf Seite 35 aufzeigt.

Aufgrund des verwendeten Technologie-Stacks haben Vaadin Applikationen die Eigenheit, dass das Styling nicht nur mittels dem gewohnten Einsatz von CSS durchgeführt, sondern parallel dazu auch im Java Quellcode vorgenommen

[2]Material Design (https://material.io)

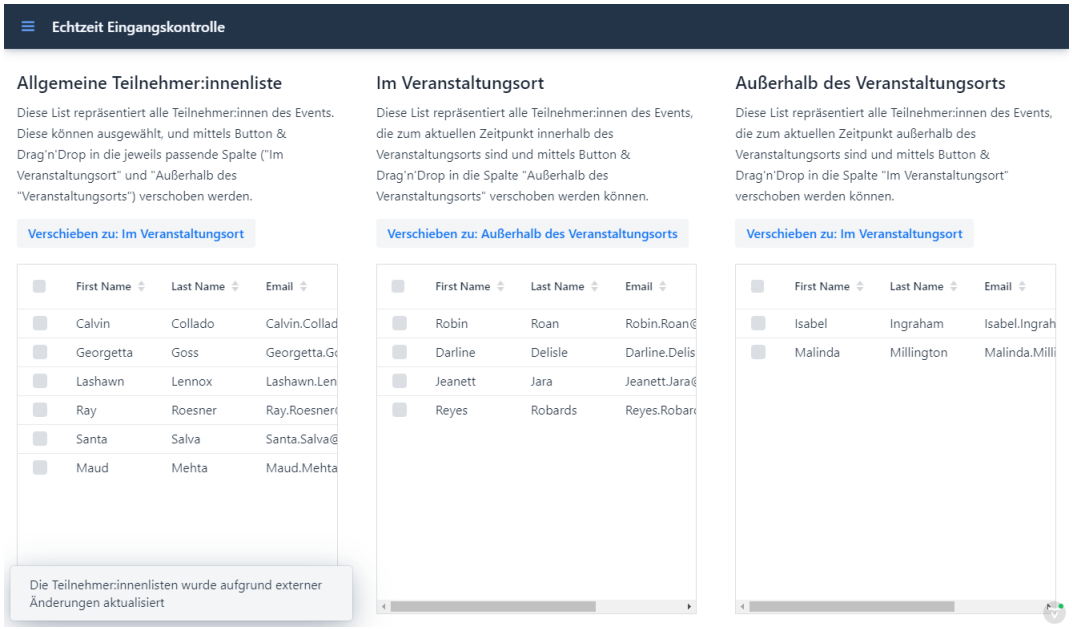

Abbildung 5.2.: Funktionalität *Echtzeit Eingangskontrolle*, umgesetzt mit Vaadin

werden kann. Das ermöglicht auf der einen Seite ein Design, welches dynamisch anpassbar ist und dabei zusätzlich auf Werte vom Backend zurückgreifen kann. Auf der anderen Seite erschwert das zweigeteilte Anpassen des Designs an manchen Stellen jedoch auch die Entwicklung, da nicht immer eindeutig klar ist, an welcher Stelle welche CSS-Regel greift, Priorität hat oder von wo diese stammt.

Aufgrund der vorgegebenen Strukturen und der Möglichkeit, CSS an mehreren Stellen einzusetzen, werden bei der für diese Arbeit entwickelten Vaadin Demo-Applikation lediglich 26 Zeilen[3] CSS benötigt, um die Anwendung anzupassen und auch für den mobilen Einsatz verwenden zu können. Bei der Angular Demo hingegen kommen mehr solcher selbst erstellter Styling-Regeln zum Einsatz, da das Framework an manchen Stellen weniger restriktiv ist, teils weniger Styling vorgibt oder auch andere Komponenten und Libraries eingesetzt werden können. Das Responsive Design hat zudem etwas mehr händische Arbeit erfordert, da Angular hierbei weniger Vorarbeit im Vergleich zu Vaadin leistet.

[3] ausgenommen hiervon sind von Vaadin generierte Stylesheets

5.2. Ergebnisse der Umsetzung als PWA

Einer der drei Kriterien, die eine wesentliche Rolle bei der Entwicklung der Demo-Applikation gespielt haben, ist die Umsetzung der Webanwendung als PWA (siehe dazu *Umsetzung als Progressive Web App* auf Seite 22).

Während der Entwicklung der Anwendungen hat sich in beiden Fällen gezeigt, dass die Frameworks eine PWA-Funktionalität anbieten und unterstützen. Der jeweilige Umfang, mit dem aus Entwickler:innensicht gearbeitet und der aus Anwender:innensicht schlussendlich genutzt werden kann, unterscheidet sich jedoch in wesentlichen Punkten.

5.2.1. Eingeschränkte PWA-Funktionalität bei Vaadin

Die Implementierung der PWA-Funktionalität gestaltet sich bei Vaadin Flow als sehr einfach und ist mit nur wenigen Zeilen Programmcode umgesetzt. Quellcode 10 auf Seite 36 zeigt die hierfür benötigte Java Annotation, die die bestehende Anwendung zur Installation mittels gängiger Webbrowser (sowohl am Desktop als auch auf mobilen Endgeräten) zur Verfügung stellt.

```
1 @PWA(
2     name = "Vaadin Demo App",
3     shortName = "Vaadin Demo App",
4     offlineResources = {"images/logo.png"}
5 )
6 public class Application extends SpringBootServletInitializer implements
  ↪  AppShellConfigurator {
7     [...]
8 }
```

Quellcode 10: Konfiguration der PWA-Funktionalität bei Vaadin Flow

Die Webanwendung lässt sich nach der oben beschriebenen Konfiguration sowohl als gängige Website benutzen, kann zusätzlich aber nun auch über ein browser-eigenes Interaktionsmenü auf dem jeweiligen Gerät „installiert“ werden. Unter Windows verhält sich die Anwendung (bei der Verwendung von auf Chromium basierenden Webbrowsern) dabei relativ ähnlich wie eine gewöhnliche Desktopanwendung. Auf einem iPhone kann die Vaadin Demo, nachdem

diese zum Home-Bildschirm hinzugefügt wurde, ebenfalls vergleichbar mit einer nativen App genutzt werden, wie Abbildung 5.3 auf Seite 37 zeigt.

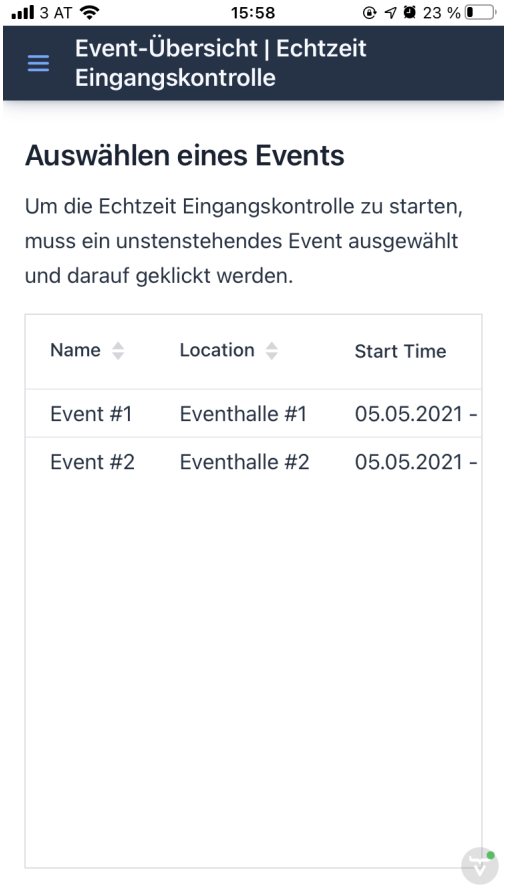

Abbildung 5.3.: Die Vaadin Demo als PWA unter iOS

Weiterführende Funktionalitäten bietet das Vaadin Framework jedoch nur noch eingeschränkt an. Eine Offline-Fähigkeit (die mit dem Kriterium *Lokale und serverseitige Datenanbindung/-haltung* auf Seite 23 zusammenspielt) wird lediglich dahingehend unterstützt, dass eine konfigurierbare Offline-Seite angezeigt wird, sollte die Internetverbindung unterbrochen werden. Das lokale Cachen von Daten oder das Navigieren durch die Anwendung ohne Daten wird nicht unterstützt. Durch diesen Umstand lässt sich die PWA ohne Netzwerkverbindung praktisch nicht nutzen und verliert die anderweitig vorhanden Vorteile.

5.2.2. PWA mit Offline-Fähigkeit bei Angular

Während die Konfiguration der PWA-Funktionalität bei Angular etwas mehr Zeit als bei Vaadin in Anspruch nimmt, bietet die zur Verfügung gestellte Integration des sogenannten *Service Workers* deutlich mehr Flexibilität bei der Entwicklung und dadurch mehr Funktionalitäten bei der tatsächlichen Nutzung der Anwendung.

Nach dem Ausführen des Befehls `ng add @angular-pwa` bei einem bestehenden Projekt wird von Angular ein Großteil der initialen Konfiguration und Installation aller Abhängigkeiten übernommen. Im Anschluss steht die Webanwendung, gleich wie im vorherigen Abschnitt beschrieben, als Website und installierbare Applikation (sprich PWA) zur Verfügung. Abbildung 5.4 auf Seite 38 zeigt die Angular Demo, ebenfalls wieder auf einem iPhone unter dem Betriebssystem iOS.

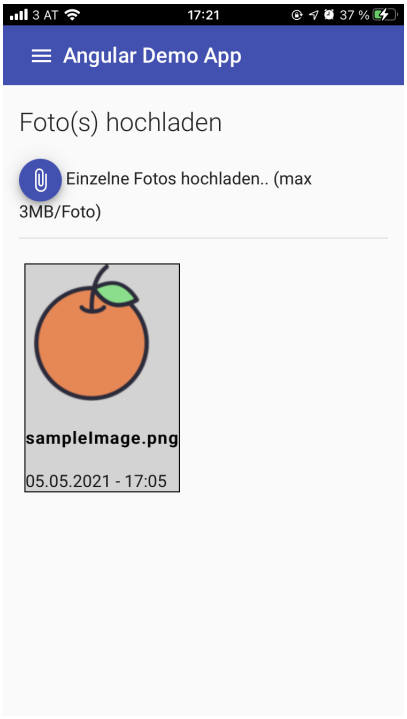

Abbildung 5.4.: Die Angular Demo als PWA unter iOS

Der entscheidende Vorteil des Ansatzes von Angular ist der, dass neben der Umsetzung als PWA selbst auch mit überschaubarem Aufwand eine Offlinenut-

zung integriert werden kann. Durch die Installation des `@angular-pwa` Package werden verschiedene Konfigurationsdateien für den Service Worker im Projekt hinterlegt. Mit nur wenigen Änderungen an diesen Dateien kann Angular mitgeteilt werden, welche Daten bei einem Aufruf eines API Endpunkts lokal zwischengespeichert beziehungsweise gecached werden sollen. Quellcode 11 auf Seite 39 zeigt die vorgenommen Änderungen an der `ngsw-config.json` Datei, welche bewirken, dass die gecacheten Daten der Endpunkte `/events`, `/attendees` und `/photos` verwendet werden sollen, sollte keine aktive Netzwerkverbindung zum Zeitpunkt des Aufrufs der Seite bestehen.

```
[...]

"dataGroups": [
  {
    "name": "angular-backend",
    "urls": [
      "/events",
      "/attendees",
      "/photos"
    ],
    "cacheConfig": {
      "strategy": "freshness",
      "maxSize": 1000,
      "maxAge": "1h",
      "timeout": "5s"
    }
  }
]

[...]
```

Quellcode 11: Konfiguration der PWA Offline-Funktionalität in der `ngsw-config.json` Datei

Die oben gezeigte Konfiguration ermöglicht es Anwender:innen also, zuvor aufgerufene Daten ansehen zu können, wenn das jeweilige Gerät offline ist. Zudem ermöglicht die Implementierung des Service Workers von Angular die Nutzung der kompletten Anwendung im Offline-Modus. Das bedeutet, dass auch zwischen bestehenden Seiten beziehungsweise Views navigiert werden kann, womit sich die Umsetzung von Angular deutlich von der von Vaadin abhebt.

5.3. Wiederverwendbare Komponenten bei Vaadin und Angular

Das zweite Kriterium, welches bei der Umsetzung der Demo-Applikationen maßgeblich betrachtet wurde, ist das gezielte Einsetzen von wiederverwendbaren Komponenten (siehe dazu *Einsatz und Entwicklung von wiederverwendbaren Komponenten* auf Seite 23). Ähnlich wie bei der Umsetzung der Anwendungen als PWA, bieten auch bei diesem Punkt beide Frameworks entsprechende Möglichkeiten, die stellenweise jedoch auf die jeweilige Programmiersprache und die dazugehörigen Programmierparadigmen zurückgeführt werden können.

5.3.1. Objektorientierte Komponenten bei Vaadin

Vaadin Flow baut auf Java und somit einer objektorientierten Programmiersprache auf. Das bedeutet, dass sowohl die von Vaadin entwickelten Vaadin Components als auch eigene Bestandteile der Anwendung in Java-typischer Weise verwendet und erweitert werden können.

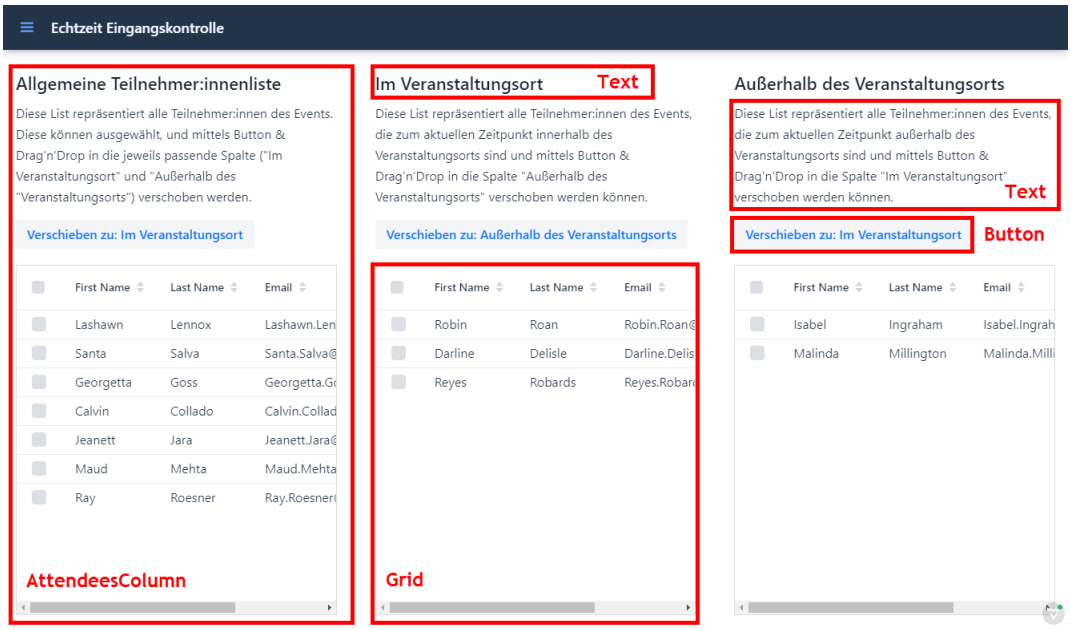

Abbildung 5.5.: Komponenten der *Echtzeit Eingangskontrolle* bei Vaadin

Ein deutliches Beispiel für die Umsetzung von wiederverwendbaren Komponenten stellt die Übersicht der *Echtzeit Eingangskontrolle* dar. Die in Abbildung 5.5 auf Seite 40 rot eingerahmten Elemente bilden eigenständige Komponenten ab, die entweder von Vaadin bereitgestellt werden oder im Zuge der Umsetzung eigenständig entwickelt wurden. Diese lassen sich wie folgt einteilen:

- `AttendeesColumn` ist eine Eigenentwicklung und entspricht einer der insgesamt drei dargestellten, großen Spalten. Diese Klasse umfasst die nachfolgend aufgelisteten Komponenten, stellt diese in einem vertikalen Layout dar und dient zur zentralen Konfiguration und zur Übergabe als übergeordnetes „Element“ an die Java API von Vaadin.

- `Grid` ist eine Vaadin Komponente und entspricht einer der drei dargestellten Tabellen. Mittels verschiedener Attribute und Methoden kann die Tabelle beliebig angepasst oder ausgetauscht oder bei Bedarf über die gewohnte Klassenhierarchie von Java weiter spezialisiert werden.

- `Text` & `Button` sind ebenfalls Vaadin Components und bieten ähnliche Konfigurationsmöglichkeiten wie die `Grid` Komponente.

Aufgrund der von Vaadin vorgegebenen Empfehlungen und Herangehensweisen enthält eine übergeordnete View-Klasse die oben genannten Elemente und die dazugehörige Logik. Quellcode 13 im Anhang A auf Seite 49 zeigt einen kurzen Ausschnitt dar, wie solch eine Klasse aufgebaut ist.

Die *Vaadin Components* selbst sind bereits eigenständige Web Components, wie in Abschnitt 2.3.4 ab Seite 15 beschrieben wurde. Für die Umsetzung der einzelnen User Stories hat die Verwendung der bestehenden Vaadin Components sowie die gezielte Entwicklung von einigen, eigenen Klassen ausgereicht, um eine ausreichende Wiederverwendbarkeit zu gewährleisten. Sofern Komponenten noch weiter abstrahiert werden müssen, bietet Vaadin jedoch die Möglichkeit, eigene Java Komponenten erstellen zu können.

5.3.2. Wiederverwendbare Angular Komponenten

Im Vergleich zu den vorgefertigten Komponenten, die Vaadin bereitstellt, unterscheidet sich die Herangehensweise von Angular bei diesem Thema. Die Komponenten, die entweder händisch oder mit dem Befehl `ng g component <name>` erstellt werden können, bestehen jeweils aus einer HTML, einer CSS und einer JavaScript/TypeScript Datei. Quellcode 12 auf Seite 42 zeigt einen Ausschnitt solch einer Komponente und wie diese grundlegend aufgebaut ist.

```
1  [...]
2
3  @Component({
4    selector: 'app-entrance-control',
5    templateUrl: './entrance-control.component.html',
6    styleUrls: ['./entrance-control.component.css']
7  })
8  export class EntranceControlComponent implements OnInit, AfterViewInit,
   ↪  OnDestroy {
9    @ViewChildren(AttendeeColumnComponent) attendeeColumns;
10
11   ngOnInit(): void {
12     // Subscribe to real-time updates via WebSockets
13     this.webSocketSubscription = this.realTimeService.getRealTimeUpdates(/*
       ↪  [...] */);
14   }
15
16   [...]
17 }
```

Quellcode 12: Ausschnitt der `EntranceControl` Komponente

In Zeile 3-7 ist deutlich zu erkennen, wie die Aufteilung der verschiedenen Technologien funktioniert und in welchen Dateien diese zu finden sind. Um diesen selbst erstellten Component beispielsweise in der Anwendung einsetzen zu können, wird der HTML-Tag `<app-entrance-control>` verwendet, der entsprechend an der gewünschten Stelle eingefügt werden muss.

Die Aufteilung der Logik fällt bei Angular zudem anders als bei Vaadin aus, da die einzelnen Komponenten vermehrt selbst Logik beinhalten. Das liegt primär daran, dass die Komponenten untereinander weitestgehend frei agieren und referenziert werden können. Es gibt zwar einige, übergeordnete Steuerungsele-

mente, diese fallen wegen der fehlenden Klassenhierarchie jedoch stellenweise deutlich weniger ins Gewicht.

Das Auftrennen in verschiedene, wiederverwendbare Komponenten fällt am Beispiel der *Echtzeit Eingangskontrolle* dennoch ähnlich wie bei Vaadin aus, wie Abbildung 5.6 auf Seite 43 zeigt. Die benötigten (HTML) Elemente werden dabei in einer HTML-Datei einer Komponente zusammengeführt und passend angeordnet und können dann samt der zugehörigen Logik und dem Styling genutzt werden.

Hervorzuheben ist hierbei jedoch, dass die von Angular Material bereitgestellten *UI components* in vielen Fällen ergänzend zu nativen HTML Elementen eingesetzt werden und deren Funktionalität lediglich erweitern, nicht jedoch komplett ersetzen. Als Beispiel dafür kann die in rot eingezeichnete Tabelle herangenommen werden. Diese entspricht einem regulären HTML `table` Tag, welcher zusätzlich mit Material Attributen versehen wurde (`mat-table matSort [dataSource]`).

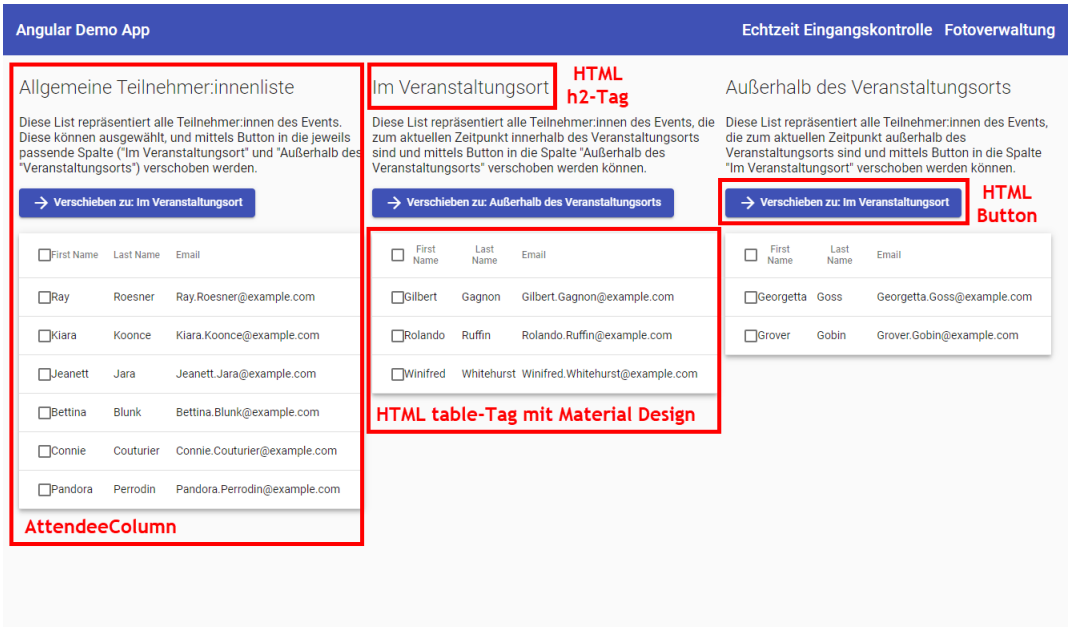

Abbildung 5.6.: Komponenten der *Echtzeit Eingangskontrolle* bei Angular

Wie bei der Umsetzung der Vaadin Anwendung, wurde auch bei der Entwicklung der Angular Demo auf den Einsatz von entkoppelten Web Components ver-

zichten und primär auf Komponenten gesetzt, die im projektspezifische Kontext wiederverwendet werden können.

6. Zusammenfassung und Ausblick

TODO

Literatur

A., Sviatoslav (Jan. 2020). *The Best JS Frameworks for Front End*. URL: https://rubygarage.org/blog/best-javascript-frameworks-for-front-end (besucht am 08. 02. 2021).

Google LLC (2021a). *Angular - In-app navigation: routing to views*. Englisch. URL: https://angular.io/guide/router#in-app-navigation-routing-to-views (besucht am 06. 03. 2021).

– (2021b). *Communicating with backend services using HTTP*. Englisch. URL: https://angular.io/guide/http (besucht am 24. 02. 2021).

Ismail, Kaya (Dez. 2019). *Why Single Page Apps Are the Hottest Trend of 2020*. Englisch. URL: https://www.cmswire.com/digital-experience/why-single-page-apps-are-the-hottest-trend-of-2020/ (besucht am 08. 02. 2021).

Mikowski, Michael und Josh Powell (Sep. 2013). *Single Page Web Applications: JavaScript end-to-end*. Englisch. 1st Edition. Shelter Island, NY: Manning Publications. ISBN: 978-1-61729-075-6.

Mozilla Contributors (Feb. 2021a). *Using custom elements - Web Components | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements (besucht am 02. 03. 2021).

– (Jan. 2021b). *Web Components | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components (besucht am 02. 03. 2021).

Scott, Emmit (Nov. 2015). *SPA Design and Architecture: Understanding Single Page Web Applications*. Englisch. 1st Edition. Shelter Island, NY: Manning Publications. ISBN: 978-1-61729-243-9.

Signell, Artur (Juni 2018). *Explanation of Flow's 2-way communication protocol? | 1. Vaadin Flow - Java | Forum*. Englisch. URL: https://vaadin.com/forum/thread/17118150/17119448 (besucht am 23. 02. 2021).

Vaadin Ltd (Jan. 2021a). *Navigating Between Routes | Routing and Navigation | Flow | Vaadin Docs*. Egnlisch. URL: https://vaadin.com/docs/latest/flow/routing/navigation (besucht am 09. 03. 2021).
– (Jan. 2021b). *Navigation Lifecycle | Routing and Navigation | Flow | Vaadin Docs*. Englisch. URL: https://vaadin.com/docs/latest/flow/routing/lifecycle (besucht am 08. 03. 2021).
– (März 2021c). *Overview | Application Basics | Flow | Vaadin Docs*. Englisch. URL: https://vaadin.com/docs/latest/flow/application/overview (besucht am 03. 03. 2021).
– (Feb. 2021d). *Overview | Flow | Vaadin Docs*. Englisch. URL: https://vaadin.com/docs/latest/flow/overview (besucht am 22. 02. 2021).
– (Jan. 2021e). *Overview | Routing and Navigation | Flow | Vaadin Docs*. Englisch. URL: https://vaadin.com/docs/latest/flow/routing/overview (besucht am 08. 03. 2021).
– (Jan. 2021f). *Query Parameters | Routing and Navigation | Flow | Vaadin Docs*. Englisch. URL: https://vaadin.com/docs/latest/flow/routing/query-parameters (besucht am 08. 03. 2021).
– (Jan. 2021g). *Typed URL Parameters | Routing and Navigation | Flow | Vaadin Docs*. Englisch. URL: https://vaadin.com/docs/latest/flow/routing/url-parameters (besucht am 09. 03. 2021).
– (Feb. 2021h). *vaadin-text-field.js*. GitHub. URL: https://github.com/vaadin/vaadin-text-field/blob/v3.0.1/src/vaadin-text-field.js (besucht am 02. 03. 2021).
– (2021i). *Documentation | Vaadin 18 Docs*. Englisch. URL: https://vaadin.com/docs/v18/ (besucht am 22. 02. 2021).
– (2021j). *Java Examples | Text Field | Components*. Englisch. URL: https://vaadin.com/components/vaadin-text-field/java-examples/text-field (besucht am 02. 03. 2021).
– (2021k). *Mobile optimized UI components for your web app*. Englisch. URL: https://vaadin.com/components (besucht am 02. 03. 2021).
– (2021l). *Vaadin Fact Sheet: Components*. Englisch. URL: https://v.vaadin.com/hubfs/Pdfs/Vaadin-components-fact-sheet.pdf (besucht am 02. 03. 2021).

Vaadin Ltd (2021m). *Vaadin Flow - Modern web apps in Java*. Englisch. URL: `https://vaadin.com/flow` (besucht am 22. 02. 2021).

– (2021n). *Vaadin Fusion*. Englisch. URL: `https://vaadin.com/fusion` (besucht am 22. 02. 2021).

Wilken, Jeremy (Apr. 2018). *Angular in Action*. Englisch. 1st Edition. Shelter Island, NY: Manning Publications. ISBN: 978-1-61729-331-3.

A. `EntranceControlView` Klasse

```
1  [...]
2
3  @Route(value = "entrance-control/:eventId", layout = MainView.class)
4  @PageTitle("Echtzeit Eingangskontrolle")
5  @CssImport("./styles/views/entrance-control/entrance-control-view.css")
6  public class EntranceControlView extends HorizontalLayout implements
   ↪  BeforeEnterObserver {
7
8  private final Grid<Attendee> defaultGrid = new Grid<>(Attendee.class);
9
10 private final Button moveToEnteredBtn = new Button(/* Translation */);
11
12 public EntranceControlView([...]) {
13     [...]
14
15     // Components & component configuration
16     add(prepareDefaultColumn(), prepareEnteredColumn(),
       ↪  prepareExitedColumn());
17 }
18
19 private VerticalLayout prepareDefaultColumn() {
20     return new AttendeesColumn(
21         getTranslation("entrance-control.default-grid.heading"),
22         getTranslation("entrance-control.default-grid.explanation"),
23         defaultGrid,
24         "default-attendees-grid",
25         moveToEnteredBtn
26     );
27 }
28
29 [...]
```

Quellcode 13: Ausschnitt der `EntranceControlView` Klasse, der einen Teil der Komponentenkonfiguration aufzeigt

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit II selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 20. Mai 2021 Dominic Luidold