

**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, werden 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 wecken beziehungsweise zu stärken.



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>viii</b>
<b>Abkürzungsverzeichnis</b>	<b>ix</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problemstellung . . . . .	2
1.3 Zielsetzung . . . . .	3
<b>2 Stand der Technik</b>	<b>5</b>
2.1 Konzept einer Single-page Application . . . . .	5
2.1.1 Server-side rendering . . . . .	6
2.1.2 Bestandteile einer SPA . . . . .	7
2.2 Vorteile einer SPA gegenüber einer MPA . . . . .	8
2.3 Funktionsweise von Vaadin . . . . .	9
2.3.1 Ansätze von Vaadin . . . . .	9
2.3.2 Frontend und Backend in Java . . . . .	10
2.3.3 Kommunikation zwischen Client und Server . . . . .	11
2.3.4 Vaadin Components und deren Grundlage . . . . .	15
2.3.5 Routing und Navigation . . . . .	17
<b>Literaturverzeichnis</b>	<b>22</b>
<b>Eidesstattliche Erklärung</b>	<b>24</b>

# Abbildungsverzeichnis

1.1	Liste möglicher JavaScript-Frameworks zur Umsetzung von Single-page Applications . . . . .	2
2.1	Aufbau einer Single-page Application . . . . .	6
2.2	Single-page Application Shell . . . . .	7
2.3	Automatisch erzeugte Kommunikation von Vaadin Flow . . . . .	13
2.4	Auswahl von Open Source Vaadin Compoments . . . . .	15
2.5	Erzeugter Output von Quellcode 5 . . . . .	17



# Abkürzungsverzeichnis

**AJAX** Asynchronous JavaScript and XML

**API** Application Programming Interface

**DOM** Document Object Model

**JSON** JavaScript Object Notation

**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*<sup>1</sup> und *Vaadin*<sup>2</sup> zu geben und deren Gemeinsamkeiten, Unterschiede sowie Vor- und Nachteile zu beleuchten.

Um ein grundlegendes Verständnis über die Thematik von Single-page Applications 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 wird 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 Multi-page Applications - setzen. 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.

---

<sup>1</sup>Angular (<https://angular.io>)

<sup>2</sup>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 Single-page Applications 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 auf Seite 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 jedoch dazu, dass sich meist nur schwer beurteilen lässt, welches Framework sich für die Umsetzung einer Applikation bestmöglich eignet.

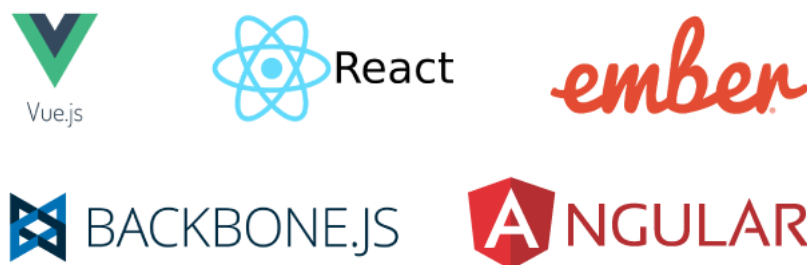


Abbildung 1.1: Liste möglicher JavaScript-Frameworks zur Umsetzung von Single-page Applications (Quelle: A. 2020)

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

müssen. Neben grundlegenden Funktionalitäten, die in den meisten Fällen von einer Vielzahl der Frameworks abgedeckt werden können, stellen sich die projektspezifischen Eigenheiten und vor allem die Auswahl der zugrundeliegenden Technologien als 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. Single-page Applications lassen sich jedoch nicht nur Frontend-seitig entwickeln (bei denen ein Großteil der Logik auf einem externen Server abläuft), sondern können ebenfalls mittels auf *Java* basierenden Frameworks umgesetzt werden. Bei diesen Frameworks - zu denen unter anderem Vaadin gehört - lässt sich sowohl die Logik als auch das User Interface (UI), stellenweise gänzlich, kombinieren.

Da die unterschiedlichen Ansätze, sowohl hinter Vaadin als auch 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 Single-page Applications 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 Herangehensweisen ist eine bedachte Wahl wichtig.

Diese Arbeit verfolgt daher das Ziel, das JavaScript Framework *Angular* dem auf Java basierenden Framework *Vaadin* gegenüberzustellen und zu vergleichen. Das Ziel ist es, mittels Literatur belegter Vergleiche einen allgemeinen Überblick über Single-page Applications 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 kann.

Um den Fokus dieser Arbeit genauer zu definieren und einzuschränken, wird die Planung, Umsetzung sowie abschließenden 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 Single-page Application und vergleicht das Konzept von SPAs mit klassischen Multi-page Applications. Im Anschluss wird im Detail auf die Funktionsweise von Vaadin und Angular beziehungsweise deren unterschiedlichen Ansätze in Hinblick auf Entwicklung der UI mittels JavaScript und Java eingegangen. Im weiteren Verlauf werden die damit verbundenen Vor- und Nachteile genauer beleuchtet.

### 2.1 Konzept einer Single-page Application

Eine klassische Multi-page Application 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 Single-page Application 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 keine komplett neue HTML-Seite geladen werden muss, sondern lediglich Teile

des User Interface - sogenannte *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 Single-page Application  
(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



Ausschnitte 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 Single-page Application - 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 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)

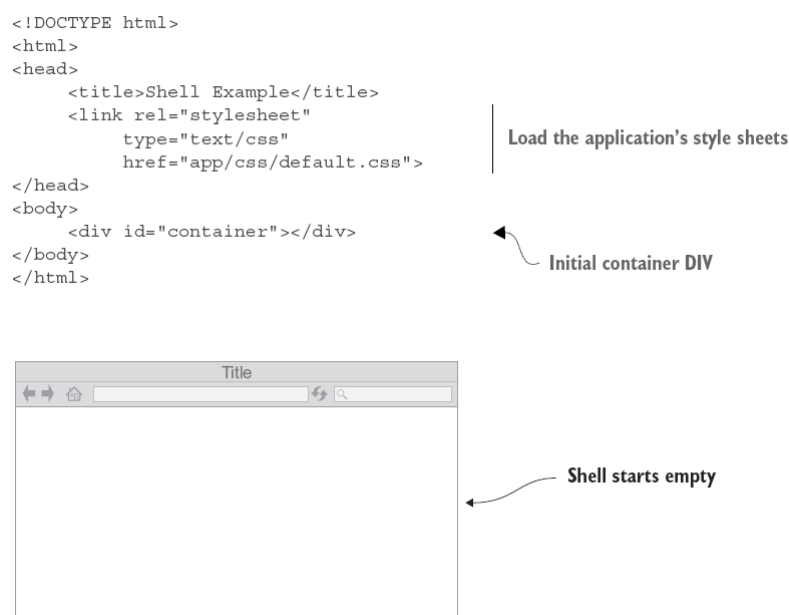


Abbildung 2.2: Single-page Application Shell  
(Quelle: Scott 2015, Seite 8)

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 Single-page Application bietet sowohl Vorteile für Entwickler:innen als auch Anwender:innen gegenüber der Nutzung einer klassischen Multi-page Application.

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 ect. mit serverseitigem Code (beispielsweise *PHP*, *JavaServer Pages*, ...) vermischt werden, kann bei SPAs zudem eine gewisse Differenzierung und Abtrennung von HTML, 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 Single-page Application effizienter und schneller stattfinden kann, als bei einer Multi-page Application. 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 Single-page Applications 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 2021f)

- **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 typischerweise 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 2021e)

Diese Arbeit befasst sich nachfolgend primär mit *Vaadin Flow*, um eine alternative Herangehensweise aufzuzeigen, wie eine Single-page Application 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 2021b, Framework - Introduction - Overview)

Vaadin Flow bietet daher die Möglichkeit, mit einer vollständig in Java geschriebenen Applikation - die auf einem Server ausgeführt 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 2021b, 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 2021b, Framework - Introduction - Overview)

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 unterstützt das mitgelieferte *Two-way data binding* die Entwicklung und Benutzbarkeit, indem Änderungen auf dem Client automatisch auch auf dem Server - und umgekehrt - abgebildet werden. (Vaadin Ltd 2021b, Framework - Introduction - Overview)

### 2.3.3 Kommunikation zwischen Client und Server

Bei einer Single-page Application spielt die Kommunikation des Clients (dem JavaScript/TypeScript Frontend) und dem Server (dem Backend) eine sehr

wichtige Rolle. Im Gegensatz zu Multi-page Applications, 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 Herangehensweise von Vaadin

Da sowohl die Persistenzschicht, 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 Document Object Model 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 2021b, Framework - Introduction - Core Concepts)

Artur Signell, CTO der Vaadin Ltd., erklärt währenddessen in einem Foren-Beitrag vom Juni 2018, dass genauere Informationen zur tatsächlichen Umsetzung der automatisch vorgenommenen Kommunikation nicht vorhanden ist, 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.

▼ Anforderungsnutzlast    Quelle anzeigen

```

▼ {csrfToken: "6d6d098f-04bf-4d5a-b05a-1c81201cddc4",...}
  clientId: 3
  csrfToken: "6d6d098f-04bf-4d5a-b05a-1c81201cddc4"
▼ rpc: [{type: "mSync", node: 6, feature: 1, property: "value", value: "test"},...]
  ▼ 0: {type: "mSync", node: 6, feature: 1, property: "value", value: "test"}
    feature: 1
    node: 6
    property: "value"
    type: "mSync"
    value: "test"
  ▼ 1: {type: "event", node: 6, event: "change", data: {}}
    data: {}
    event: "change"
    node: 6
    type: "event"
  syncId: 3

```

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

### 2.3.3.2 Herangehensweise von 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.

```
1  [...]
2
3  @Injectable()
4  export class SampleService {
5      constructor(private http: HttpClient) {}
6
7      getSampleData() {
8          return this.http.get<Array<Sample>>(/* URL to server or API endpoint */);
9      }
10 }
11
12 [...]
```

#### 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.

```
1  [...]
2
3  private loadSampleData() {
4      this.sampleService.getSampleData().subscribe(data => {this.data = data});
5  }
6
7  [...]
```

#### 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 2021d, 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 2021c)

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 2021a) 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, wie bereits angesprochen, 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 2021c)

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 beliebige

ger Stelle genutzt werden. Weiterführende beziehungsweise benötigte Funktionalitäten 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 2021c, „Text Field“ - Examples)

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

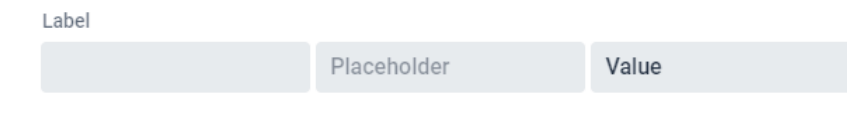


Abbildung 2.5: Erzeugter Output von Quellcode 5  
(Quelle: Vaadin Ltd 2021c, „Text Field“ - Examples)

### 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. Multi-page Applications laden beim Aufruf einer Seite die hinterlegten HTML, CSS und JavaScript Dateien von einem Webserver

und nutzen dafür die angegebene URL. Single-page Applications 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 2021b, Framework - Routing and Navigation - Defining Routes with `@Route`)

```
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 2021b, Framework - Routing and Navigation - Defining Routes with `@Route`)

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, Framework - Routing and Navigation - Navigation Lifecycle)

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. Query Parameter lassen sich währenddessen mit dem `BeforeEvent` ebenfalls in der selben Methode abfragen. (Vaadin Ltd 2021b, Framework - Routing and Navigation - Routing and URL Parameters)

```
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 2021b, Framework - Routing and Navigation - Navigating Between Routes)
- 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 2021b, Framework - Routing and Navigation - Navigating Between Routes)

- Serverseitig kann mittels `UI.navigate(/* Route name */)` der Wechsel zu einer neuen Route ausgelöst werden. (Vaadin Ltd 2021b, Framework - Routing and Navigation - Navigating Between Routes)

### 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. 20 zeigt. (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 AppRoutingModuleModule) -->
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 Angulars  
(Quelle: Google LLC 2021a)

# 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).
- 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](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](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 (Feb. 2021a). *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).



- (2021b). *Documentation / Vaadin 18 Docs*. Englisch. URL: <https://vaadin.com/docs/v18/> (besucht am 22.02.2021).
- (2021c). *Mobile optimized UI components for your web app*. Englisch. URL: <https://vaadin.com/components> (besucht am 02.03.2021).
- (2021d). *Vaadin Fact Sheet: Components*. Englisch. URL: <https://v.vaadin.com/hubfs/Pdfs/Vaadin-components-fact-sheet.pdf> (besucht am 02.03.2021).
- (2021e). *Vaadin Flow - Modern web apps in Java*. Englisch. URL: <https://vaadin.com/flow> (besucht am 22.02.2021).
- (2021f). *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.

# 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