HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Fakultät Informatik und Medien

Studiengang Medieninformatik

Vergleichende Analyse der komponentenbasierten Frontend-Frameworks Angular und React

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

vorgelegt von

Felix Maximilian Schmeißer

geb. am 22. 02. 1999

in München

Matr. 69578

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. Klaus Hering
Leipzig, Juni 2020 – September 2020

Erklärung

Ich versichere wahrheitsgemäß, diese Arbeit selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

.....
Felix Maximilian Schmeißer
Leipzig, den 29. September 2020

Inhaltsverzeichnis

1 Einleitung

Problemstellung

Es gibt unzählige JavaScript-Frameworks und es kommen ständig neue hinzu. Als Entwickler muss abgewägt werden, welche Lösung für das speziell vorliegende Szenario geeignet ist.

Angular ist ein umfangreiches Frontend-Framework und kann damit nahezu jede Aufgabe in diesem Bereich abdecken. Für sehr viele Problemstellungen gibt es eine Lösung direkt aus dem Framework.

React als JavaScript-Bibliothek ist deutlich reduzierter. Abseits der elementaren Funktion von React werden externe Bibliotheken oder Frameworks verwendet. Demzufolge muss hier zwischen verschiedenen Möglichkeiten entschieden werden.

Ziel der Arbeit

Das Ziel der Arbeit ist, die Technologien umfassend einzuführen und daraus Gemeinsamkeiten und Unterschiede zu erkennen. Auch werden mögliche Vor- und Nachteile hinsichtlich der Architektur angeschnitten. Dazu wird im Rahmen der Arbeit eine Testanwendung mit beiden Frameworks implementiert. Das Ziel ist, die geeigneten Anwendungsszenarien der Technologien daraus abzuleiten.

Vorgehensweise

Zunächst erfolgt eine allgemeine Einführung mit der Entstehungsgeschichte, zugrundeliegenden Technologien und dem Aufbau neuer Projekte. Im Anschluss werden die Konzepte von Angular und React beschrieben. Danach wird der Vergleich praktisch durchgeführt. Ziel ist es, eine identische Anwendung sowohl in Angular als auch in React zu implementieren, um die Unterschiede zu verdeutlichen und Grenzen aufzuzeigen. Abschließend werden die Beobachtungen eingeordnet und Schlüsse über Vor- und Nachteile beider Technologien gezogen.

Abgrenzung

Die Arbeit gibt eine fundierte Übersicht über die Grundprinzipien von Angular und React. Eine komplette Abhandlung ist nicht möglich und für die Beantwortung der zentralen Fragestellung auch nicht notwendig.

Bibliothek und Framework

Ein Framework gibt einen Rahmen für die Entwicklung vor und bestimmt die Architektur der Anwendung. Einige Frameworks liefern außerdem fertige Komponenten und Klassen, sondern auch Laufzeitumgebungen und Interfaces zum Erstellen neuer Bestandteile, Testsysteme und Auslieferung der Anwendung als Produktionsversion.

Eine Bibliothek ist eine Sammlung von einzelnen Programmteilen wie Klassen oder Komponenten, Routinen oder ganzen Programmen, die für die Entwicklung genutzt

werden können und die Arbeit unterstützen. Im Gegensatz zu Frameworks definieren Sie nicht den Rahmen der Anwendung, sondern sind eigenständige Bausteine.

Angular ist ein Webframework, React eine Bibliothek. Im Folgenden werden beide Technologien zur Vereinfachung als Framework zusammengefasst.

2 Allgemeines

2.1 Entstehung

2.1.1 Angular

Die Historie von Angular beginnt 2009 als Nebenprojekt der Entwickler Miško Hevery und Adam Abrons. Hevery störte die notwendige Menge an Boilerplate für Datenbanken, Datenbankzugriff und Sicherheitsvorkehrungen. Die Idee war, diese Bestandteile zu abstrahieren und ein Framework zu entwickeln, dass von Designern ohne fortgeschrittene Programmierkenntnisse verwendet werden kann.

AngularJS

Hevery arbeitete zu dieser Zeit bei Google mit 2 weiteren Entwicklern an Google Feedback. Die Entwicklung wurde immer schwerfälliger, woraufhin er das Projekt mit dem eigenen Framework in 3 Wochen neu schrieb und den Quelltext um 90% reduzieren konnte. Google begann, immer mehr interne Projekte mit dem in AngularJS benannten Framework umzusetzen.

Im Oktober 2010 wurde AngularJS mit der Versionsnummer 0.9.0 als erste stabile Version auf GitHub veröffentlicht, im Juni 2012 Version 1.0.0. Das Framework war durch innovative Ansätze von Beginn an erfolgreich, dazu gehören:

- Dependency Injection

- wiederverwendbare HTML-Bausteine
- zweiseitige Datenbindung (hält Model und View konsistent)
- Single-Page-Applikation (einzelnes HTML-Dokument lädt Inhalte dynamisch)

Die Version 1.7.0 im Mai 2018 führte als letzter Release zu vorherigen Versionen inkompatible Änderungen ein. Bis Juli 2021 wird AngularJS im Long Time Support unterstützt. Es werden lediglich Sicherheitsfehler und durch neue Versionen der gängigen Browser erzwungene Bugs behoben.[1][2]

Angular 2

2016 veröffentlichte Google den Nachfolger von AngularJS unter dem verkürzten Namen Angular, von Grund auf neu geschrieben und nicht abwärtskompatibel. Die neue Version beinhaltet die JavaScript-Obermenge TypeScript und verzichtet auf den Support älterer Browser. Die Template Syntax ist weiter verbessert worden, die Kapselung in Modulen und dynamisches Nachladen von Komponenten kommt hinzu. [3]

2.1.2 React

Die Entstehungsgeschichte von React ähnelt der von Angular. In 2011 standen Entwickler der Facebook Ads App vor einigen Problemen. Neue Features ließen sich mit der existierenden Codebase immer schwieriger umsetzen und eine grundlegende Überarbeitung war notwendig, um weiter entwickeln zu können. Aus diesem Umstand heraus entwickelte Jordan Walke einen ersten Prototypen namens FaxJs[1]. Dieser wurde in der folgenden Zeit auch eingesetzt und mit dem Namen React weiterentwickelt. Mit der Übernahme von Instagram

[1] siehe: https://github.com/jordwalke/FaxJs

durch Facebook gab es Gründe für eine Open-Source-Veröffentlichung der Bibliothek. Ende Mai 2013 stellte Walke React im Rahmen der JS ConfUS der Öffentlichkeit vor. In den folgenden Jahren verbreitete sich React weiter. Anfang 2015 begannen große Unternehmen wie Netflix und Airbnb ihre Anwendung mit dem Framework umzusetzen.[4]

2.2 Verwendete Sprachen

In den folgenden Abschnitten werden die von Angular und React angewendeten Webtechnologien eingeführt. Die Angular Template Syntax und JSX sind spezifisch für Angular respektive React entwickelte Syntaxerweiterungen für HTML und JavaScript.

2.2.1 Angular Template Syntax

Angular führt eine spezielle Syntax ein, die den aktuellen HTML5-Standard um einige Funktionen erweitert. Das `script`-Tag wird mit einer Warnung ignoriert, um Injection-Angriffe zu vermeiden. Sonst sind alle bekannten Tags des Standards erlaubt. Im folgenden Abschnitt werden einige wichtige Syntaxerweiterungen vorgestellt.

Interpolation und Ausdrücke

Mit Interpolation lassen sich Template Expressions (Ausdrücke) im HTML-Template verwenden. Angular wertet die Ausdrücke aus und ersetzt sie im finalen Template durch einen String. Rechenoperationen oder Funktionsaufrufe sind erlaubt, das Ergebnis muss aber als String ausgewertet werden können.

Die Ausdrücke können neben Interpolationen auch an HTML-Attribute gebunden werden: `<img [src]="imageSourceUrl">`.

```
<h1>Benutzername: {{ userName }}</h1>
<!-- mit userName = 'Max' ... -->
<h1>Benutzername: Max</h1>

<span>Zufallszahl: {{ getRandomNumber() }}</span>
<!-- mit getRandomNumber -> 23 -->
<span>Zufallszahl: 23</span>
```

Grundsätzlich ist beliebiger JavaScript-Code erlaubt. Nebeneffekte sollen vermieden werden, dadurch fallen Zuweisungen (`=`, `+=`, `-=,`), einige Schlüsselwörter (`new`, `typeof`, `instanceof`), In- und Dekrementierung (`++`, `--`) und einige weitere Befehle ab dem ES2015-Standard weg. Auch Bitoperationen sind verboten, neu eingeführt werden der Pipe-Operator (`|`) zum Verwenden von Pipes (mehr dazu unter 3.2.1) und Null-Überprüfungen mit `!` oder `?`.

Funktionen können weiterhin Nebeneffekte verursachen. Die Ausdrücke stehen dabei immer im Kontext der aktuellen Komponente und greifen folglich auf Eigenschaften der verknüpften TypeScript-Klasse zu. Die Datenbindung ist einseitig von Model zu View. Mehr dazu in Kapitel 2.

Anweisungen

Das Gegenstück Template Expressions sind Template Statements (Anweisungen). Die Datenbindung verhält sich identisch. Anweisungen werden ausgeführt, wenn das korrespondierende Event aufgerufen wird, z.B. ein Klick-Event:

```
<button (click)="login()">Einloggen</button>
```

Anweisungen erlauben, wie Ausdrücke, beliebiges JavaScript. Allerdings sind Nebeneffekte in diesem Fall der zentrale Punkt, um entsprechende Datenänderungen oder Navigation anzustoßen. Zuweisungen (operative wie `+=` oder `-=` ausgenommen) und Schlüsselwörter außer `new` sind erlaubt. Der Kontext wird um den Kontext des Templates selbst erweitert. Anweisungen können folglich auch auf `$event`-Objekte und im Template definierte Variablen zugreifen. Im folgenden Beispiel wird der Variable `adress` eine Referenz auf ein Input-Element zugewiesen:

```
<input #address placeholder="Adresse" />
```

Banana In A Box

Datenbindung in Angular kann auch zweiseitig stattfinden: Wird die View verändert, etwa durch Eingabeevents, dann wird das Model aktualisiert. Das gilt umgekehrt für Änderungen des Models. Ausdrücke werden dazu in eckige und runde Klammern geschrieben, daher die Bezeichnung „Banana In A Box“.

```
<input [(ngModel)]="username">.
```

2.2.2 JSX

JSX ist eine Syntaxerweiterung für JavaScript und ermöglicht die Verwendung von HTML innerhalb von JavaScript. Die HTML-Elemente werden am Ende zu vollständigem JavaScript kompiliert (Listing 2.1).

JSX wertet wie Angulars Template Syntax Ausdrücke aus, allerdings wird nur eine geschweifte Klammer um den Ausdruck gesetzt: `<h1>Hallo  user.name !</h1>`. String-Literale können auch in Anführungszechen gesetzt werden. Der volle Funktionsumfang

Listing 2.1: Äquivalente Syntax in JSX

```
const element = (
  <h1 className="hello">
    Hallo Welt!
  </h1>
);

const element = React.createElement(
  'h1',
  {className: 'hello'},
  'Hallo Welt!'
);
```

Listing 2.2: HTML-Templates mit JavaScript

```
  function greet(user) {
  if (user) {
    return <h1>Hallo {user.name}!</h1>;
  }
  return <h1>Hallo Gast!</h1>;
}
```

von JavaScript bleibt erhalten, somit können `if`-Statements und `for`-Schleifen verwendet werden (Listing 2.2)

Möglicherweise schädliche Inhalte wie Nutzereingaben können problemlos in JSX eingebettet werden. Vor dem Rendern im React DOM werden die Ausdrücke in Strings konvertiert und spezielle Character umgewandelt (z.B. & zu &amp). Das verhindert XSS-Angriffe (Cross-site scripting).

2.2.3 JavaScript

JavaScript (JS) ist die Programmiersprache des Internets und wurde ursprünglich im Jahr 1995 von Brendan Eich entwickelt. Alle gängigen Browser besitzen eine Engine zum

Listing 2.3: Laufzeit- und Typfehler

```
// Dynamische Typisierung
var sampleNumber = 123;
splitStringCharacter(sampleNumber); // -> Laufzeit-Fehler

// Statische Typisierung
string sampleString = 123; // -> Compiler-Fehler
number sampleNumber = 123;
splitStringCharacter(sampleNumber) // -> Compiler-Fehler
```

Kompilieren von JavaScript-Code, Google Chrome setzt beispielsweise auf das eigene Open-Source-Projekt V8. JavaScript wird unter der Bezeichnung ECMAScript standardisiert.

Type Checking

JavaScript ist eine dynamisch typisierte Sprache. Typen werden zur Laufzeit bestimmt und müssen nicht angegeben werden. Damit geht die Programmierung nur vermeintlich schneller, denn Typfehler werden erst zur Laufzeit erkannt, auch wenn der Code vorher kompiliert werden konnte. Statisch typisierte Sprachen fordern die Angabe von Typen und kompilieren bei Typ-Fehlern nicht. TypeScript und Flow sind 2 Varianten, JavaScript typsicher zu verwenden.

TypeScript

Angular forciert die Nutzung des JavaScript-Supersets TypeScript (TS), ebenfalls eine Implementierung des ECMAScript-Standards, entwickelt von Microsoft. Es erweitert JavaScript mit zusätzlichen Features. JavaScript-Code ist valides TypeScript, nicht umgekehrt. TS wird mit seinem Compiler in gültigen JavaScript-Quelltext kompiliert.

Die neuen Features von TS kennen Entwickler aus der objektorientierten Programmierung. TS setzt, wie der Name schon andeutet, auf strengere Typisierung. Zwar gehen die Typings nach der Übersetzung verloren, allerdings werden Laufzeitfehler durch die vorgeschobene Kontrolle bereits bei der Entwicklung mit TypeScript reduziert.[5]

Flow

React legt sich nicht auf einen Type Checker fest. In der Dokumentation wird sowohl die Installation von TypeScript, als auch Flow beschrieben. Flow ist keine eigenständige Sprache mit eigenem Compiler wie TypeScript, sondern ermöglicht Type Checks durch Annotationen. Mit dem Befehl `flow` werden markierte JavaScript-Dateien überprüft. Flow wird wie React von Facebook entwickelt.[6]

2.2.4 CSS

Mit CSS werden die mit HTML semantisch strukturierten Elemente formatiert, das Styling bleibt dadurch unabhängig von den Inhalten. CSS wird vom W3C standardisiert, Version 3 seit 2000 fortlaufend weiterentwickelt. Neue Versionen gibt es nur noch für einzelne CSS-Module, nicht für den gesamten Standard.[7]

Die Arbeitsweise mit CSS selbst ändert sich, anders als bei HTML und JavaScript, für React und Angular nicht. Entwickler können frei unter verschiedenen Präprozessoren und anderen Lösungen entscheiden. In HTML-Templates wird CSS mit kleinen Anpassungen auf gewohnte Art verwendet.

Angular

Angular erlaubt weiterhin die Nutzung von `class`-Selektoren für HTML-Elemente. Mit Class-Bindings[2] können Strings, String-Arrays oder Objekte angegeben werden. Die gewohnte Verknüpfung von HTML-Knoten mit CSS-Klassen ist also weiter gegeben: `<Example class="first second">...</Example>`. Class-Binding kann sich auch auf einzelne Klassen beziehen: `[class.menu-active]="isMenuActive"`. Inline-Styles sind auch in Angular erlaubt und mit dem Style-Binding gibt es erweiterte Möglichkeiten. Es kann direkt auf CSS-Attribute zugreifen: `[style.width.px]="100"`. Auch auswertbare Ausdrücke sind erlaubt: `[style]="styleExpression"`.

Allerdings sind externe Stylesheets leichter zu pflegen, eine Vermischung mit Inline-Styles erschwert die Fehlersuche und bricht die Trennung von View-Content und Design.

React

In React werden Klassen mit dem className-Attribut zugewiesen:

```
<span className="headline-big highlighted">Headline</span>
```

Listing 2.4 zeigt, wie Styles in JSX auch dynamisch hinzugefügt werden können. Das `style`-Attribut bleibt erlaubt, allerdings gilt hier ebenfalls, dass solche Inline-Styles vermieden und eigene Stylesheets verwendet werden sollten[3].[8]

[2] siehe: https://angular.io/guide/attribute-binding#class-binding
[3] siehe: https://reactjs.org/docs/dom-elements.html#style

Listing 2.4: Dynamisches Styling mit JSX

```
render() {
  let className = 'button-save';
  if (!this.props.hasPermission) {
    className += 'button-deactivated';
  }
  return <button className={className}>Speichern</button>
}
```

Sass

Sass vereinfacht CSS und fügt Funktionalitäten hinzu, die die Les- und Wartbarkeit der Stylesheets verbessern, beispielsweise mit Mixins: CSS-Blöcke können als Variablen definiert und an anderen Stellen verwendet werden. Sass vereinfacht die Selektierung dadurch, dass sich Selektoren schachteln lassen. Sass wird als Präpozessor[4] bezeichnet, weil ein Compiler die erweiterte Syntax zu nativem CSS verarbeitet. Neben Sass gibt es noch weitere Präprozessoren wie LESS, Stylus oder PostCSS.

2.3 Projektaufbau

2.3.1 Projekterstellung

Angular CLI

Die Erstellung eines Angular Projektes erfolgt mit dem Angular CLI (Command Line Interface), welches mit dem Node Package Manager (NPM) installiert wird:

```
npm install -g @angular/cli
```

[4]siehe: https://developer.mozilla.org/de/docs/Glossary/CSS_Praeprozessor

Das CLI verfügt über Befehle, die z.B. externe Bibliotheken hinzufügen:

```
ng add <Bibliothek>
```

Neue Bestandteile (z.B. Components) anlegen:

```
ng generate <Schema>
```

Oder Tests starten:

```
ng test
```

Mit `ng new <Projektname>` wird ein neues Projekt in einem entsprechend benannten Ordner angelegt. Das Projekt ist hier schon startbereit, mit `ng serve -open` kann die Anwendung im Development Modus gestartet werden. Durch das Flag wird automatisch der Browser geöffnet (http://localhost:4200/.). Änderungen im Projekt werden erkannt und die Anwendung sofort aktualisiert.[9]

React Toolchains

React benötigt kein umfangreiches Setup. Die einfachste Möglichkeit der Einbindung besteht darin, React innerhalb des `<script>`-Tags zu verknüpfen. Damit lässt sich React zu bestehenden Projekten hinzufügen, um etwa eine schrittweise Migration zu ermöglichen. Für neue Projekte, die von Beginn an mit React entwickelt werden sollen, bietet sich eine Toolchain für das entsprechende Anwendungsszenario an. Damit wird die Umgebung für die React-Entwicklung optimiert: Häufig wird bereits eine Grundstruktur und Scripte zum Starten der Anwendung mit erweiterten Debugoptionen generiert, ähnlich dem Setup einer Angular-Applikation. Die einfachste Toolchain stammt ebenfalls von Facebook und nennt sich *create-react-app*. Mit dem Befehl `npx create-react-app <Projektname>` wie

beim Angular CLI ein Projektordner generiert. Durch `npm start` wird die Applikation gehostet (http://localhost:3000/).[10]

2.3.2 Dateistruktur

Die Frameworks haben nach der Erstellung mit den eben beschriebenen Tools einen ähnlichen Aufbau. Neben einem *src*-Order, der die Ressourcen der Anwendung beinhaltet, gibt es einige Konfigurationsdateien für JavaScript respektive TypeScript, außerdem eine Ordner (*node_modules*), der die eingebundenen Pakete beinhaltet. Das sind zunächst die Core-Funktionalitäten der Frameworks, gegebenenfalls kommen weitere Bibliotheken dazu (Angular[5], React[6])

Der Entwicklungsprozess beginnt bei beiden Frameworks mit der Root-Komponente, die beim Start eingebunden wird. Dort beginnt die Komponentenstruktur des Projektes. Für Angular ist das `app.component.ts` und das dazugehörige `app.module.ts`, bei React `App.jsx`.

Die Dateistruktur von Angular und React Anwendungen ist direkt abhängig von der Komponentenstruktur. Der Entwickler muss in der Lage sein, gesuchte Dateien schnell zu finden. Eine ordentliche Struktur ist nicht relevant für das Endprodukt, kann jedoch die Entwicklungsgeschwindigkeit beeinträchtigen. Für den jeweiligen Use-Case finden sich im Internet Lösungen, die React-Dokumentation[7] bietet einen grundsätzlichen Ansatz, der auch für Angular gelten kann.

[5]siehe: https://angular.io/guide/file-structure
[6]https://www.pluralsight.com/guides/file-structure-react-applications-created-create-react-app
[7]siehe: https://reactjs.org/docs/faq-structure.html

3 Technische Aspekte

Das folgende Kapitel behandelt den technischen Vergleich zwischen Angular und React. Zunächst werden grundlegende Entwurfsmuster und Modelle erläutert, die relevant für die weiteren Betrachtungen sind. Das Kapitel dient als Vorbereitung auf die noch folgende Implementation (Kapitel 4), die dort relevanten Bestandteile der Frameworks werden hier erläutert.

3.1 Grundlegende Schnittstellen und Entwurfsmuster

3.1.1 Komponenten

2004 erscheint im Blog des Entwicklers und Architekturspezialisten Martin Fowler ein Artikel mit dem Titel „Inversion of Control Containers and the Dependency Injection pattern“. In diesem Artikel beschreibt er unter Anderem das Entwurfsmuster Dependency Injection (DI) als präziseren Begriff für Inversion of Control. Die Erkenntnisse aus dieser Betrachtung definieren grundlegende Entwurfsmuster von Angular, die in den folgenden Abschnitten erläutert werden. React ist nicht so stark an diese Konzepte gebunden.

Komponenten sind in der Terminologie nicht eineindeutig. Martin Fowler beschreibt seine Einordnung wie folgt:

„I use component to mean a glob of software that’s intended to be used, without change, by an application that is out of the control of the writers of the component. By ’without change’ I mean that the using application doesn’t change the source code of the components,

although they may alter the component's behavior by extending it in ways allowed by the component writers."[11]

Mit diesem Ansatz können Komponenten in React und Angular ebenfalls betrachtet werden. Eine Komponente ist in sich abgeschlossen und sollte von außen nicht verändert werden. Das Verhalten lässt sich ändern: Eine Beispielkomponente bietet eine Tabelle zum Anzeigen von Daten. Von außen lässt sich die Komponente bezüglich der Spalten, Zeilen, Styles und Inhalten in einem definierten Rahmen beeinflussen. Das Ziel ist wiederverwendbare Funktionalitäten mit erleichterter Wartbarkeit zu realisieren.

3.1.2 Dependency Injection

Das Entwurfsmuster Dependency Injection liefert einen Teil der Erklärung bereits mit seinem Namen: Abhängigkeiten werden „injiziert". Die Tabellenkomponente dient in Listing 3.1 als Beispiel.

Das finder-Objekt liefert die benötigten Daten für die Tabelle. Damit muss sich die `getFilteredData`-Methode nicht die Beschaffung der Entry-Objekte kümmern, das liegt in der Implementation der `Finder`-Klasse. Mit einem Interface kann die Kapselung noch klarer gemacht werden, dennoch ist eine konkrete Implementation notwendig (siehe Konstruktor). Diese ändert sich, abhängig von der konkreten Speicherung unserer Daten, z.B. im JSON-Format oder als SQL-Datenbank. Die Komponente ist also idealerweise nur vom Interface abhängig.

3.1.3 Flux

Flux ist ein von Facebook im Zusammenhang mit React entwickeltes Architekturmuster zum Management des Anwendungs-Zustands. Es berücksichtigt den unidirektionalen

Listing 3.1: Dependency Injection

```
class TableComponent {
    private finder: EntryFinder;

    constructor(){
        this.finder = new EntryFinderImpl();
    }

    public getFilteredData(filter: string): Entry[] {
        let allEntries: Entry[] = this.finder.getAllEntries();
        let filteredEntries: Entry[] = allEntries.filter(entry =>
            entry.matches(filter));
        return (filteredEntries);
    }
}
interface EntryFinder{
    findAll(): Entry[];
}
```

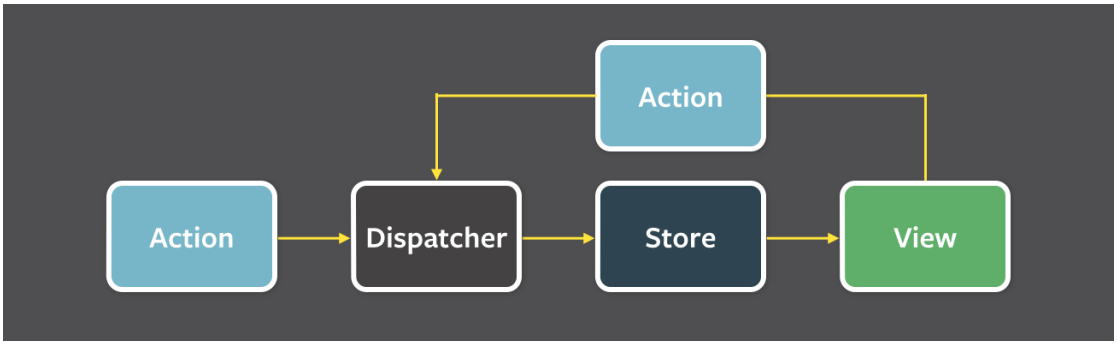
Datenfluss in React-Applikationen. Flux besteht aus 4 Teilen (siehe Abbildung 3.1).

Der Store hält den für das Frontend relevanten anwendungsübergreifenden State. Die Views erhalten aus diesem Store ihre notwendigen Daten. Durch vom Nutzer angestoßene Änderungen in der View stößt diese Actions an, die der Dispatcher entgegen nimmt. Der Dispatcher veranlasst dann die Änderung des globalen States im Store.[12]

3.1.4 Document Object Model

Das Document Object Model, kurz DOM, ist eine API, die den Zugriff und die Manipulation von HTML- und XML-Dokumenten erlaubt. Aus diesen Dokumenten lässt sich eine Baumstruktur ableiten.

Browser verarbeiten HTML-Dokumente in eine solche Struktur, der DOM repräsentiert



Quelle: https://facebook.github.io/flux/docs/in-depth-overview

Abbildung 3.1: Datenfluss mit Flux

den aktuellen Zustand der gezeigten Seite. Dynamische Änderungen der Seite, die mittels JavaScript umgesetzt werden, sind einfache Veränderungen des DOM über die Schnittstelle – beispielsweise mit den Funktionen `getElementById(...)` oder `removeChild(...)`.

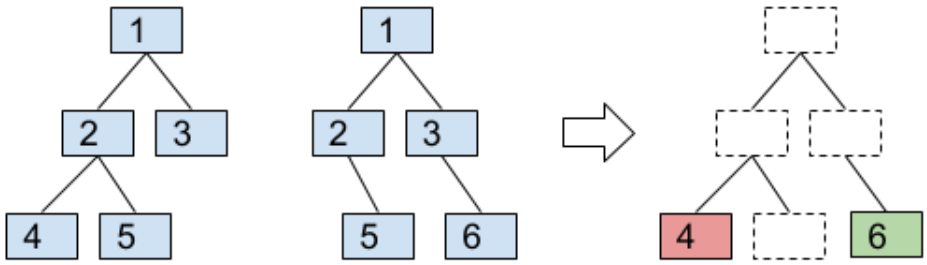
Virtual DOM

React verwendet einen virtuellen DOM als Repräsentation des aktuellen Zustandes einer Komponente (siehe Abbildung 3.2). Ändern sich dargestellte Werte oder Elemente, so muss der DOM neu aufgebaut und dargestellt (gerendert) werden. Gerade bei kleineren Änderungen ist es aber nicht nötig, den gesamten DOM neu zu rendern. Der virtuelle DOM wird mit dem echten DOM abgeglichen, daraufhin werden nur die veränderten Elemente und Kindelemente neu gerendert[13]. Mehr dazu im Abschnitt 3.3.2.

3.2 Angular

3.2.1 Komponenten und Direktiven

Angular-Applikationen besitzen mindestens eine Komponente. Diese besteht aus einer Klasse, die Daten und Programmlogik beinhaltet. Nach dem MVVM-Entwurfsmuster



Quelle: https://tinyurl.com/yyxtx766

Abbildung 3.2: Reacts Virtual DOM

Listing 3.2: Component-Dekorator

```
1 @Component({
2     selector: 'app-example',
3     templateUrl: './example.component.html',
4     styleUrls: ['./example.component.css'],
5     providers: [ ExampleService ],
6 })
7 export class ExampleComponent {
8     /* ... */
9 }
```

kann diese Klasse als das ViewModel bezeichnet werden. Die dazugehörige View ist ein HTML-Template, in welchem Daten für den Nutzer aufbereitet und dargestellt werden. Optional sind dazugehörige CSS-Styles, die bevorzugt in einem eigenen Stylesheet definiert werden. Komponenten beschreiben nur die Präsentation von Daten. Der Zugriff auf die Daten selbst sollte immer über einen Service stattfinden. Dadurch bleibt die Komponente unabhängig und einfacher testbar, etwa mit Mock-Up-Daten.

Listing 3.2 zeigt, wie die Klasse mit einem Dekorator[1] versehen wird, der für Angular notwendige Metadaten bereitstellt.

[1] siehe: www.typescriptlang.org/docs/handbook/decorators.html

Listing 3.3: ngFor und ngIf

```
<span *ngFor="let number of numbers"> {{ number}} </span>

<!-- erzeugt ... -->
<span> 1 </span>
<span> 2 </span>
<span> 3 </span>

<!-- order-details erscheint nur im DOM,
     wenn isPlanned wahr ist -->
<order-details *ngIf="order.isPlanned"></order-details->
```

Über den Dekorator können noch weitere Optionen angegeben werden. So lässt sich die Variante der Change Detection (siehe 3.2.4) deklarieren, HTML und CSS als Strings auch direkt angegeben und zu injizierende Klassen definiert werden, die dann der Komponente und den Kindelementen über Dependency Injection zur Verfügung stehen.[14][15]

Direktiven definieren Programmlogik ohne eine dazugehörige View. Komponenten sind damit ebenfalls Direktiven, allerdings mit dazugehörigen Templates, die im DOM sichtbar sind. Ein Beispiel für Direktiven sind `ngFor` und `ngIf`. Erstere iteriert über iterierbare Objekte wie Arrays und gibt für jedes enthaltene Element wiederum HTML-Elemente zurück, die Zweite zeigt das Element abhängig von einer Bedingung.

Komponenten können miteinander kommunizieren: Eigenschaften werden dazu mit einem Input- oder Output-Dekorator versehen. Inputs sind dabei beliebige Objekte, im folgenden Beispiel eine Movie-Objekt, das relevante Daten zu einem Film enthält. Outputs sind Event-Emitter, über die ein Handler registriert werden kann. Dieser erhält dann den definierten Wert. Bei Verwendung der Komponente werden Klammern zur Bindung der jeweiligen Objekte verwendet, eckige für Input, runde für Output.

Listing 3.4: In- und Output in Angular-Komponenten

```
// movie-list.component.html
<movie-card [movie]="movie" (selected)="onSelected($event)">
// movie-card.component.ts
class MovieCardComponent {
    @Input()  movie: Movie;
    @Output() selected = new EventEmitter<number>();
    /* ... */
    public selected(): void {
        this.selected.emit(this.movie.id);
    }
}
// movie-list.component.ts
class MovieListComponent {
    public onSelected(event: number) {
        // movieId verarbeiten ...
    }
}
```

Lifecycle

Der Lebenszyklus (Lifecycle) von Komponenten (und Direktiven) beginnt mit der Instanziierung der Komponenten-Klasse und dem Rendern der dazugehörigen View. Das gilt ebenso für alle Kindelemente. Die Change Detection erfasst Änderungen von Properties, die für die View relevant sind. Die Instanz wird gelöscht, wenn sie nicht weiter benötigt wird und die View aus dem DOM entfernt. Entwickler haben die Möglichkeit, in diesen Prozess einzugreifen, etwa um selbst Change Detection oder zusätzliche Aufräumarbeiten beim Löschen der Komponente zu initiieren. Das geschieht über Hooks, `ngOnInit()` wird beispielsweise direkt nach der Instanziierung der Komponente aufgerufen und eignet sich für das asynchrone Laden von Daten. Weitere Hooks sind z.B. `ngAfterViewInit()`, der Aufruf geschieht direkt nach der Initialisierung aller Views (auch Kindelemente) oder

`ngOnDestroy()`, wird vor der Löschung der Komponente ausgeführt.[16]

Services

Als Services werden in Angular spezielle Klassen bezeichnet, die sich um die Bereitstellung von Daten kümmern oder Programmlogik kapseln, die nicht in Komponenten gehört. Häufig werden mit Services Anwendungsdaten über das HTTP-Protocol von einem Server abgerufen, dort verändert oder abgespeichert. Angular liefert eine eigene HTTP API mit, welche diesen Zugriff ermöglicht. Ein Beispiel findet sich später in der Angular-Implementation (4.2.3). Ein anderes Beispiel wäre ein Service zum Loggen von definiertem Anwendungsverhalten, um etwa speziellen Zugriff zu dokumentieren oder die Entwicklung zu unterstützen.

Service-Klassen werden wie Komponenten und Direktiven mit einem Dekorator versehen und damit als solche markiert. Dieser nimmt lediglich eine Option entgegen: `providedIn`. Diese nimmt einen Injector für die Dependency Injection entgegen, üblicherweise 'root'. Damit steht der Service allen Komponenten zur Verfügung und kann wie soeben beschrieben über den Konstruktor eingebunden werden. Soll der Service nur in ausgewählten Modulen oder Komponenten nutzbar sein, dann muss der Service dort im Dekorator unter der Option `providers` deklariert werden. Services können wiederum andere Services einbinden.[17]

Pipes

Angular-Pipes basieren auf dem bekannten Prinzip von Pipes, welches ursprünglich für das Unix-Betriebssystem entwickelt wurde. In diesem Sinne ist eine Pipe(line) die Verbindung zwischen zwei datenverarbeitenden Prozessen, die Ausgabe des ersten wird zur Eingabe

Listing 3.5: AsyncPipe

```
<div *ngIf="orders$ | async as orders; else #noResults">
    <!-- Das Objekt 'orders' kann verwendet werden ... -->
</div>
<ng-template #noResults>
    <span> Keine Aufträge gefunden. </span>
</ng-template>
```

des zweiten Prozesses. Angular setzt auf dieses Grundprinzip. Mit Pipes sind allerdings transformierende Operationen gemeint, nach dem ursprünglichen Modell also der zweite Prozess selbst.

Pipes definieren eine `transform`-Funktion, die Parameter entgegen nimmt und einen transformierten Wert zurückgibt. Angular besitzt vordefinierte Pipes, unter anderem eine `DatePipe` zur Umwandlung von Daten in das lokale Zeitformat, eine `CurrencyPipe` zur Umwandlung von Zahlen in einen String mit der lokalen Währung oder eine `UpperCasePipe` zur Umwandlung von Text in Großbuchstaben. Pipes werden üblicherweise in Templates verwendet[18]:

```
<span>Auftragsdatum: {{ orderDate | date }}</span>
```

Komplexer ist die `AsyncPipe`, diese gibt den letzten emittierten Wert von Observables[2] oder Promises[3] durch eine Subscription[4] zurück. Ein häufiger Anwendungsfall ist die Einbindung von Daten aus asynchronen Operationen (Listing 3.3).[19]

[2]siehe: https://rxjs-dev.firebaseapp.com/guide/observable
[3]siehe: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise
[4]siehe: https://rxjs-dev.firebaseapp.com/guide/subscription

3.2.2 Modules

Angular-Applikationen werden mit `NgModules` modular aufgebaut. Module fassen Komponenten, Services und sonstigen Quelltext zusammen, die ein gemeinsames Feature abbilden oder auf andere Art und Weise zusammen gehören. Beispiele: Das `BrowserModule` stellt die notwendige Infrastruktur bereit, um Angular Applikationen im Browser starten zu können und ist damit Standard. Um Direktiven wie `ngIf` oder `ngFor` zu nutzen, muss das `CommonModule` importiert werden. Services benötigen das `HttpClientModule` zur Kommunikation mit Servern. Die zwei zuerst genannten Module werden für alle Komponenten benötigt und daher direkt im `app.module` importiert. Jede Anwendung deklariert mindestens dieses Modul. Module sind ebenfalls Klassen, werden aber ausschließlich durch den `NgModule`-Dekorator beschrieben (Listing 3.6).[20]

3.2.3 Dependency Injection

Es gibt verschiedene Varianten der Dependency Injection, die von Angular verwendete Variante nennt sich Constructor Injection (Listing 3.7). Die Objekte, zu welchen eine Abhängigkeit der Komponente besteht, werden dem Konstruktor übergeben. In der Regel sind abhängige Objekte Services, es können allerdings auch Funktionen oder Werte injiziert werden.

Angular initialisiert einen `Injector`, der einen Container mit Dependency-Instanzen verwaltet. Über einen Provider wird dem Injektor mitgeteilt, wie eine Instanz erzeugt wird, üblicherweise sind das die Klassen der Services. Wenn Angular eine neue Komponente erzeugt, dann wird überprüft, welche Abhängigkeiten bestehen. Gibt es Abhängigkeiten ohne bestehende Instanz, dann werden diese erzeugt und der Komponente zur Verfügung gestellt. Erkennbar ist, dass DI auf dem Singleton-Entwurfsmuster aufbaut. Damit werden

Listing 3.6: NgModule

```
@NgModule({
  declarations: [
    // Komponenten, Direktiven und Pipes,
    // die zu diesem Modul gehören
    ExampleComponent,
    ExampleDirective,
    ExamplePipe
  ],
  imports: [
    // Andere Module, die den deklarierten Be-
    // standteilen des Moduls zur Verfügung stehen
    BrowserModule,
    SomeOtherModule
  ],
  providers: [
    // Mittels DI injizierbare Objekte
    ExampleService
  ],
  exports: [
    // Bestandteile des Moduls, die beim Import
    // des Moduls bereitgestellt werden
    ExampleComponent
  ]
})
export class ExampleModule { }
```

Listing 3.7: Verwendung von Services über Constructor Injection

```
export class ExampleComponent {
    constructor(
        private finderService: EntryFinderService,
        public printService: PrinterService
    ) {}
}
```

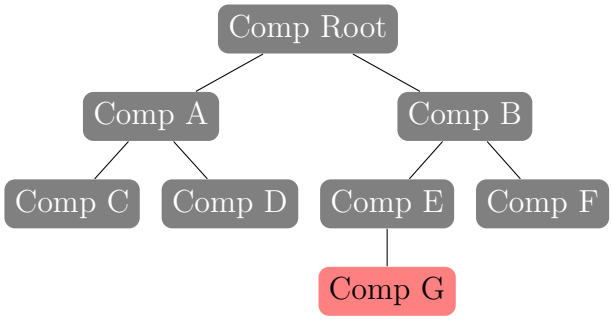


Abbildung 3.3: Komponentenbaum mit Veränderung in Komponente G

Inkonsistenz und redundante Objekte vermieden. Angulars DI bietet viele Möglichkeiten, die hier nicht in die Tiefe behandelt werden[5]. Relevant für die noch folgende Implementation ist die einfache Injektion von Services.[21]

3.2.4 Change Detection

Change Detection versucht Veränderungen des Anwendungszustandes zu erkennen und die View entsprechend zu updaten. In Angular geschieht das automatisch. Wird in einem Template ein Verweis auf Daten erzeugt (z.B. `{{user.name}}`) – sogenannte *Bindings* – dann beobachtet Angular dieses Element und prüft auf Wertveränderungen: Change Detection. Ausgelöst durch Browser-Events (Mausklick, Tastatureingaben), HTTP-Anfragen oder spezielle Funktionen wie `setInterval` oder `setInterval` wird der komplette Komponentenbaum abgetastet (siehe Abbildung 3.3)

Diese Variante kostet unter Umständen zu viel Zeit. Als Alternative kann daher die Strategie auf `onPush` geändert werden. Mit onPush markierte Komponenten werden nicht automatisch überprüft. Change Detection findet hier nur statt, wenn:

- Eingabewerte verändert werden, z.B. die Datenquelle einer Tabellenkomponente

[5]siehe: https://angular.io/guide/dependency-injection

- die Komponente oder Kindkomponenten spezielle Events auslösen, z.B. ein Button wird angeklickt
- sie manuell ausgelöst wird
- in Templates referenzierte Observables neue Werte ausliefern

Die Change Detection sollte vor allem bei größeren Anwendungen mit vielen Komponenten beachtet werden. Ein Wechsel auf die onPush-Strategie verbessert gegebenenfalls die Performance.[22]

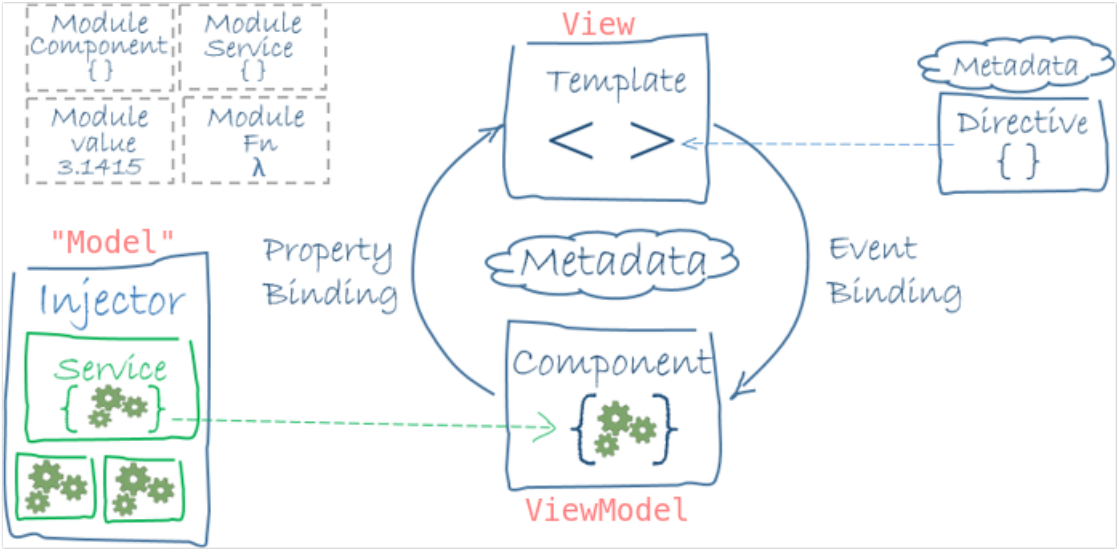
3.2.5 State-Management

State-Management kann in Angular auf verschiedenen Wegen umgesetzt werden. Über Input-Attribute kann übergreifender State im Komponentenbaum verteilt und bei Änderung mit Output-Events nach oben gereicht werden. Für komplexere Anwendungen gibt es besser Ansätze: Redux[6], eine auf Flux basierende Bibliothek, oder Observable Data Services[7].

3.2.6 Projektarchitektur

Angular gibt durch die beschriebenen Features Empfehlungen über den Aufbau der damit gebauten Anwendung und zu verwendende Entwurfsmuster ab. Die Dokumentation stützt sich auf diese Prinzipien, auch wenn sie nicht zwangsläufig befolgt werden müssen, um eine funktionsfähige Anwendung zu entwickeln. Angular empfiehlt, den Code in möglichst kleine Komponenten aufzuteilen. Jede Komponente übernimmt möglichst nur

[6]siehe: https://redux.js.org/
[7]siehe: https://coryrylan.com/blog/angular-observable-data-services



Quelle: https://angular.io/guide/architecture

Abbildung 3.4: Angular Architektur

eine Aufgabe. Komponenten die zu dem gleichen Teilbereich gehören werden als Modul zusammengefasst.

Einzelne Komponenten lassen sich mit dem MVVM-Entwurfsmuster vergleichen. Das Model sind einfache Klassen und Interfaces, welche die Daten der Anwendung modellieren, z.B. Nutzer, Auftrag oder Rezept. Models enthalten keine Programmlogik, die Ausnahme bildet die Validierung der Daten. Services werden von Komponenten genutzt, um auf Models zuzugreifen. Komponenten vereinen die View und das ViewModel aus MVVM: Die HTML-Templates mit dazugehörigen Styles beschreiben die Darstellung von Daten für den Endnutzer. Die TypeScript-Klassen sind das ViewModel, stehen also zwischen dem Model und der View. Die View greift auf das ViewModel zu und kann es gegebenenfalls modifizieren, etwa durch Nutzereingaben (Zweiseitige Datenbindung). Das Model wird vom ViewModel abgerufen und ebenfalls modifiziert, sollten die Änderungen in der View das erfordern.

Entwickler haben die Freiheit, wie konsequent sie die Trennung vornehmen. Services passen nur bedingt in das MVVM-Entwurfsmuster, sie stehen zwischen Model und ViewModel. Angulars Nähe zu MVVM ist erkennbar und hilft beim Verständnis. Am Ende definiert das Framework aber eigene Prinzipien und gibt dem Entwickler genügend Spielraum, wie er Programmlogik strukturieren möchte.[23][24]

3.3 React

3.3.1 Komponenten

Die Grundeinheiten sind die React Elements. Diese können gängige DOM-Tags:

```
const element = <div />;
```

oder deklarierte Komponenten enthalten:

```
const helloMax = <Greeting name="Max">;
```

Funktions- und Klassenkomponenten

Es gibt mehrere Möglichkeiten in React, Komponenten zu deklarieren. Das Konzept ähnelt Funktionen: Sie erhalten beliebigen Input über die sogenannten `props` und geben eine Beschreibung dessen, was auf dem Bildschirm angezeigt werden soll als React-Objekte zurück. In Listing 3.8 werden zwei prinzipiell äquivalente Varianten gezeigt[25]. Komponenten können wie React Elements andere Komponenten enthalten.

Darüber hinaus gibt es Higher-Order-Components[8]. Diese verarbeiten Komponenten als Argumente und geben wiederum Komponenten zurück. Mit Higher-Order-Components

[8]siehe: https://reactjs.org/docs/higher-order-components.html

Listing 3.8: Funktions- und Klassenkomponenten in React

```
// Komponente als Funktion
function Greeting(props) {
    return <h1>Guten Tag {props.name}!</h1>;
}

// Komponente als Klasse, implementiert die render-Funktion
class Greeting extends React.Component {
    render() {
        return <h1>Guten Tag {props.name}!</h1>;
    }
}
```

lässt sich Logik extrahieren und wiederverwenden. Beispiel: Zwei Komponenten zeigen unterschiedliche Medientypen (z.B. Videos und Bilder) in einer Listenansicht an. Beide Komponenten abonnieren eine Datenquelle, verändern ihren State wenn sich die Datenquelle ändert und deabonnieren die Datenquelle zum Ende ihres Lebenszyklus. Diese gemeinsame Funktionalität lässt sich in einen HoC extrahieren. HoC sind jedoch keine eigene Klasse aus der React-Bibliothek, sondern nur ein empfohlenes Pattern der Entwickler.

UI-State und Lifecycle

Nicht alle Komponenten liefern nur statische Inhalte abhängig von Props. Um komplexere Komponenten zu erstellen benötigen Komponenten eine State. Dieser State beinhaltet definierte Werte und existiert solange wie die Komponente selbst. Ein einfaches Beispiel ist ein Timer: Hier muss die abgelaufene Zeit gespeichert werden, sonst bleibt der Timer beim initialen Wert stehen. React liefert für Klassenkomponenten das Attribut `state` und die Funktion `setState` mit.

React besitzt ebenfalls Lifecycle-Hooks. `componentWillMount` und `componentWillUnmount`

entsprechen z.B. Angulars `ngOnInit` und `ngOnDestroy` und dienen zur Vorbereitung der View oder Aufräumen zur Vermeidung von Memory-Leaks (Listing 3.9).

Hooks

Hooks sind ein relativ neues Feature in React, sie wurden mit Version 16.8 Anfang 2019 eingeführt. Sie versuchen einige Probleme zu lösen, die besonders mit Klassenkomponenten verursacht werden:

- Wrapper Hell: Schachtelung von Komponenten ermöglicht die Wiederverwendung von Logik. Das führt zu Unübersichtlichkeit, Debugging wird erschwert
- Klassenkomponenten sind durch viel Boilerplate schwer zu lesen und überladen
- Klassenkomponenten sind nicht intuitiv. `this` verhält sich in JavaScript anders, Memberfunktionen müssen mit `bind()` im Konstruktor registriert werden[26]

Hooks lösen die Probleme mit Klassenkomponenten durch eine lesbare und funktionale API, mit Custom Hooks lässt sich mehrfach benötigte Logik extrahieren. Die zwei wichtigsten Hooks sind `useState` und `useEffect`, ein Beispiel ist in Listing 3.10 zu sehen.

3.3.2 Reconciliation

Reconciliation ist der Change Detection Algorithmus von React. Ausgelöst wird der Algorithmus durch die Veränderung des Status über Hooks oder `setState`. Jeder Aufruf der `render()` Funktion liefert einen neuen Baum mit React Elementen, der nun möglichst effektiv verarbeitet werden muss. React vergleicht zunächst die `root`-Knoten, sind sie unterschiedlich, dann wird der komplette Baum neu aufgebaut. Sind die Elemente gleich,

Listing 3.9: Lifecycle-Methoden in React

```
class Timer extends React.Component {
    constructor(props) {
        super(props);
        this.state = { time: 0 };
    }

    componentDidMount() {
        this.timerInterval = setInterval(
            () => this.setState({
                    time: this.state.time + 1()
             });
            1000
        );
    }

    componentWillUnmount() {
        clearInterval(this.timerInterval);
    }

    render() {
        return (
            <div>
            <h1>Timer</h1>
            <h2>Es sind {this.state.time}s vergangen.</h2>
            </div>
        );
    }
}
```

Listing 3.10: React-Hooks useState und useEffect

```
function Greeting() {
    // Status und Setter definieren, Hook setzt initialen Status
    const [userName, setUserName] = useState('Max');
    const [timeOnline, setTimeOnline] = useState(0);

    // fasst Lifecycle-Methoden zusammen, die zurückgegebene Funktion wird beim Clean-up a
    useEffect(() => {
        function handleStatusChange(status) {
            setUserName(status.userName)
            setTimeOnline(status.onlineTime);
        }

        UserAPI.subscribeToUserStatus(handleStatusChange);

        return function cleanup() {
            UserAPI.unsubscribeFromUserStatus(handleStatusChange);
        }
    );

    return (
        <span>
            {name} ist seit {timeOnline} Minuten online!
        </span>
    );
}
```

dann werden als Nächstes die Attribute verglichen. React iteriert im Anschluss über die Kindelemente. Einfügungen sind problematisch, da für den Algorithmus aufgrund der einfachen Iteration nicht feststellbar ist, ob ein Unterbaum intakt bleiben kann. Umgehen lässt sich das Problem mit Keys. Über das gleichnamige Attribut werden die Kindelemente verglichen, einfache Verschiebungen im Unterbaum werden so erkannt. Die ID wird in der Regel von der Objekt-ID abgeleitet:

```
<li key={user.id}>{user.name}</li>
```

Reconciliation ist eng verzahnt mit dem Lebenszyklus von Komponenten. Der Unterschied zu Angular liegt besonders darin, das Komponenten inklusive State komplett gelöscht und neu gerendert werden, wenn sich der Status ändert, während Angular lediglich die Bindings mit dem neuen Wert anpasst.[27]

3.3.3 State-Management

State Management in React ist eine komplexere Aufgabe. Der bereits betrachtete UI-State bezieht sich nur auf eine Komponente und ist beim nächsten Rendern verloren. Darüber hinaus können Komponenten nicht miteinander kommunizieren. Beispiel: Eine Komponente zeigt den aktuell angemeldeten Nutzer und einen Statustext an. In einer anderen Komponente gibt es ein Formular, mit dem Name und Statustext verändert werden können. Beide Komponenten benötigen den initialen State, der Formulareintrag in Komponente 2 ändert zwar dort die Werte, Komponente 1 verbleibt aber im ursprünglichen Zustand. Eine Lösungsmöglichkeit besteht darin, neben dem User-Objekt eine Callback-Funktion aus der Elternkomponente durch den Baum zu geben. Komponente 2 übergibt die Änderungen an diese Funktion, welche dann das User-Objekt modifiziert. Die nachfolgende Change Detection aktualisiert dann den neuen State in beiden Komponenten. Diese

Methode ist zwar einfach, für umfangreiche Anwendungen mit komplexeren States und Abhängigkeiten führt dieses Vorgehen schnell zu chaotischem Programmcode.

Die von den React-Entwicklern vorgeschlagen Lösung ist das Flux-Entwurfsmuster. React besitzt mit der Context API[9] eine integrierte Lösung, um Daten mehreren Komponenten zur Verfügung zu stellen, ohne Props durch den ganzen Baum zu reichen. Damit lässt sich Flux implementieren. Redux ist wie bei Angular eine andere Möglichkeit. Eine weitere Alternative ist eine eigene Implementation von Dependency Injection und Services mit Observables. Egal worauf die Entscheidung fällt, für komplexere Anwendungen ist State-Management notwendig, React selbst bietet hier keine Lösungen an und überlässt dem Software-Architekten die Wahl.

3.3.4 Projektarchitektur

React bricht im Gegensatz zu Angular mit klassischen MV*-Entwurfsmustern, die *Separation Of Concerns* (dt. Trennung der Verantwortlichkeiten) als Zielstellung verfolgen. JSX vereint Logik und View, statt sie zu trennen. Separation of Concerns ist mit der richtigen Komposition von Komponenten trotzdem umsetzbar, lediglich die Herangehensweise verändert sich (Listing 3.11).

Mit React lassen sich wiederverwendbare Bausteine entwickeln, die dann als Komposition verschachtelt werden können. React verfolgt damit das Ziel, die Menge an Programmcode weiter zu reduzieren[28]. Am Ende bleibt React eine Frontend-Bibliothek zur Generierung von HTML. Die Architektur der Anwendung bleibt dem Software-Architekten und die Einhaltung der definierten Regeln den ausführenden Entwicklern überlassen. React forciert

[9] siehe: https://reactjs.org/docs/context.html

Listing 3.11: Trennung von View und Logik durch Komposition

```
class FetchEvents extends React.Component {
    /* Konstruktor und State ... */
    componentDidMount() {
        /* Aufrufe zur Backend-API ... */
    }
    /* View-Logik ... */
    render() {
        return this.props.children({ this.state.isLoading, this.state.events });
  }
}

class App extends React.Component {
    render (
        <FetchEvents>
        {({ isLoading, events }) => {
            if (isLoading)
                return <p>Termine laden ...</p>;
            if (events)
                return events.map(({ title }) =>
                    <p>Terminname: {title}</p>);
            else return null;
        }}
        </FetchEvents>
  );
}
```

anders als Angular keine Entwurfsmuster, das Entwicklerteam von Facebook gibt nur Empfehlungen ab.

4 Implementation

Das folgende Kapitel dokumentiert die wichtigsten Bestandteile der Implementation einer Testanwendung mit den betrachteten Frameworks. Dabei werden grundsätzliche Konzepte durch einfache Anwendungsfälle abgedeckt. Das Ziel sind nach Möglichkeit identische Webapplikationen, um die Vorgehensweise und Probleme vergleichen zu können. Sie hat nicht den Anspruch, alle erdenklichen Use-Cases abzudecken. Auf Drittbibliotheken- und Frameworks wird weitestgehend verzichtet. Der vollständige Quelltext ist auf GitHub zu finden[1], im diesem Kapitel werden nur Ausschnitte abgebildet.

4.1 Anforderungsbeschreibung

Die Anwendung soll Rezepte und Zutaten verwalten und erfüllt genauer die folgenden Anforderungen:

1. Rezepte können mit Namen und Anweisungen, benötigten Zutaten und Mengen eingetragen werden
2. Es können neue Zutaten hinzugefügt werden
3. Eine Übersicht zeigt die gespeicherten Zutaten und Mengen an
4. Eine Übersicht zeigt Rezepte und gibt an, ob die geforderten Zutaten in ausreichender Menge vorhanden sind

[1] siehe: https://github.com/fschmeis/bthesis_react_angular

Die Applikation verfügt über 3 mit Tabs navigierbare Ansichten für die Anforderungen 1, 3 und 4. Die Eingabe der Zutaten erfolgt über einen Dialog. Über den Tabs liegt eine einfache Leiste mit dem Titel der Anwendung.

Rezepte hinzufügen

Der Nutzer kann im linken Bereich der Ansicht über ein Formular den Titel und die Anweisen des Rezeptes eingeben. Auf der rechten Seite ist ein Button, der beim Klick den Zutaten-Dialog öffnet. Die bereits hinzugefügten Zutaten werden darunter in einer Liste angezeigt.

Unter diesem Formular sind zwei Buttons angeordnet: Ein Button zum Speichern des Rezeptes, der andere zum Zurücksetzen des Formulars.

Zutaten

Der Nutzer sieht eine Tabelle der angelegten Zutaten mit Einheit und verfügbarer Menge. Über einen Button darüber öffnet sich der Zutaten-Dialog, dort können neue Zutaten hinzugefügt werden. Die Tabelle lässt sich sortieren und filtern.

Kochbuch

Hier werden die angelegten Rezepte seitenweise angezeigt. Links stehen Titel und Anweisungen, rechts eine Liste mit benötigten und vorhandenen Zutaten. Ist alles nötige vorhanden, dann werden mit Klick auf den Button "Kochen"die Mengen der verbrauchten Zutaten entsprechend reduziert.

Unter dem Rezept kann über Pfeile geblättert werden. Darüber ist einer Filterleiste, um nach bestimmten Rezepten zu suchen.

Zutaten-Dialog

Ein einfacher Dialog, der Zutatenname, Einheit und Menge abfragt. Mit dem Button „Hinzufügen“werden die Eingaben bestätigt, ein weiterer Button ermöglicht den Abbruch der Aktion.

4.1.1 Umsetzung

Die Umsetzung soll möglichst ohne Drittlösungen auskommen, um die möglichen Grenzen der Standard-Versionen aufzuzeigen. Aus den Material Design Frameworks Angular Material[2] und Material UI (React) [3] werden sowohl einfache Komponenten als auch Schriftarten und Themes übernommen. Letzteres besitzt deutlich mehr Komponenten, daher wird mit Angular begonnen. Für das Styling wird Sass verwendet.

4.2 Angular

4.2.1 Dateistruktur

Das Projekt wird in verschiedene Teilbereiche strukturiert. Die Hauptkomponente der Anwendung (*app.component.html/ts/scss*) befindet sich automatisch direkt im *app*-Ordner und beinhaltet den Einstiegspunkt der Anwendung. Unter *app/components/* finden sich die 3 Hauptkomponenten der Anwendung. In einem weiteren Ordner (*app/shared/components*) befindet sich die Dialog-Komponente. Der Dialog wird an zwei Stellen benötigt und es ist

[2]siehe: https://material.angular.io
[3]siehe: https://material-ui.com

zur Übersicht sinnvoll, diesen Umstand schon in der Dateistruktur ersichtlich zu machen. Im *shared*-Ordner befinden sich außerdem die Models und Services.

Die Implementation in Angular mit den entsprechenden Vorüberlegungen unkompliziert. Das Angular CLI erzeugt automatisch neue Komponenten oder Services und fügt sie dem App-Modul hinzu.

4.2.2 Models

Die Models bilden das grundlegende Datenmodell der Anwendung ab (Listing 4.1). Unter *Unit* werden die verfügbaren Einheiten (string) deklariert, die der Nutzer für Zutaten wählen kann. Eine Zutat *Ingredient* definiert sich aus Name, Anzahl und Einheit, ein Rezept (*Recipe*) aus Name, Anweisungen und Zutaten.

4.2.3 Services

Die Anwendung besitzt zwei Services als Schnittstelle zum Backend, die mit dem HTTP-Client auf die Backend-API zugreifen. Die Services werden mit der beschriebenen Dependency Injection über den Konstruktor übergeben und können dann in den Komponenten genutzt werden. Im Listing 4.2 in der `OnInit`-Methode, die bei der Initialisierung der Komponente aufgerufen wird.

4.2.4 Komponenten

Die Anwendung besteht aus 4 Komponenten. Die *app*-Komponente ist die Hauptkomponente und Startpunkt der Anwendung. Sie enthält die Toolbar mit dem Anwendungstitel und eine Tab-Leiste, über welche die anderen Teilkomponenten erreicht werden können (Abbildung 4.1).

Listing 4.1: Datenmodell

```
// unit.enum.ts
export enum Unit {
  tl = 'TL',
  el = 'EL',
  stk = 'Stück',
  bnd = 'Bund'
  /* ... */
}
// ingredient.model.ts
export interface Ingredient {
  name: string;
  amount: number;
  unit: Unit;
}
// recipe.model.ts
export interface Recipe {
  name: string;
  instructions: string;
  ingredients?: Ingredient[];
}
```

Listing 4.2: Verwendung der Services

```
// cookbook.component.ts
export class CookbookComponent implements OnInit {
  /* ... */
  constructor(
    private recipeService: RecipeService,
    private ingredientService: IngredientService,
    /* ... */
  ) {}

  public ngOnInit(): void {
    this.fetchData();
  }

  private fetchData(): void {
    this.recipeService.getRecipes().subscribe((recipes) => {
      this.recipes = recipes;
      this.currentRecipe = this.recipes[0];
    });

    this.ingredientService.getIngredients().subscribe((ingredients) => {
      this.storedIngredients = ingredients;
    });
  }
  /* ... */
}
```

Abbildung 4.1: Kochbuch

Kochbuch

Die Komponente enthält neben der Service-Abfrage aus Listing 4.2 die ViewModel-Logik zum Vor- und Zurückblättern der Rezepte, zur Berechnung der fehlenden Zutatenmenge und Entnahme der Zutaten, wenn das Rezept gekocht wird. Die Template (Auszug im Listing 4.3) definiert lediglich die Darstellung des ViewModels. Die mit `mat` beginnenden Tags referenzieren Komponenten der Material-Bibliothek.

In Zeile 6 wird mit `ngFor` über alle Zutaten des aktuellen Rezeptes iteriert, für jede dieser Zutaten wird ein neues `mat-list-item` erzeugt, welches entweder Text (fehlende Menge, Einheit und Zutat) oder ein Häkchen-Icon enthält. Bestimmt wird das durch die Abfrage

Listing 4.3: Template der Zutaten-Checkliste

```
<div *ngIf="insufficientIngredients(); else cook">
  <span>Dir fehlen:</span>

  <mat-list role="list">
    <mat-list-item role="listitem"
      *ngFor="let ing of currentRecipe.ingredients; last as isLast">

      <ng-container *ngIf="getMissingAmount(ing) > 0; else tick">
        {{getMissingAmount(ing)}} {{ing.unit}} {{ing.name}}
      </ng-container>

      <ng-template #tick>
        <mat-icon id="tick">done</mat-icon>
      </ng-template>

      <mat-divider *ngIf="!isLast"></mat-divider>

    </mat-list-item>
  </mat-list>
</div>

<ng-template #cook>
  <div id="cook">
    <button mat-raised-button color="primary"
      (click)="cookRecipe()">Kochen
    </button>
  </div>
</ng-template>
```

Listing 4.4: Setzen und Filtern der Datenquelle

```
1  export class IngredientsComponent {
2    dataSource!: MatTableDataSource<Ingredient>;
3    /* ... */
4    private fetchIngredients(): void {
5      this.ingredientService.getIngredients().subscribe((ingredients) => {
6        this.dataSource = new MatTableDataSource(ingredients);
7      });
8    }
9    /* ... */
10   public applyFilter(event: Event): void {
11     const filterValue = (event.target as HTMLInputElement).value;
12     this.dataSource.filter = filterValue.trim().toLowerCase();
13
14     if (this.dataSource.paginator) {
15       this.dataSource.paginator.firstPage();
16     }
17   }
18 }
```

in Zeile 8, sind genug Zutaten vorhanden, dann wird die Template-Variable `tick` wahr und das Icon gerendert. `ng-template` und `ng-container` sind strukturelle Direktiven, die selbst nicht gerendert werden.

Zutaten

Die Übersicht der vorhandenen Zutaten besteht aus einer Tabellenkomponente der Material-Bibliothek und einem Button, der den Zutaten-Dialog öffnet. Die Material-Tabelle verfügt über die Möglichkeit, Spalten zu sortieren und durch Einträge zu schalten. Eingaben in der Suchleiste filtern die Datenquelle (Listing 4.4).

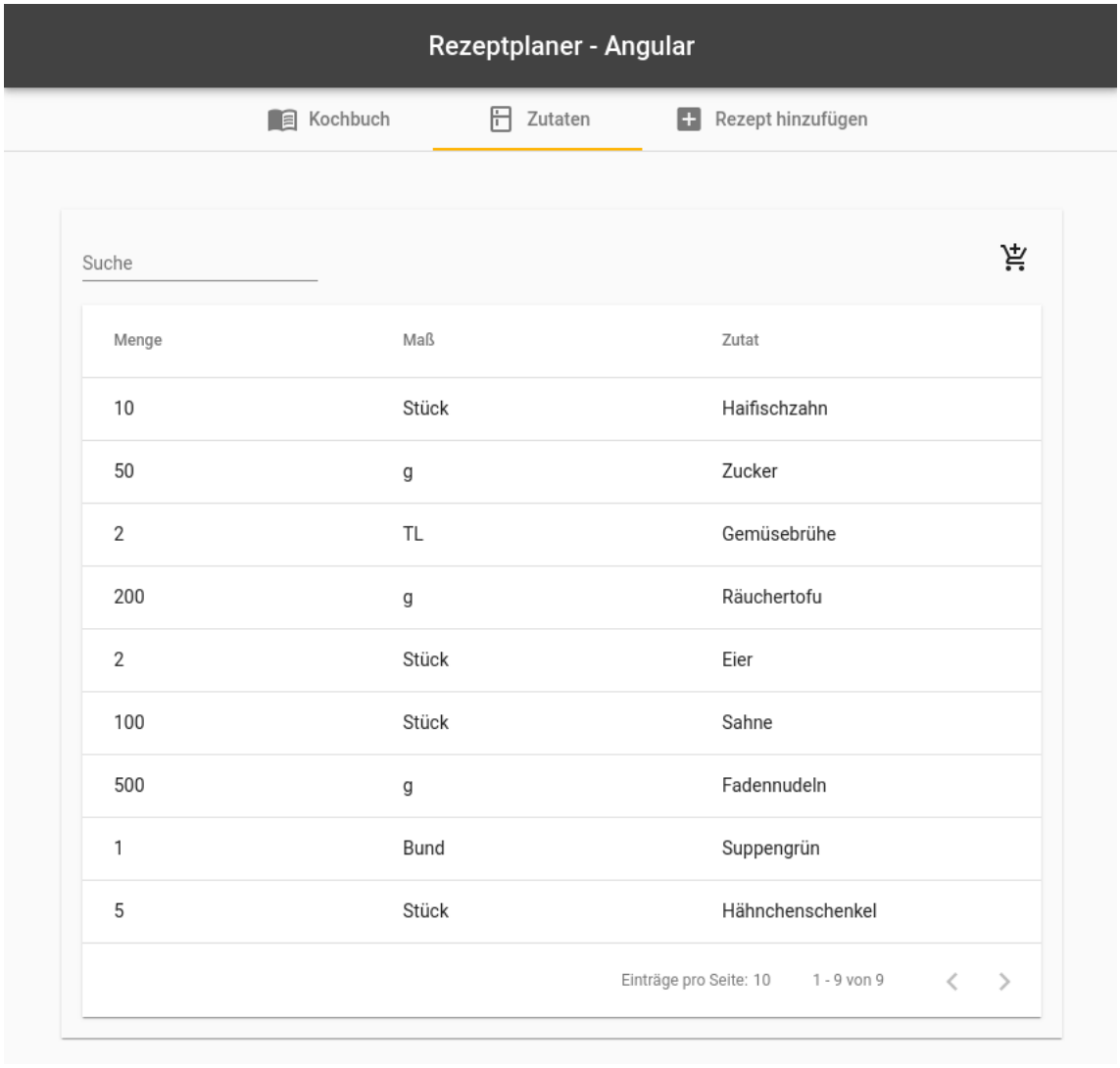


Abbildung 4.2: Zutaten-Übersicht

Listing 4.5: Dialogaufruf und Rückgabe der Daten

```
// add-recipe.component.ts
public onClickAddIngredient(): void {
  this.dialog
    .open(AddIngredientDialogComponent)
    .afterClosed()
    .subscribe((newIngredient) => {
      this.updateIngredients(newIngredient as Ingredient);
    });
}

// add-ingredient-dialog.component.ts
// Nutzereingabe wird zurückgegeben
add(): void {
  const newIngredient = {
    ...this.ingredientFormGroup.value,
    id: 0
  } as Ingredient;
  this.dialogRef.close(newIngredient);
}
```

Rezepte hinzufügen

Die Komponente definiert Methoden, die das Formular zurückzusetzen, Zutaten hinzuzufügen oder entfernen und das Rezept bei entsprechender Nutzereingabe zu speichern.

Zutaten-Dialog

Der Dialog wird aus anderen Komponenten heraus aufgerufen. Mit Angular Formularen lassen sich die eingegeben Werte validieren. Das Ergebnis des Dialogs wird im Anschluss wieder an die aufrufende Komponente zurückgegeben (Listing 4.5).

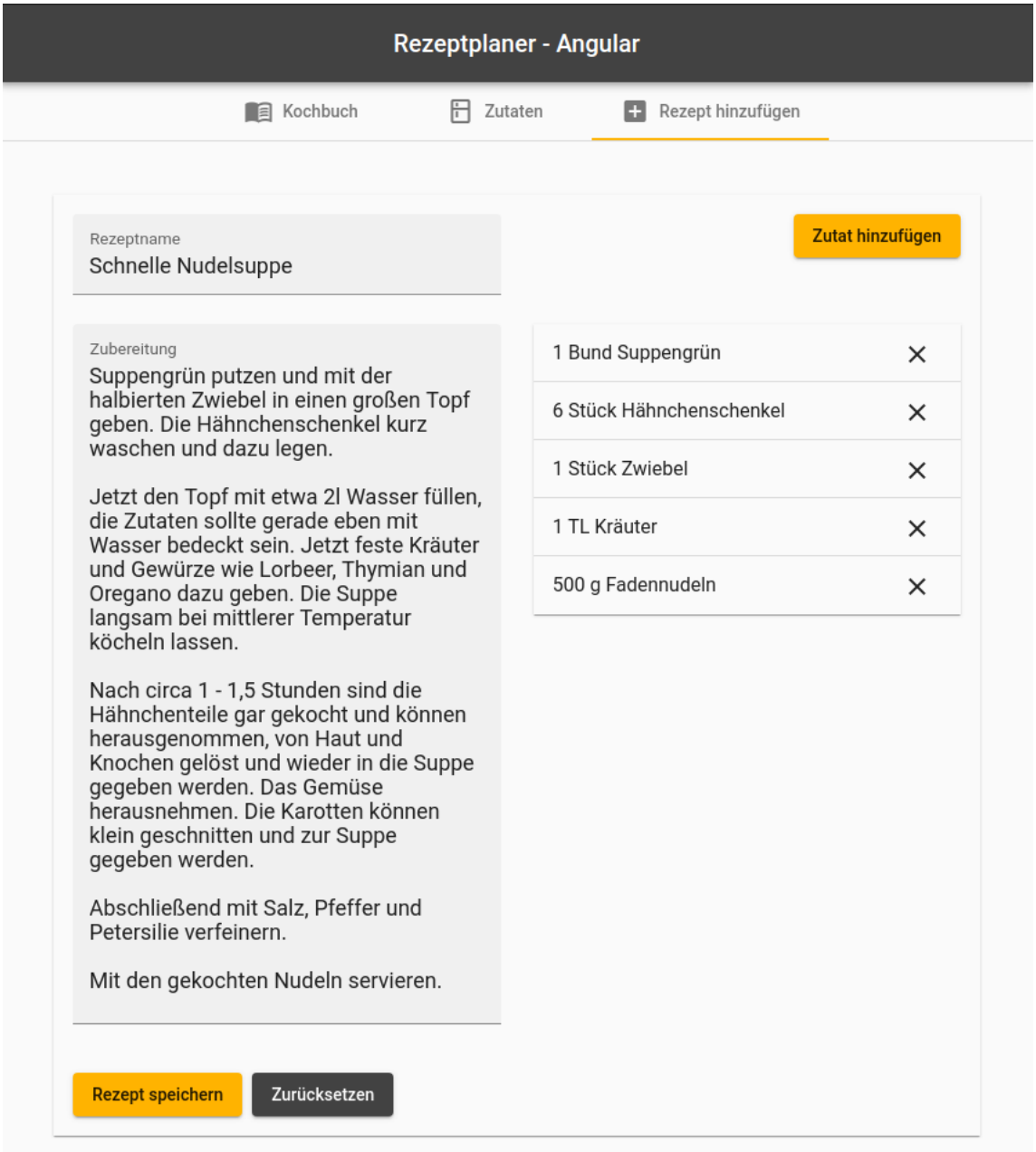


Abbildung 4.3: Rezepte hinzufügen

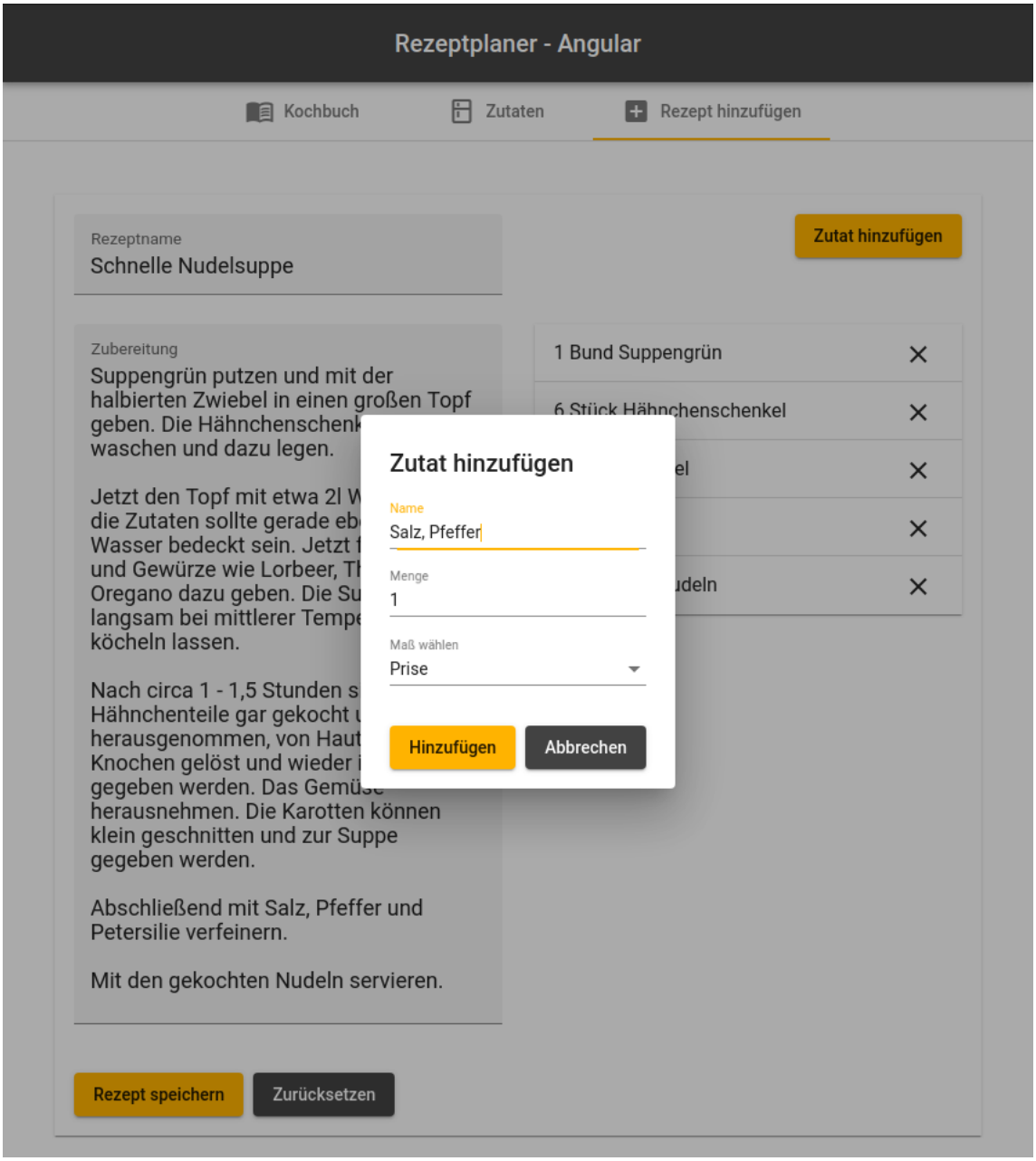


Abbildung 4.4: Zutaten-Dialog

4.3 React

Für die Implementation mit React müssen durch die Einfachheit der Bibliothek einige Vorüberlegungen getroffen werden. Aufgrund der geringen Komplexität der Testanwendung wird auf weitere Lösungen wie Redux verzichtet. Die Services aus der Angular-Implementation werden übernommen. Die Root-Komponente reicht den für die Views relevanten Teil des States (z.B. Zutaten für die Übersicht) zusammen mit einer Callback-Funktion an die Komponenten weiter, über welche eine Aktualisierung der Daten erreicht wird. Die Komponenten werden als Funktionen mit Hooks geschrieben.

Für die React-Implementation wird ebenfalls TypeScript verwendet. Die Vorteile einer statischen Typisierung wurden bereits erläutert, die Entscheidung fällt aus Zeitgründen auf TypeScript. Die Dateien haben statt `.jsx` die Endung `.tsx`.

Auf Abbildungen wird verzichtet, da die Anwendung bis auf die Zutatenübersicht identisch aussieht. Die Tabellenkomponente aus der React Material UI erfordert im Gegensatz zu Angular größeren Aufwand, aus diesem Grund wird hier zusätzlich `material-table`[4] als fertige Komponente verwendet. Die Funktionalität ist gleich, das Aussehen etwas verändert.

4.3.1 Dateistruktur

Die Einstiegskomponente `App.tsx` befindet sich der Anwendung Unter *app/components/* finden sich die 3 Hauptkomponenten der Anwendung. In einem weiteren Ordner (*app/shared/components*) befindet sich die Dialog-Komponente. Der Dialog wird an zwei Stellen benötigt und es ist zur Übersicht sinnvoll, diesen Umstand schon in der Projektstruktur ersichtlich zu machen. Im *shared*-Ordner befinden sich außerdem die Models und Services.

[4] siehe: https://material-table.com

Die Implementation in Angular mit den entsprechenden Vorüberlegungen unkompliziert. Das Angular CLI erzeugt automatisch neue Komponenten oder Services und fügt sie dem App-Modul hinzu.

4.3.2 Übernommene Teile

Die Models werden übernommen. Services ebenfalls, allerdings wird eine einzelne Service-Instanz exportiert: `export const ingredientService = new IngredientService();`. Diese Instanz kann dann importiert werden. Die Styles werden ebenfalls übernommen, allerdings erfordern die Unterschiede der Material-UI Bibliotheken entsprechende Anpassungen der Selektoren.

4.3.3 Komponenten

MainTabs

Die `MainTabs`-Komponente ist die Elternkomponente für alle von Daten abhängigen Komponenten, sie reicht diese zusammen mit Callbacks an die Kindkomponenten. Die Callbacks sind einfache Funktionen, welche Rezepte respektive Zutaten entgegen nehmen und an die entsprechenden Services zur Speicherung übergeben (Listing 4.6).

Kochbuch

Diese Komponente beinhaltet die Navigation zwischen den Rezepten und eine weitere Komponente, welche die Rezepte abbildet. Diese `RecipePage`-Komponente bringt den Rezeptnamen und die Anweisungen zur Darstellung und enthält ebenfalls eine weitere Komponente. Die `IngredientChecklist` rendert die Zutatenlisten (siehe 4.1).

Listing 4.6: MainTabs und Kindkomponenten

```
<!-- MainTabs.tsx -->
<TabPanel value="1">
    <Cookbook isLoading={areRecipesLoading}
              recipes={recipes}
              storedIngredients={storedIngredients} />
</TabPanel>
<TabPanel value="2">
    <Ingredients storedIngredients={storedIngredients}
                 handleIngredient={handleIngredient} />
</TabPanel>
<TabPanel value="3">
    <AddRecipe handleRecipe={handleRecipe} />
</TabPanel>
<!-- ... -->
```

Zutaten

Hier kommt das `material-table`-Paket zum Einsatz. Die externe Komponente ist ein gutes Beispiel für die generelle Problematik, die durch Drittlösungen entstehen kann. In der Konsole erscheint ein Error, wenn die Tabelle gerendert wird, auch wenn er keine sichtbarem Folgen hat: „React does not recognize the ‘scrollWidth‘ prop on a DOM element“. Das Paket nutzt ein Prop, das in der aktuellen Material-Version nicht mehr existiert.

Über Props kann die Tabelle konfiguriert werden, um unter anderem die Daten zur Verfügung zu stellen, die Spalten zu definieren oder deutsche Übersetzungen der Label anzugeben. Der Zutaten-Dialog wird ebenfalls im JSX deklariert, aber erst gerendert, wenn das `open`-Attribut einen wahren Wert übergeben bekommt (Listing 4.7).

Listing 4.7: Zutaten-Übersicht mit material-table

```
// Daten und Callback aus der Elternkomponente MainTabs
interface IngredientsProps {
    storedIngredients: Ingredient[];
    handleIngredient: (Ingredient: Ingredient) => void;
}

export default function Ingredients(props: IngredientsProps) {
    const [dialogOpen, setDialogOpen] = useState(false);

    return (
        <Paper className="tab-paper">
            <MaterialTable
                data={props.storedIngredients}
                actions={[
                    {
                        icon: "add_shopping_cart",
                        tooltip: "Zutat hinzufügen",
                        isFreeAction: true,
                        onClick: () => setDialogOpen(true),
                    },
                ]}
            />
            {/* weitere Optionen ... */}

            <IngredientDialog
                open={dialogOpen}
                setOpen={setDialogOpen}
                onConfirm={props.handleIngredient}></IngredientDialog>
        </Paper>
    );
}
```

Rezepte hinzufügen

Die Komponente ist mit 142 Zeilen deutlich größer als die anderen, aber nicht zu lang. Ähnlich zum Kochbuch hätten einzelne Teile in weitere Komponenten ausgelagert werden können. Man muss hier entsprechend abwägen, das Kochbuch ist zwar übersichtlicher, allerdings muss zwischen mehreren Dateien gesprungen werden.

Zutaten-Dialog

Die Einbindung des Zutaten-Dialogs wurde bereits in Listing 4.7 gezeigt. Die Komponente selbst setzt ebenfalls stark auf die Material-Komponenten (Listing 4.8).

Listing 4.8: Zutaten-Dialog

```tsx
// IngredientDialog.tsx
/* ... */
const handleChange = (name: string) =>
    (event: ChangeEvent<{ value: unknown }>) => {
        setValues({ ...values, [name]: event.target.value });
};

const nameError = values.name === "";

return (
  {/* ... */}
  {/* Eingabefeld für den Zutatennamen */}
  <TextField
      required
      value={values.name}
      label="Name"
      onChange={handleChange("name")}
      error={nameError}
      helperText={nameError ? "Bitte gib eine Zutat an." : ""}
  />
  {/* ... */}
  {/* Bestätigung der Eingabe */}
  <Button
      onClick={() => {
          setOpen(false);
          onConfirm({
              name: values.name,
              amount: values.amount,
              unit: values.unit as Unit,
          });
      }}
      color="primary"
      variant="contained">
      Hinzufügen
  </Button>
```

5 Auswertung

5.1 Vergleich

Komponenten

Komponenten in Angular bestehen aus Klassen mit einem dazugehörigen Template. Lifecycle-Methoden kann der Entwickler für bestimmte Aufgaben (z.B. Backend-Anfragen) verwenden. Die Template Syntax erweitert HTML, um Komponenten und Direktiven verwenden zu können, auf Eigenschaften der zugrundeliegenden Komponente zuzugreifen und bei Änderung in der View aktualisieren (bidirektionale Bindung). Mitgelieferte Direktiven wie `ngFor` oder `ngIf` ermöglichen die dynamische Generierung von HTML, Pipes verarbeiten Daten auf Ebene der Templates. Mit In- und Outputs können Komponenten Daten in der Hierarchie nach unten oder oben weitergeben.

Komponenten in React bestehen aus Klassen, die eine render-Funktion bereitstellen, oder Funktionen, die JSX zurückliefern. Auf den Lifecycle kann für Klassen ebenfalls mit Methoden, für Funktionen mit React Hooks zugegriffen werden. Die Templates werden als JSX direkt in JavaScript definiert, Direktiven und Pipes gibt es nicht. Eigenschaften/Variablen der Klasse/Funktion können direkt verwendet werden. Mit Props werden Daten oder Callback-Funktionen an Kindelemente weiter gereicht, Komponenten werden bei Änderungen neu gerendert (unidirektionale Bindung).

Angular setzt auf Services, um von Komponenten unabhängige Logik abzukapseln. Sie werden per Dependency Injection bereitgestellt. React hat dafür keine Entsprechungen

und überlässt das dem Entwickler.

Change Detection und Performance

Für die Change Detection verwenden beide Frameworks unterschiedliche Algorithmen, die in der Vergangenheit viele Optimierungen erhalten haben. Bezogen auf die Geschwindigkeit kann hier keine gesicherte Aussage getroffen werden, da dazu keine Benchmarks gefunden werden konnten. Häufig wird nur die initiale Ladezeit in Betracht gezogen. Der Unterschied ist dabei nicht signifikant und davon abhängig, welche zusätzlichen Frameworks mit React verwendet werden[29]. In der Kochbuch-Testapplikation waren die Übergänge in der Angular-Variante spürbar flüssiger. Daraus sollte allerdings kein genereller Schluss gezogen werden, aufgrund der geringeren Erfahrung mit React könnten eventuelle Fehler die Performance beeinträchtigen.

Projektarchitektur

Angular orientiert sich an klassischen MV*-Mustern und erweitert sie mit Services. Projekte werden mit Modulen strukturiert. Komplexes State-Management kann mit Services gelöst werden, alternativ werden Frameworks wie Redux verwendet. Für viele Aufgaben gibt es eine Lösung direkt aus dem Framework: Internationalisierung, komplexe Formulare, eine integrierter HTTP Client oder Animation, nur in seltenen Fällen sind weitere Frameworks notwendig.

React hat keine Module oder Services, JSX ist per Definition gegensätzlich zu MV*Mustern. Für komplexes State-Management muss eine eigene Lösung implementiert oder ebenfalls auf externe Bibliotheken/Frameworks wie Redux zugegriffen werden. Diese Abhängigkeit

von externen Lösungen gilt für die meisten Aufgaben, beispielsweise Internationalisierung (react-intl), HTTP-Clients (axios) oder Animationen (react-animations).

5.1.1 Lernkurve

Die benötigte Einarbeitungszeit ist subjektiv und abhängig von den Fähigkeiten und der Erfahrung des Entwicklers. Trotzdem lässt sich die These aufstellen, dass React an sich eine flachere Lernkurve besitzt. Der Grund liegt im deutlich kleineren Funktionsumfang von React. Es gibt keine Dependency Injection, Services, Module oder Direktiven, TypeScript ist keine Pflicht. Selbst für kleinere Angular-Projekte sollte grundsätzliches Verständnis vorliegen. Wird React mit zusätzlichen Bibliotheken als Einheit gegenüber Angular betrachtet, dann nähern sich die Lernkurven an.

5.1.2 Anwendungsbereiche

Angular ist für große Enterprise-Anwendungen geeignet, da es out-of-the-box beliebig skalierbar ist. Bemerkbar macht sich dieser Umstand in der Größe, unverpackt verbraucht Angular 566KB Speicher, React lediglich 97,5KB[1]. Bei kleineren Projekten stellt sich dann die Frage, ob ein Framework dieser Größe notwendig ist. Die notwendige Boilerplate für Dekoratoren und Module macht einen größeren Anteil aus.

React bringt selbst nur einen Bruchteil an Lösungen mit sich, das wirkt sich entsprechend auf die Größe aus. Der Vorteil ist, dass sich React auch für kleine Projekte anbietet. Mit rein funktionalen Komponenten ohne State lassen sich in sehr kurzer Zeit dynamische Websites erstellen, durch JSX muss nicht zwischen HTML und JavaScript hin und her gesprungen werden. Unter Zuhilfenahme zusätzlichen Frameworks sind React-Projekte trotzdem beliebig skalierbar. Der Nachteil hierbei ist allerdings, dass das Projekt dadurch

[1] siehe: https://gist.github.com/Restuta/cda69e50a853aa64912d

zusätzlichen Abhängigkeiten unterliegt, da die Pakete ebenfalls up-to-date gehalten werden müssen. Der Verwaltungsaufwand ist in diesem Bereich also höher. Bei Angular-Projekten muss in der Regel nur das Framework selbst aktualisiert werden.

Die Entscheidung für oder gegen Angular respektive React ist am Ende immer abhängig von der zu lösenden Aufgabe und den gegebenen Voraussetzungen. React ist grundsätzlich in jedem Szenario verwendbar, aber je komplexer die Aufgabe ist, desto durchdachter muss die gewählte Architektur im Zusammenspiel verschiedener Frameworks sein. Angular ist für kleinere Websites mit statischen oder nur begrenzt dynamischen Inhalten ungeeignet, weil ein Großteil der gebotenen Lösungen nicht benötigt wird. Für Unternehmen ist die Entscheidung auch vom Personal abhängig, sie müssen sich der Fähigkeiten und Erfahrungen der beteiligten Entwickler bewusst sein. Das kann im Zweifel schwerer gewichtet sein, als die Eignung der Technologie.

6 Fazit

Angular und React sind zwei Technologien, mit deren Hilfe dynamische Webanwendungen entwickelt werden können. Die einzelnen Bestandteile des Frontends werden in wiederverwendbare Komponenten aufgeteilt. Beide erweitern klassisches HTML mit einem Templating-Ansatz, um eine stärkere Bindung mit JavaScript zu erreichen. Dadurch ist direkte DOM-Manipulation mit jQuery oder JavaScript selbst nicht mehr notwendig.

React bietet als für das Rendern von Komponenten optimierte Bibliothek nicht viel mehr als die genannten Punkte. Für komplexere Anwendung ist man auf Drittlösungen angewiesen. Das erhöht den Wartungsaufwand. Mit React können aber in kurzer Zeit dynamische Websites oder Webanwendungen gebaut werden.

Angular macht dem Entwickler mehr Vorgaben, es finden sich für gängige Problemstellungen immer mitgelieferte Lösungen. Diese sind allerdings immer enthalten, unabhängig davon, ob sie auch benötigt werden. Angular erfordert deutlich mehr Boilerplate, allerdings wird im Gegensatz zu React eine gute Basis auch für komplexere Anwendungen geboten. Das Angular CLI unterstützt die Generierung neuer Komponenten.

Diese Arbeit hat die Kernelemente der Frameworks durch eine tiefere Vorbetrachtung und Implementation vorgestellt und angewandt. Daraus wurde abgeleitet, für welche Art von Anwendung Angular und React geeignet sind. Es wurden mögliche Schwierigkeiten dargestellt, die mit Verwendung der Frameworks einher gehen.

6.1 Probleme

Die Wahl, Material-UI als Design-Bibliothek zu verwenden, war in der Nachbetrachtung eine schlechte Entscheidung. Die gelieferten Komponenten haben sich doch stark unterschieden. Eine bessere Entscheidung wäre ein Framework mit Komponenten gewesen, die nur aus HTML und CSS bestehen, wie beispielsweise Bootstrap. Diese hätten dann ohne Änderungen verwendet werden können und die Implementationsdetails der Frameworks hätten noch besser hervorgehoben werden können.

Der Umstand, das dieses Thema sehr aktuelle Technologien behandelt, wirkt sich auch auf die Literatur aus. Abseits von Handbüchern, die mit einem Update veraltet sein können, gibt es kaum Fachliteratur. Die wichtigste Quelle war daher die Dokumentation der Frameworks. Ansonsten finden sich unzählige Artikel oder Videomaterial im Netz, die Qualität variiert dabei stark.

Ein weiteres Problem war meine Einstellung zu Beginn der Implementation, den optimalen Weg zu finden und nach Best Practice zu arbeiten. Es gibt keinen allerdings nicht den einzig richtigen Best Practice, man findet immer wieder neue Lösungsansätze und Ansichten, die sich häufig auch widersprechen. Wichtig ist, Best Practices zu hinterfragen, um sie zu verstehen und auszuwählen, ob sie das vorliegende Problem lösen.

6.2 Erfahrungen und Lernerfolge

Mit Angular arbeite ich bereits aktiv seit einem Jahr, allerdings hatte ich hier die Möglichkeit Konzepte, die mir vorher nur oberflächlich bekannt waren, genauer zu betrachten und die Funktionsweise zu verstehen. Das betrifft insbesondere die Dependency Injection und Change Detection. Die Entwicklung mit Angular fällt mir dadurch leichter, auch weil ich Fehler schneller zuordnen kann.

Mit React hatte ich vor Beginn der Arbeit noch keine praktische Erfahrung, jetzt habe ich einen sehr guten Überblick über die Möglichkeiten, aber vor allem über die Grenzen der Bibliothek. In kleinen Projekten werde ich React zukünftig definitiv anwenden. Bei größeren Applikationen fällt meine Entscheidung vorerst auf Angular, da ich mit der klareren Trennung zwischen Template und Klassen strukturierter und besser arbeiten kann.

Ich habe gelernt, wie wichtig Architekturentscheidungen in Bezug auf Webapplikationen sind und das ein Verständnis der internen Abläufe des Frameworks und der Designphilosophie der Framework-Entwickler notwendig ist. Gleichzeitig muss man die Projektanforderungen richtig analysieren und eine Architektur ableiten, die eine umständliche Problemlösung darstellt. Für React stellt sich beispielsweise die Frage, ob eine State-Management Bibliothek wie Redux wirklich notwendig ist.

Literatur

[1] Ryan from ryadel.com. *What about Angular? Angular JS history through the years (2009-2019)*. 3. Okt. 2019. URL: https://www.ryadel.com/en/angular-angularjs-history-through-years-2009-2019/ (besucht am 20. 10. 2020).

[2] ng-conf. *Miško Hevery and Brad Green - Keynote - NG-Conf 2014*. 1. Jan. 2014. URL: https://www.youtube.com/watch?v=r1A1VR0ibIQ.

[3] *Angular*. URL: https://de.wikipedia.org/wiki/Angular (besucht am 20. 10. 2020).

[4] Ferenc Hámori. *The History of React.js on a Timeline*. 4. Apr. 2018. URL: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ (besucht am 20. 09. 2020).

[5] Markus Wingler. *Was ist eigentlich TypeScript?* 14. Juni 2018. URL: https://blog.doubleslash.de/was-ist-eigentlich-typescript/ (besucht am 20. 09. 2020).

[6] Facebook. *Static Type Checking*. URL: https://reactjs.org/docs/static-type-checking.html (besucht am 20. 09. 2020).

[7] Tab Atkins-Bittner. *A Word About CSS4*. 19. Feb. 2013. URL: https://www.xanthir.com/b4Ko0 (besucht am 20. 09. 2020).

[8] Facebook. *Styling and CSS*. URL: https://reactjs.org/docs/faq-styling.html (besucht am 21. 09. 2020).

[9] Google. *Setting up the local environment and workspace*. URL: https://angular.io/guide/setup-local (besucht am 21. 09. 2020).

[10] Facebook. *Create a New React App*. URL: https://reactjs.org/docs/create-a-new-react-app.html%5C#create-react-app (besucht am 21. 09. 2020).

[11] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 23. Jan. 2004. URL: https://martinfowler.com/articles/injection.html#ComponentsAndServices (besucht am 17. 09. 2020).

[12] Facebook. *flux-concepts*. 19. Jan. 2017. URL: https://github.com/facebook/flux/tree/master/examples/flux-concepts (besucht am 17. 09. 2020).

[13] Facebook. *Virtual DOM and Internals*. URL: https://reactjs.org/docs/faq-internals.html (besucht am 17. 09. 2020).

[14] Google. *Introduction to Angular concepts*. URL: https://angular.io/guide/architecture (besucht am 17. 09. 2020).

[15] Google. *Component Decorator*. URL: angular.io/api/core/Component (besucht am 17. 09. 2020).

[16] Google. *Hooking into the component lifecycle*. URL: https://angular.io/guide/lifecycle-hooks (besucht am 17. 09. 2020).

[17] Google. *Add services*. URL: https://angular.io/tutorial/toh-pt4 (besucht am 17. 09. 2020).

[18] Google. *Transforming Data Using Pipes*. URL: https://angular.io/guide/pipes (besucht am 17. 09. 2020).

[19] Google. *AsyncPipe*. URL: https://angular.io/api/common/AsyncPipe (besucht am 17. 09. 2020).

[20] Google. *NgModule*. URL: https://angular.io/api/core/NgModule (besucht am 17. 09. 2020).

[21] Google. *Introduction to services and dependency injection*. URL: https://angular.io/guide/architecture-services (besucht am 17. 09. 2020).

[22] Michael Hoffmann. *The Last Guide For Angular Change Detection You'll Ever Need*. 18. Nov. 2019. URL: https://www.mokkapps.de/blog/the-last-guide-for-angular-change-detection-you-will-ever-need/ (besucht am 17. 09. 2020).

[23] Andreas Kühnel. *C# 8 mit Visual Studio 2019*. 2019. URL: http://openbook.rheinwerk-verlag.de/visual_csharp_2012/1997_28_005.html (besucht am 17. 09. 2020).

[24] Cyrille Tuzi. *Architecture in Angular projects*. 2. März 2018. URL: https://medium.com/@cyrilletuzi/architecture-in-angular-projects-242606567e40 (besucht am 17. 09. 2020).

[25] Facebook. *Components and Props*. URL: https://tinyurl.com/ybl9nnbd (besucht am 17. 09. 2020).

[26] Dan Abramov Sophie Alpert und Ryan Florence. *React Today and Tomorrow and 90% Cleaner React With Hooks*. 26. Okt. 2018. URL: https://youtu.be/dpw9EHDh2bM?t=706 (besucht am 17. 09. 2020).

[27] Facebook. *Reconciliation*. URL: https://reactjs.org/docs/reconciliation.html (besucht am 17. 09. 2020).

[28] Krasimir Tsonev. *React and separation of concerns*. 16. Okt. 2018. URL: https://krasimir.gitbooks.io/react-in-patterns/content/chapter-13 (besucht am 17. 09. 2020).

[29] Jacek Schae. *A RealWorld Comparison of Front-End Frameworks 2020*. URL: https://medium.com/dailyjs/a-realworld-comparison-of-front-end-frameworks-2020-4e50655fe4c1 (besucht am 24. 09. 2020).