BACHELORARBEIT

zur Erlangung des akademischen Grades
„Bachelor of Science in Engineering“
im Studiengang Informatik

An Overview of Stateless vs Stateful Web Programming using React and Redux

Ausgeführt von: Jan Calanog

Personenkennzeichen: 1510257042

Begutachter: DI Harald Steinlechner, BSc.

Wien, den 5. April 2018

FH University of Applied Sciences
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Wien, 5. April 2018 Unterschrift

Kurzfassung

Der Hauptfokus dieser Arbeit ist die Anwendung der JavaScript-Bibliothek React in Verbindung mit der Redux-Bibliothek. Dabei wird der Unterschied zwischen React-Komponenten untersucht, die ihren eigenen Zustand verwalten, und Komponenten die keinen Zustand haben, wobei das Zustandsmanagement auf Redux verlagert wird. Dafür werden zwei Beispiel React-Applikationen, mit den genannten Unterschieden miteinander verglichen und bezüglich Wartbarkeit und Erweiterbarkeit ausgewertet. Weiters werden die Grundlagen, die für diese Technologien benötigt werden, vorgestellt.

Schlagworte: React, Flux, Redux, Zustand, Zustandslos

Abstract

The main focus of this work is the usage of the JavaScript library React in conjunction with the Redux library. By doing so the differences between stateful React components and stateless React components are reviewed. Therefore two example React applications, with the mentioned differences are compared and evaluated in terms of maintainability and extensibility. Furthermore, the basics needed for these technologies are presented.

Keywords: React, Flux, Redux, Stateful, Stateless

Inhaltsverzeichnis

1 Einführung

Single-Page-Applikationen (SPAs) werden in der Webentwicklung immer wichtiger. Eine SPA ist eine Webapplikation, welche im Gegensatz zu klassischen Webapplikationen nur aus einer Seite besteht. Das bedeutet, dass die Seite nicht bei jeder Userinteraktion neu geladen werden muss. Somit wird die Nutzererfahrung einer schnellen und responsiven Desktopapplikation in den Webbrowser gebracht.

Mit der steigenden Beliebtheit und dem steigenden Einsatz von SPAs, steigt auch die Komplexität der Anforderungen, die SPAs erfüllen müssen. In der Dokumentation von Redux wird betont:

> *"Unser Code muss mehr Zustände verwalten, als jemals zuvor." [11, Motivation]*

Dieser Zustand beinhaltet, ist aber nicht beschränkt auf

- Persistente Daten,
- Rückmeldungen vom Server,
- die aktuell aktive Route,
- sowie Benutzeroberflächen bezogener Zustand usw.

Die derzeit meist verwendeten JavaScript (JS) Front-End Frameworks, neben React [11], sind AngularJS [1], Angular [9], Vue.js [14] und Backbone.js [2]. [13] Die genannten JS Frameworks basieren entweder auf einer komponentenbasierten Architektur, auf der MVC-Architektur (Model-View-Controller) oder einer Ableitung von MVC, wie MVVM (Model-View-ViewModel) oder MV* (Model-View-Whatever). [2, 14, 6, 9] In diesen Architekturen wird weder festgelegt wo Zustände gelagert werden noch ob der Datenfluss einfachgerichtet oder doppeltgerichtet sein soll. Das bedeutet, im Falle von einem doppeltgerichteten Datenfluss, dass die View automatisch aktualisiert wird, wenn sich der Zustand ändert und andersherum sich der Zustand automatisch aktualisiert, wenn die View verändert wird.
Wie das Team von Facebook erkannt hat, ist das bei überschaubaren Applikationen noch kein Problem. Bei größeren Applikationen jedoch, wo mehrere Komponenten und Features einen geteilten Zustand haben und die Veränderung des Zustands Aktualisierungen von mehreren Views auslöst und diese wiederum andere Teile des Zustands aktualisieren usw., führt das zu einer Kette von Zustandsveränderungen und macht den aktuellen Zustand der Applikation unberechenbar. [10]

1.1 Hintergrund und Motivation

Seit dem Sommer 2017 arbeite ich bei dem IT-Dienstleister Iteratec GmbH. In den ersten Monaten wurde ich einem Team zugeteilt, welches angefangen hatte die Webapplikation http://www.malsato.com zu entwickeln. Diese Applikation liest automatisch aktuelle Mittagsmenüs aus mehreren Quellen in der DonauCity-Umgebung in Wien und zeigt sie gesammelt auf der Webseite an. Mein Team und ich wollen die Webapplikation expandieren. Eine Strategie um dies zu erreichen ist die Entwicklung einer Schnittstelle für Restaurants in der ausgewählte Gastronomieunternehmen selbständig ihre Mittagsmenüs eintragen können. Diese Arbeit dient dazu zu evaluieren, in welcher Art die Single-Page-Applikation entwickelt werden soll.

1.2 Ziele

Die Ziele dieser Arbeit sind:

- Einen Überblick darüber zu verschaffen, wie die Flux-Architektur, im Gegensatz zur MVC Architektur, mit Zustandsveränderungen umgeht.
- Zu untersuchen wie Redux, eine konkrete Implementierung von Flux, in Verbindung mit React, durch funktionale Prinzipien, das Zustandsmanagement optimieren kann.
- Zu eruieren, ob eine React-Applikation mit der Redux-Bibliothek leichter zu warten und zu erweitern ist.
- Und zu ermitteln, wie einfach oder schwer es ist, eine Undo/Redo-Funktion mit Redux zu implementieren.

1.3 Struktur dieser Arbeit

In Kapitel 2 werden die Grundlagen von MVC, Flux, Redux und React durchgenommen. In Kapitel 3 werden zwei React-Applikationen vorgestellt. Dabei wurde eine Applikation ausschließlich mit Klassenkomponenten entwickelt, diese verwalten ihren eigenen Zustand. Die zweite Applikation hingegen, wurde ausnahmslos mit Funktionalen Komponenten entwickelt. Das Zustandsmanagement wird in der zweiten Applikation auf Redux ausgelagert. Danach werden beide Applikationen miteinander verglichen und die Unterschiede ausgewertet.

2 Grundlagen

In diesem Kapitel werden unterschiedliche grundlegende Begriffe und Prinzipien erklärt, sowie ein Einblick in die wichtigsten, für diese Arbeit relevanten Frameworks gegeben.

2.1 Model-View-Controller

Das Konzept von MVC, ursprünglich Thing-Model-View-Editor, wurde im Jahr 1978 von Trygve Reenskaug veröffentlicht.

> *"MVC wurde als eine offensichtliche Lösung für das allgemeine Problem geschaffen, Benutzern die Kontrolle über ihre Informationen zu geben, wie sie aus verschiedenen Perspektiven betrachtet werden können." [12, Seite 1]*

Das Schwierigste an der Erschaffung von MVC, so Reenskaug, soll lediglich die Namensgebung der einzelnen architektonischen Komponenten gewesen sein. [12] Nichtsdestotrotz, gehört MVC bis heute zu den wichtigsten und meistgenutzten Softwarearchitekturen in der Webentwicklung.

2.1.1 Ziel

Seit der Schöpfung wurde MVC immer wieder angepasst und modifiziert. So oft, dass es in den Jahren seine Spezifität verloren hat und nur noch eine Richtlinie geworden ist, an der sich der Entwickler halten kann. Wie MVC funktioniert und anzuwenden ist, hängt letztendlich davon ab, wie das eingesetzte Framework MVC implementiert.
Das Ziel ist jedoch immer noch das Gleiche geblieben: *Die Trennung von der Logik der Applikation, seines Datenmodells und seiner Präsentation [3].*

2.1.2 Komponenten

Eine MVC-Applikation in drei Teile unterteilt:

- Model
- View
- Controller

Wobei jedes dieser Teile seine eigenen Aufgaben hat. [12]

Das Model ist ein Objekt, das einige Informationen über die Domäne darstellt. Es ist ein nichtvisuelles Objekt, das alle Daten und das Verhalten enthält. [8]

Die View ist gleichzeitig die Darstellung des Models in der Benutzeroberfläche (eine visuelle Repräsentation der Informationen) und die Schnittstelle für Benutzerinteraktion, welches es ermöglicht das Model zu aktualisieren. [8, 12]

Der Controller verbindet das Model und die View indem es die Benutzereingaben im View interpretiert und an das Model weiterleitet. Umgekehrt aktualisiert er auch die View in geeigneter Weise. [8, 12]

2.1.3 Datenfluss

In einer MVC-Applikation ist nicht definiert wie der Datenfluss abläuft. Abbildung 1 zeigt, dass die Kommunikation zwischen den Komponenten in beide Richtungen gehen kann.

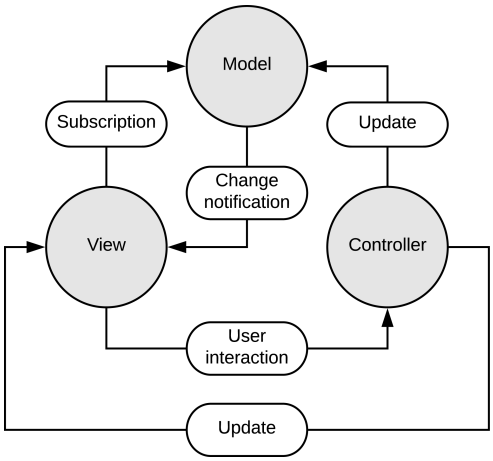

Abbildung 1: Datenfluss einer MVC Applikation

2.1.4 Zustandsmanagement

Wie und wo der Zustand verwaltet wird, ist in der MVC-Architektur nicht definiert und liegt in der Designentscheidung des Entwicklers.

2.2 React

React ist eine komponentenbasierte JavaScript-Bibliothek um User Interfaces (UI) für Single-Page-Applikationen zu erstellen. React benützt das Virtuelle DOM (Document Object Model), um die Logik für das direkte Manipulieren des Html DOMs für den Programmierer zu abstrahieren. So ist der Programmierer in der Lage, anstelle eines imperativen Programmierstils, einen deklarativen Programmstil zu verwenden. Dies führt zu weniger Code und einer besseren Wartbarkeit gegenüber imperativem Code. [7]

Quellcode 1 und 2 sind beide eine Beispielimplementierung für ein Input mit einem Initialzustand. Dieser Zustand wird durch das `onChange`-Event des Inputs aktualisiert.

```
1    <input id="dish-name" type="text" onChange="handleChange(this.value)"/>
2    <script>
3        var dishName = "Wiener Schnitzel";
4        const dishInput = document.getElementById("dish-name");
5        dishInput.value = dishName;
6        function handleChange(value) {
7            dishName = value;
```

```
        }
    </script>
```

Quellcode 1: Beispiel für imperativen Code

```
class Input extends React.Component {
    this.state = { dishName: "Wiener Schnitzel" }
    handleChange = (ev) => { this.setState({ dishName: ev.target.value }) }
    render = () => React.createElement("input", { value:
        this.state.dishName, type: "text" })
}
```

Quellcode 2: Beispiel für deklartiven Code in React

In Quellcode 1 wird das DOM direkt manipuliert. Der Unterschied in Quellcode 2 ist, dass in React nicht das DOM selbst manipuliert wird. Es deklariert lediglich ein Element, dass mit dem Zustand gerendert werden soll.

2.2.1 Das Virtuelle DOM

Das virtuelle DOM von React ist eine Abstraktion des Html DOM, wobei eine Komponente einem Node im Html DOM entspricht. *"Das Virtuelle DOM ist eine schnelle, In-Memory Repräsentation des echten DOM."* [7, Seite 19], so Fedosejev und erklärt den Prozess in drei Schritten:

- Wenn sich der Zustand des Datenmodells verändert, dann wird das UI von der Virtuellen DOM und React zu einer Virtuellen DOM Repräsentation neu gerendert .
- Danach berechnet React den Unterschied zwischen beiden Virtuellen DOM Repräsentationen: Die vorige Virtuelle DOM Repräsentation, die berechnet wurde, bevor sich die Daten verändert haben und die aktuelle Virtuelle DOM Repräsentation, die nach der Datenveränderung berechnet wurde. Dieser Unterschied zwischen beiden Virtuellen DOM Repräsentationen, ist was tatsächlich im echten DOM verändert werden muss.
- React aktualisiert nur das, was wirklich im echten DOM aktualisiert werden muss.

2.2.2 JSX

Das Erstellen von React-Elementen kann erleichtert werden, wenn die JavaScript-Spracherweiterung JSX verwendet wird. Dies ermöglicht React-Elemente in Html ähnlichen Syntax zu deklarieren. [6]

```
// Normale Syntax
const h1 = React.createElement("h1", { className: "title"}, "Hello, world!");
// JSX Syntax
const h1 = <h1 className={"title"}>Hello, world!</h1>;
```

Quellcode 3: JSX Syntax Beispiel

Quellcode 3 zeigt zwei äquivalente Deklarierungen eines React-Elements.

2.2.3 Erstellen einer React-Komponente

React bietet zwei verschiedene Arten von Komponenten: Funktionale- und Klassenkomponenten. Ersteres ist eine einfache JavaScript Funktion mit einem Argument namens `props`, welche ein React-Element zurückgibt [6]. Zweiteres kann mit einer ES6-Klasse wie in Quellcode 5 definiert werden.

```
1  function Welcome(props) {
2      return <h1>Hello, {props.name}</h1>;
3  }
```

Quellcode 4: Ein Beispiel für eine Funktionale Komponente

```
1   class Welcome extends React.Component {
2       render() {
3           return <h1>Hello, {this.props.name}</h1>;
4       }
5   }
```

Quellcode 5: Ein Beispiel für eine Klassenkomponente

Laut Dokumentation sind beide Komponenten aus Reacts Sicht äquivalent. Jedoch können Klassenkomponenten im Gegensatz zu Funktionalen Komponenten ihren eigenen lokalen Zustand verwalten, wie in Quellcode 2 zu sehen ist. Der Zustand einer Klassenkomponente sollte nur mit der Methode `setState()` mutiert werden, so wird der Zustand mit dem neuen State ersetzt und nicht direkt manipuliert. Darüber hinaus wird die Komponente über diese Methode informiert, ob sie neu gerendert werden muss. [6]

Lebenszyklus einer Komponente

Zusätzlich können in Klassenkomponenten die Lebenszyklus-Funktionen von React.Component überschrieben werden, welches in Funktionalen Komponenten nicht möglich ist. [6] Fedosejev unterteilt Reacts Lebenszyklus-Hooks in 3 Phasen [7]:

Mounting: Diese Phase tritt ein, wenn eine Komponente ins DOM eingefügt wird.

Updating: Diese Phase tritt ein, wenn eine Komponente ins Virtuelle DOM neu gerendert wird und berechnet muss, ob das echte DOM aktualisiert werden muss.

Unmounting: Diese Phase tritt ein, wenn eine Komponente aus dem DOM entfernt wird.

2.2.4 Hierarchie von Komponenten

In React können Komponenten, wie im Html DOM verschachtelt werden, siehe Quellcode 6.

```
1    <ComponentA>
2        <ComponentB>
3            <ComponentC>
4                <ComponentD />
5            </ComponentC>
6        </ComponentB>
7    </ComponentA>
```

Quellcode 6: Hierarchie von Komponenten in React

In einem ähnlichen Beispiel zeigt Fedosejev: Wenn die Komponente `ComponentD` den Zustand von der Rootkomponente `ComponentA` mutieren will, muss die Callback-Funktion über die `props` an `ComponentB` und `ComponentC` weitergeleitet werden, bis sie schließlich an `ComponentD` übergeben wird. Auf gleicherweise muss der Zustand über die `props` durch jede Hierarchieebene geleitet werden, wenn die Komponente `ComponentD` den Zustand von `ComponentA` darstellen will. [7] Dabei betont Fedosejev:

> *"Das schaut vielleicht wie eine Zeitverschwendung aus, aber dieses Design vereinfacht das Debuggen und das Nachvollziehen, wie unsere Applikation funktioniert."* [7, Seite 56]

Gleichzeitig erwähnt Fedosejev, dass es immer Strategien geben wird, um die Architektur der Applikation zu optimieren. [7] Eine davon ist Flux, welche im nächsten Kapitel näher erklärt wird.

2.3 Flux

Flux wurde zum ersten Mal auf der Facebook F8 Konferenz, im April 2014, als die Lösung für die Probleme von MVC vorgestellt. [10] Flux ist eine von Facebook entwickelte Architektur, die mit den folgenden Akteuren, den Datenfluss einer Applikation regelt [5]:

- Action
- Dispatcher
- Store
- View

2.3.1 Komponenten

Actions

Actions sind einfach Objekte, welche mindestens ein Attribut namens *“type”* haben. Mit diesem Attribut können die Stores erkennen, ob die empfangene Action für sie geltend ist. Weitere Attribute werden als Payload verwendet, um Daten an die Stores zu senden, welche für die Veränderung des Zustands relevant sind. [5]

Store

Der Applikations-Zustand wird in ein oder mehreren Stores verwaltet, welche auf die vom Dispatcher zugestellten Actions reagieren und den Zustand verändern können. [5]

Dispatcher

Der Dispatcher entnimmt Actions entgegen und leitet sie an alle Stores weiter. Wichtig dabei ist, dass es nur einen Singleton Dispatcher gibt, welcher Actions nur der Reihe nach verabeiten kann. [5]

View

In der View werden Daten der Stores dargestellt. Wenn die View von seinem Store benachrichtigt wird, dass sich der Zustand verändert hat, kann die View mit den neuen Daten neu gerendert werden. Auch werden Actions normalerweise von der View weitergeleitet, wenn der User mit dem Userinterface interagiert. [5]

2.3.2 Datenfluss

Das besondere an Flux ist, so Chen, dass der Datenfluss, im Gegensatz zu MVC, nur in eine Richtung geht. [10] Dieser kann in drei vereinfachten Schritten erklärt werden und wird in Abbildung 2 dargestellt:

1. Die View sendet eine Action an den Dispatcher.
2. Der Dispatcher leitet die Action an alle Stores weiter.
3. Die Stores senden die neuberechneten Daten an die View.

2.3.3 Zustandsmanagement

Der Applikationszustand wird in ein oder mehreren Stores verwaltet, welche in gewisser Weise die Rolle des Models von MVC übernehmen. [5]

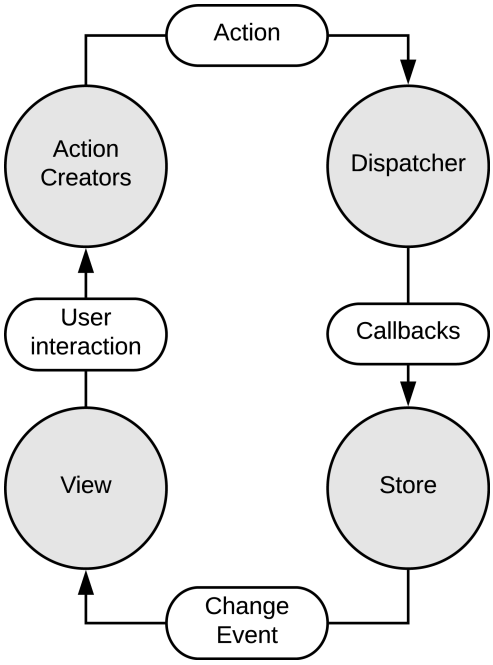

Abbildung 2: Datenfluss einer Flux-Applikation

2.4 Redux

Redux ist eine konkrete Implementierung von Flux, jedoch unterscheidet sie sich in dem Sinn, dass es von einigen Ideen der Elm-Architektur [4] inspiriert wurde. [11] Im Gegesatz zu Flux folgt Redux drei Prinzipien, die aus der Elm-Architektur stammen und gleichzeitig die signifikantesten Unterschiede zu Flux darstellen:

- Eine einzige Quelle der Wahrheit.
- Der Zustand ist unveränderlich.
- Änderungen werden mit Puren Funktionen gemacht.

Das bedeutet, dass es nur noch einen einzigen Store gibt, wo der gesamte Applikationszustand gespeichert wird. Dieser ist eine unveränderliche (*immutable*) Datenstruktur. Dadurch lässt sich der Zustand nicht direkt manipulieren. Um ihn verändern zu können, muss man ihn ersetzen. Deswegen ist die Datenveränderungslogik in eine Pure Funktion namens “Reducer” ausgelagert worden. (siehe Kapitel 2.4.1).

2.4.1 Komponenten

Actions, ActionTypes und ActionCreators

Wie durch die Flux-Architektur empfohlen, sind Actions in Redux einfache JS-Objekte, die zumindest das Attribut `type` haben. Jedes zusätzliche Attribut ist optional und kann verwendet werden um andere relevante Daten mitzuschicken. In Quellcode 7 ist zu sehen, dass das `type`-Attribut ein einfacher String ist. Es ist möglich den String direkt im Objekt zu definieren, jedoch

ist es best practice ActionTypes in einer externen Datei zu definieren, damit sie im Reducer wiederverwendet werden können. Außerdem ist in Quellcode 7 zu sehen, dass eine Funktion `someAction(_name, _age);` definiert wurde, welche ActionCreator genannt wird. Diese Funktion hat als Aufgabe Actions zu erstellen. Laut der Redux Dokumentation werden Actions dadurch portierbar und einfach zu testen. [11]

```
1    // ActionTypes.js
2    const SOME_ACTION = "SOME_ACTION";
3    // Actions.js
4    const someAction = (_name, _age) => ({
5        type: SOME_ACTION,
6        payload: { name: _name, age: _age }
7    });
```

Quellcode 7: Actions, ActionTypes und ActionCreators in Redux

Reducer

Der Reducer ist eine Pure Funktion, welcher den aktuellen Applikationszustand und die empfangene Action als Parameter übergeben bekommt. Der Reducer verändert den Zustand nicht direkt, er erstellt lediglich eine Kopie mit den benötigten Veränderungen und gibt den neuen Zustand als Resultat zurück. [11] Mit diesem Resultat wird dann der Zustand im Store ersetzt. Ein Beispiel für einen Reducer wird anhand des berühmten Counter-Beispiels in Quellcode 8 gezeigt. Dabei sind `INCREASE` und `DECREASE` die benötigten ActionTypes für das Beispiel.

```
1    const initialState = { count: 0 };
2    const reducer = (state = initialState, action) => {
3        switch (action.type) {
4            INCREASE:
5                return { count: state.count + 1 };
6            DECREASE:
7                return { count: state.count - 1 };
8        }
9    }
```

Quellcode 8: Beispiel für einen Reducer in Redux

Store

Der Store ist die Datenstruktur, welche den gesamten Applikationszustand hält und die Views notifiziert, wenn sich der Zustand geändert hat. [11] In Redux ist der Store ein einfaches JS-Objekt mit einem oder mehreren Name-Value Paaren. Die Struktur des Objekts ergibt sich aus dem Resultat des Reducers.

Damit React-Komponenten mit dem Store kommunizieren können, stellt Redux die Funktion `connect()` zur Verfügung.

Komponenten die mit der Funktion `connect()` generiert werden, werden Container-Komponenten genannt. Mit dieser Funktion kann festgelegt werden, welcher Teil des Zustands der Komponente übergeben werden soll und welche Actions die Komponente an den Store abschicken kann.

Komponenten die mit dem Store kommunizieren werden in zwei Komponenten geteilt. Dabei besteht die eine Hälfte aus der zuvor genannten Container-Komponente und die andere Hälfte aus einer Presentational-Komponente. Zweiteres legt fest, wie die Komponente aussieht. Tabelle 1, welche aus der Redux-Dokumentation stammt, zeigt die Unterschiede beider Komponentenarten deutlicher.

Tabelle 1: Der Unterschied zwischen Presentational- und Container-Komponenten. [11]

	Presentational-Komponente	Container-Komponente
Zweck	Wie Dinge aussehen (markup, styles)	Wie Dinge funktionieren (Daten holen, Zustandsveränderungen)
Kenntnis von Redux	Nein	Ja
Daten lesen	Daten lesen über `props`	Den Redux-Zustand abonnieren
Daten manipulieren	Callback-Funktionen aufrufen über `props`	Redux-Actions abschicken
Sind geschrieben	Händisch	Normalerweise generiert durch React Redux

2.4.2 Datenfluss

Da Redux eine konkrete Implementierung von Flux ist, ist der Datenfluss einer Redux-Applikation ebenfalls einfachgerichtet. [11] Im Gegegensatz zum Flux-Datenfluss, liegt im Redux-Datenfluss-Diagram in Abbildung 3 kein Dispatcher in der Kommunikationskette. Dieser fällt in Redux weg, da es nur noch einen einzigen Store gibt.

2.4.3 Zustandsmanagement

Ein Unterschied zu der Flux-Architektur liegt darin, dass Redux nur einen einzigen Store besitzt. Das heißt, der ganze Applikationszustand befindet sich im Store.

Umso größer eine Applikation jedoch wird, desto mehr Zustände muss die Applikation folglich verwalten. Um die Übersicht nicht zu verlieren, bietet Redux die Funktion

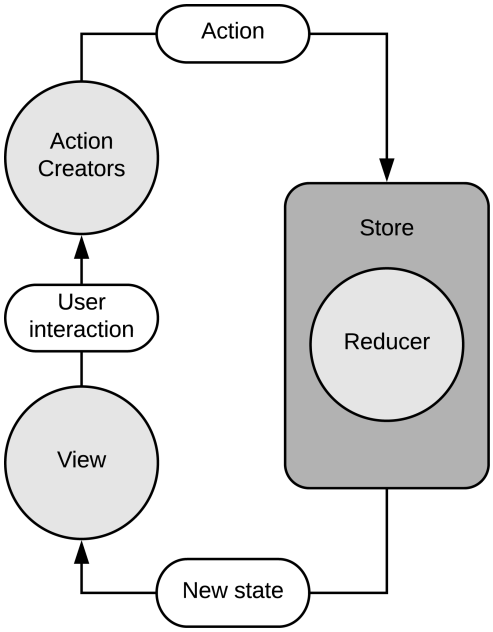

Abbildung 3: Datenfluss einer Redux-Applikation

`combineReducers()` an, welche mehrere (auch verschachtelte) Reducer zu einem Reducer kombiniert. So ist der Programmierer in der Lage mehrere kleine Reducer zu definieren, die jeweils unabhängige Teile des Zustands verwalten können. [11]

3 Fallstudie

In dieser Fallstudie wird die gleiche Applikation mit unterschiedlichen Implementierungsarten programmiert. In beiden Fällen wird das React Framework verwendet. Applikation A wird ausschließlich mit Klassenkomponenten implementiert, welche ihren eigenen Zustand verwalten. Applikation B hingegen, wird ausschließlich mit Funktionalen Komponenten implementiert, welche keinen Zustand haben. Dabei wird der Zustand in Applikation B, mithilfe des Redux Frameworks verwaltet.
Beide Implementierungsarten werden miteinander verglichen und ausgewertet um die Ziele dieser Arbeit zu erreichen.

3.1 Anforderungen der Applikation

3.1.1 Funktionale Anforderungen

- Ein Dish muss folgende Attribute haben:

  - Name : string
  - Preis : number
  - Zusatztext : string
  - Vegetarisch : boolean
  - Schärfegrad : number
- Das Hinzufügen, Aktualisieren und Löschen von Dishes
- Eine Live Preview Anzeige
- Eine Undo/Redo Funktion

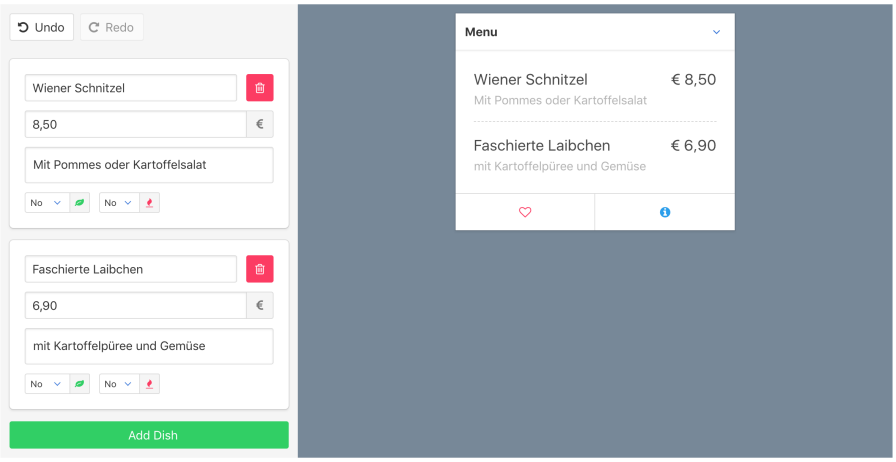

Abbildung 4: Screenshot der Applikation

3.2 Applikation A

Applikation A wurde ohne Redux und ausschließlich mit Klassenkomponenten implementiert. Der Quellcode dieser Applikation, inklusive einer Live-Demo, ist auf GitHub unter https://github.com/reakAleek/react-without-redux-example-application erhältlich.

3.2.1 NPM Abhängigkeiten

- bulma: 0.6.2
- lodash: 4.17.4
- prop-types: 15.6.0
- react: 16.2.0
- react-dom: 16.2.0
- react-tippy: 1.2.2

3.2.2 Aufbau

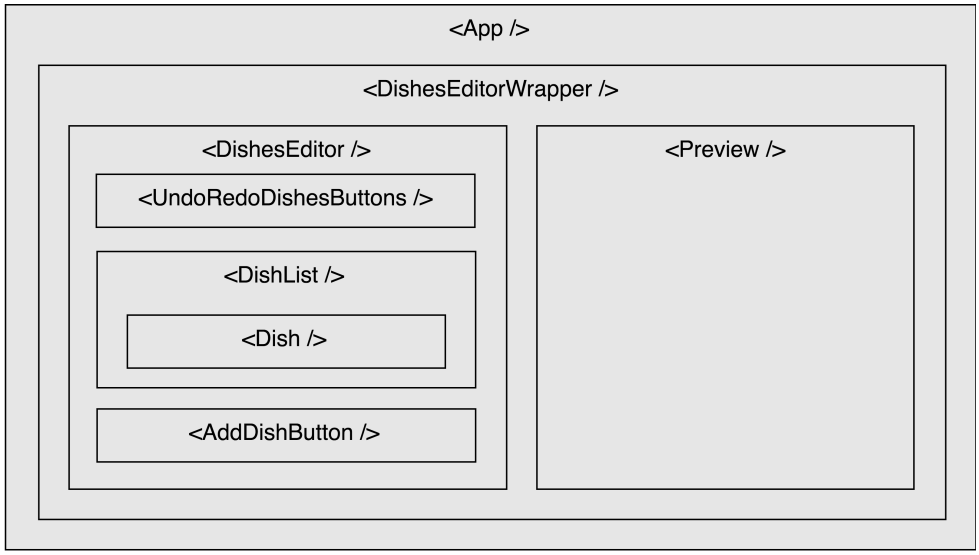

Abbildung 5: Aufbau der Applikation A

Wie in Abbildung 5 abgebildet, besteht die Applikation aus acht Klassenkomponenten. Die Zusammensetzung der Komponenten ist stark eingeschränkt, da der Datenfluss der Komponenten nach unten verläuft und deswegen Datenabhängigkeiten entstehen. Damit die Komponente DishesEditor und die Komponente Preview miteinander kommunizieren können, müssen beide unter der DishesEditorWrapper-Komponente liegen, da eine direkte Geschwisterkomponentenkommunikation nicht möglich ist. Gleichermaßen müssen die UndoRedoDishesButtons-, die DishList- und die AddDishButton-Komponente innerhalb der Dishes-Komponente deklariert werden, damit diese die Daten und die Funktionalität der DishesEditor-Komponente übernehmen können.

3.2.3 Die Komponenten im Detail

App-Komponente Die `App`-Komponente ist der Einstiegspunkt der Applikation und beinhaltet die anderen vier Komponenten.

DishesEditor-Komponente Die `DishEditor`-Komponente beinhaltet den Zustand der Applikation und die Logik für das Hinzufügen, Aktualisieren und Entfernen von Dishes, sowie die Undo/Redo-Logik.

DishList-Komponente Diese Komponente erhält eine Liste von Dishes von ihrer Elternkomponente `DishesEditor`. Sie wandelt diese Liste in eine Liste von `Dish`-Komponenten um und übergibt ihr die nötigen Daten.

Dish-Komponente Die `Dish`-Komponente enthält die Inputfelder, die Dropdownfelder und den Delete-Button. Die Input- und die Dropdownfelder hören auf das `onChange`-Event und leiten die Daten mithilfe von Callback-Funktionen an die Elternkomponenten, bis sie schließlich in der `DishesEditor`-Komponente in den Zustand aufgenommen wird.

DishesEditorWrapper-Komponente Die `DishesEditorWrapper`-Komponente umhüllt die `DishesEditor`- und die `Preview`-Komponente, damit die diese miteinander einfacher kommunizieren können.

AddDishButton-Komponente In der `AddDishButton`-Komponente ist nur ein Button, welcher bei einem Klick die Callback-Funktion ihrer Elternkomponente ausführt. Diese Funktion fügt dem Zustand ein leeres Dish-Element hinzu.

UndoRedoDishesButtons-Komponente Diese Komponente beinhaltet zwei Buttons, den Undo- und den Redo-Button. Beide Buttons rufen ebenfalls die Callback-Funktion ihrer Elternkomponente `DishesEditor` auf, welche die Undo- und Redo-Funktionalität darstellt.

Preview-Komponente Die `Preview`-Komponente stellt die Daten von der `DishesEditor`-Komponente in Echtzeit dar. Sie erhält ihre Daten über die `DishesEditorWrapper`-Komponente, welche widerrum ihre Daten von der `DishesEditor`-Komponente bekommt.

3.2.4 Zustandsmanagement und Datenfluss

Der Domainzustand der Applikation A befindet sich in der `DishesEditor`-Komponente.

Über die `Dish`-Komponente kann der Zustand durch Userinput manipuliert werden, indem sie ihre Elternkomponente `DishList` notifziert und die Werte des `inputChange`-Events über eine Callback-Methode weiterleitet. Die `DishList`-Komponente leitet die erhaltenen Daten umgehend an ihre Elternkomponente `DishEditor` auf die gleiche Weise weiter. Angelangt bei der `DishesEditor`-Komponente wird der Zustand mit neuen Daten aktualisiert. Des Weiteren entnehmen auch die `AddDishButton`-Komponente und die UndoRedoDishesButtons-Komponente Userinteraktion entgegen und notifzieren die `DishesEditor`-Komponente, welche dann die entsprechende Logik ausführt.

Bei jeder Veränderung des Zustands in der `DishesEditor`-Komponente, werden die neuen Daten ebenfalls über ein Callback-Methode an die `DishesEditorWrapper`-Komponente geschickt. Diese übermittelt die Daten nochmals weiter an die `Preview`-Komponente. Da in React keine direkte Geschwisterkommunikation möglich ist, fungiert die `DishEditorWrapper`-Komponente als Mittelsmann zwischen der `DishesEditor`-Komponente und der `Preview`-

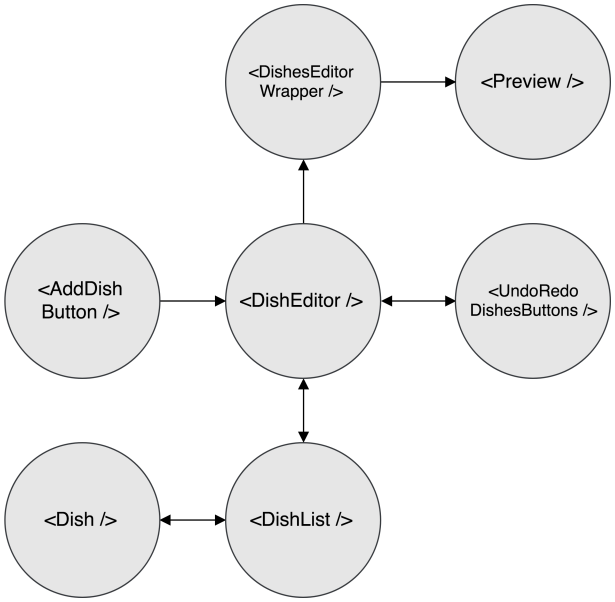

Abbildung 6: Datenfluss in Applikation A.

Komponente.

Abschließend verdeutlicht Abbildung 6, dass der Datenfluss von Applikation A sowohl unidirektional als auch bidirektional sein kann.

3.2.5 Undo/Redo-Feature

Für die Undo- und Redo-Funktionalität wurde der Algorithmus, welcher in der Redux-Dokumention erhältlich ist, verwendet. Dieser besagt, dass die Struktur des Undo/Redo-Zustands immer gleich ist, unabhängig von seinem Datentyp. [11]

```
1    {
2        past: Array<T>,
3        present: T,
4        future: Array<T>
5    }
```

Quellcode 9: Struktur des Zustands einer Undo–und Redo–Historie

Bei jedem Aufruf einer Methode in der `DishesEditor`-Komponente, die den Zustand aktualisien soll, wird der `present`-Zustand dem `past`-Array hinzugefügt und der `present`-Zustand mit dem neuen Zustand ersetzt. Wenn die `onUndo`-Methode aufgerufen wird, so wird der `present`-Zustand dem `future`-Array hinzugefügt. Gleichzeitig wird das jüngste Element in der `past`-Array aus dem Array entfernt und für den `present`-Zustand eingesetzt. In der gleichen Weise wird das jüngste Objekt des `future`-Arrays entfernt und für das `present`-Objekt verwendet, wenn die `onRedo`-Methode ausgeführt wird.

Die Logik für das Hinzufügen, Aktualisieren und Entfernen von `Dishes` ist sehr stark mit der Logik für das Erstellen der Undo- und Redo-Historie gekoppelt. Teilweise befinden sich sogar beide Logiken in der gleichen Methode, wie man in Abbildung 10 sehen kann.

```
1     ...
2     deleteDish = (id) => () => {
3         this.setState((prevState) => ({
4             past: [prevState.present, ...prevState.past],
5             present: {
6                 byId: _.omit(prevState.present.byId, id),
7                 allIds: _.without(prevState.present.allIds, id)
8             },
9             future: []
10        }), this.updateParentState);
11    }
12    ...
```

Quellcode 10: Ausschnitt aus der DishesEditor–Komponente aus Applikation A

3.3 Applikation B

Applikation B wurde mit Redux und ausschließlich mit Funktionalen Komponenten implementiert. Der Quellcode dieser Applikation, inklusive einer Live-Demo, ist auf GitHub unter https://github.com/reakAleek/react-redux-example-application erhältlich.

3.3.1 NPM Abhängigkeiten

- bulma: 0.6.2
- lodash: 4.17.4
- prop-types: 15.6.0
- react: 16.2.0
- react-dom: 16.2.0
- react-redux: 5.0.6
- react-tippy: 1.2.2
- redux: 3.7.2
- redux-logger: 3.0.6

3.3.2 Aufbau

In Abbilung 7 sind zwar sieben Komponenten zu sehen, jedoch ist diese Abbildung irreführend. Denn die Provider-Komponente ist eine von Redux zur Verfügung gestellte Komponente, die nicht selber definiert werden muss. Außerdem gibt es für jede Komponente die mit dem

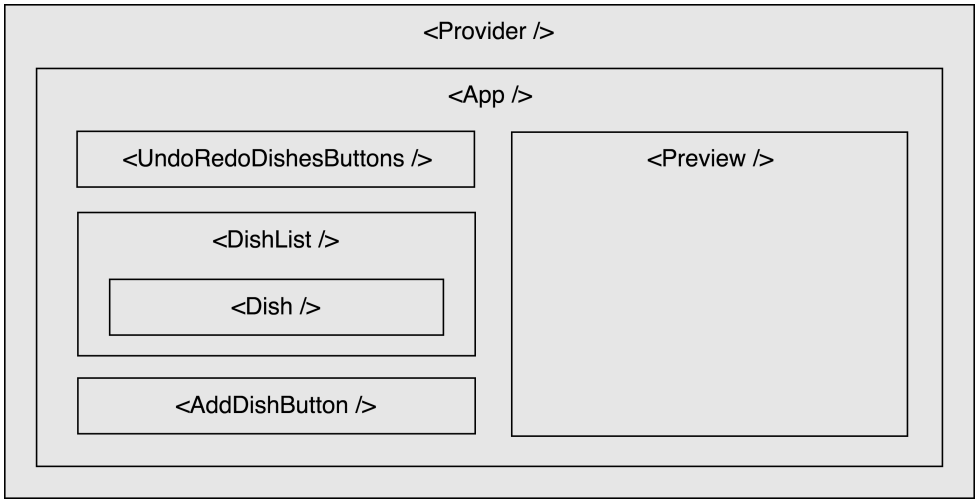

Abbildung 7: Aufbau der Applikation B

Redux-Store verbunden ist einen Container- und einen Presentational-Teil. Somit werden in Applikation B insgesamt 11 Komponenten definiert. Mit Ausnahme der Dish-Komponente, hat keine Komponente eine Datenabhängigkeit von seiner Elternkomponente. Da die Daten aus dem Redux-Store stammen und jede Komponente mit ihm verbunden werden kann, is die Zusammensetzung der Komponenten sehr flexibel.

Actions

Applikation B verfügt über fünf Actions mit folgenden ActionType-Identifier:

- `ADD_DISH`
- `REMOVE_DISH`
- `UPDATE_DISH`
- `UNDO_DISHES`
- `REDO_DISHES`

Reducer

Für jeden der oben genannten ActionTypes gibt es einen adäquaten Zweig im Reducer. In den jeweiligen Zweigen befindet sich die Logik für das Hinzufügen, Entfernen und Aktualisieren, sowie die Undo/Redo-Logik.

3.3.3 Die Komponenten im Detail

Provider-Komponente Die Provider-Komponente ist eine von Redux zur Verfügung gestellte Komponente. Sie umhüllt die App-Komponente und ermöglicht jeder Komponente eine Verbindung zum Redux-Store.

App-Komponente Die App-Komponente ist eine reine Presentational-Komponente.

UndoRedoDishesButtons-Komponente Diese Komponente ist mit dem Redux-Store verbunden. Sie enhält den Undo- und den Redo-Button. Bei einem `onClick`-Event werden die entsprechenden Actions `UNDO_DISHES` und `REDO_DISHES` an den Store geschickt. Gleichzeitig erhält die Komponente Informationen, ob der Button deaktiviert sein soll oder nicht.

DishList-Komponente Die DishList-Komponente ist ebenfalls mit dem Redux-Store verbunden und erhält von ihm eine Liste von Dishes und bildet diese auf eine Liste von Dish-Komponenten ab.

Dish-Komponente Wie in Applikation A enhält auch in Applikation B die Dish-Komponente Inputfelder, Dropdownfelder und den Delete-Button. Die Dish-Komponente erhält ihre Daten von ihrer Elternkomponente DishList. Trotzdem ist sie mit dem Redux-Store verbunden, um die Actions `UPDATE_DISH` und `REMOVE_DISH` an den Redux-Store senden zu können.

AddDishButton-Komponente Die AddDishButton-Komponente beinhaltet einen Button, dieser sendet die Action `ADD_DISH` an den Store und fügt bei einem Klick der Datenstruktur ein leeres Dish hinzu.

Preview-Komponente Die Preview-Komponente erhält ihre Daten auch direkt vom Store und aktualisiert nach jeder Action automatisch.

3.3.4 Zustandsmanagement und Datenfluss

Der Datenfluss von Applikation B, wie in Abbildung 8 dargestellt, erfolgt unidirektional. Bei Userinteraktion mit den Komponenten Dish, AddDishButton und UndoRedoDishesButtons wird eine Action an den Store gesendet. Durch den ActionType kann der Reducer die entsprechende Logik ausführen und ersetzt mit dem Ergebnis den Zustand des Stores. Sobald der neue Zustand gesetzt ist, werden die DishList-Komponente und die Preview-Komponente vom Redux-Store benachrichtigt, dass sie mit den neuen Daten neu gerendert werden müssen.

3.3.5 Undo/Redo-Feature

In Applikation B wird der gleiche Algorithmus wie in Applikation A verwendet. (siehe Kapitel 3.2.5) Allerdings befindet sich die Logik für den Undo/Redo-Algorithmus in einem *Reducer En-*

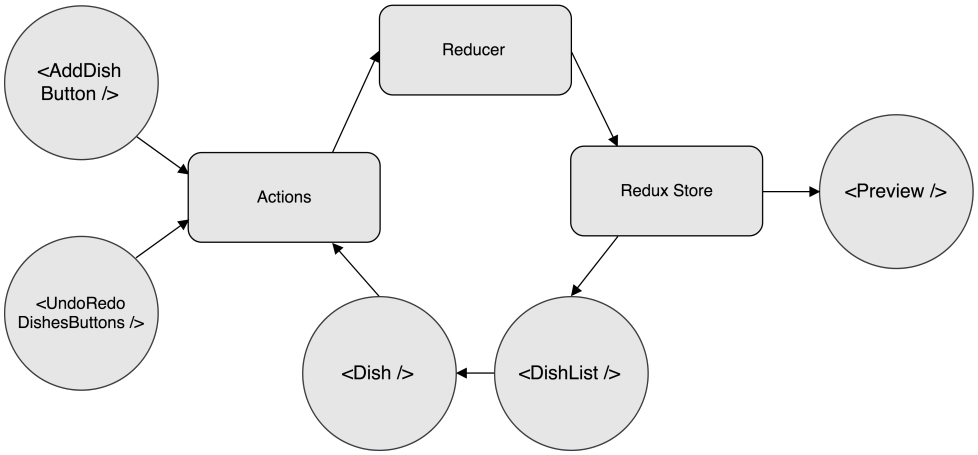

Abbildung 8: Datenfluss der Applikation B

hancer. Ein *Reducer Enhancer* kann um jeden Reducer umhüllt werden und gibt dem umhüllten Reducer seine Funktionalität weiter.

3.4 Applikation A vs Applikation B

3.4.1 Zustandsmanagement und Datenfluss

Beide Applikationen verwalten den Applikationszustand an einer Stelle. Applikation A haltet den gesamten Zustand in der `DishesEditor`-Komponente, in Applikation B hingegen, wird der Zustand im Redux-Store gelagert. Der Unterschied liegt jedoch darin, wie die Komponenten auf diesen Zustand zugreifen und wie sie ihn manipulieren.

In Applikation A müssen die Daten und die Callback-Funktionen an jede Komponente in der Komponentenhierarchie übergeben werden um von der Komponente X and die Komponente Y zu gelangen. Konkret bedeutet das: Damit der Userinput aus der `Dish-Komponente` bis zur `Preview`-Komponente gelangt, müssen die Daten von der `Dish-Komponente` zuerst an der `DishList`-Komponente, der `DishEditor`-Komponente und der `DishEditorWrapper`-Komponente vorbei, bis sie letztendlich in der `Preview`-Komponente ankommen.

Im Gegensatz dazu, kann der Zustand in Applikation B von jeder Komponente direkt ausgelesen und manipuliert werden. Das heißt, dass der Userinput aus der `Dish`-Komponente über eine Action an den Redux-Store geschickt wird. Dieser notifiziert die `Preview`-Komponente, dass sich der Zustand verändert hat und löst somit das neurendern der Komponente aus.

3.4.2 Kombinierbarkeit der Komponenten

In Applikation A ist es nicht möglich die Komponenten beliebig in der Komponentenhierarchie zu platzieren, da Datenabhängigkeiten zwischen den Komponenten herrschen. Weil in

Applikation B jede Komponente mit dem Store verbunden werden kann, bestehen mit Ausnahme der Dish-Komponente keine Datenabhängigkeiten zwischen den Komponenten. Deswegen ist es möglich die Komponenten an beliebiger Stelle zu deklarieren.

3.4.3 Boilerplate-Code

Im Verhältnis zu Applikation A benötigt Applikation B aufgrund der Redux-Bibliothek mehr Boilerplate-Code. Für jede Handlung in der Applikation müssen ActionTypes, ActionCreators und Actions definiert, sowie ein Zweig im Reducer angelegt werden. Hinzu kommt, dass jede Komponente die mit dem Redux-Store kommunizieren muss, zusätzlich eine Container-Komponente benötigt. Die Auswirkungen sind besonders in den Dateigrößen beider Source-Ordner zu sehen. (Tabelle 2) Applikation B hat einen Zuwachs von 40% in der Dateigröße gegenüber seiner Vergleichsapplikation.

Tabelle 2: Applikation A vs Applikation B

	Applikation A	Applikation B
Dateigröße des Source-Ordners	49kb	70kb
Dateigröße der kompilierten JS-Datei	456kb	484kb
Anzahl der Dateien im Source-Ordner	9	15
Anzahl der Codelinien im Source-Ordner	460	495
Anzahl der Komponenten	8	11

3.4.4 Undo/Redo-Feature

Da der Zustand in Applikation A und der Zustand in Applikation B beide wie unveränderliche Datenstrukturen behandelt werden und beide an einer Stelle gesammelt sind, konnte der gleiche Algorithmus in beiden Applikationen verwendet werden. Das macht die Implementierung der Undo/Redo-Funktion trivial, da bei jeder Interaktion der gesamte Zustand in eine Datenstruktur (zB ein Array) gepeichert werden muss.

Allerdings ist in Applikation A nicht gewährleistet, dass der gesamte Applikationszustand in der `DishesEditor`-Komponente bleibt. Jede neue Klassenkomponente, könnte ihren eigenen Zustand verwalten. Demnach wäre der Zustand auf mehrere Komponenten verteilt und macht den Algorithmus um komplexer.

Im Gegenteil dazu wird der gesamte Zustand in Applikation B immer im Redux-Store bleiben, auch wenn neue Features hinzugefügt werden.

3.4.5 Wartbarkeit

Da die Komponenten in Applikation B durch Redux flexibel zusammengesetzt werden können, lassen sich Komponenten in Applikation B leichter als in Applikation A umpositionieren. Deswegen kann Applikation B einfacher und schneller umstrukturiert werden. Weiters können das Aussehen und die Logik von Komponenten unabhängig voneinander verändert werden, da diese nochmals in Container- und Presentational-Komponenten geteilt werden. Hinzu kommt, dass sich die Logik und der Applikationszustand gesammelt an ihren vorgesehenen Stellen befinden. Dadurch können Features einfacher und schneller gewartet werden, da sogar neue Mitentwickler sofort den zu bearbeitenden Codeteil finden und ihn reparieren können. Dem ungeachtet ist die Wartbarkeit in Applikation A nicht mühsamer als in Applikation B, da die Größe der Applikation überschaubar ist und in Applikation A der Zustand und die Logik auch an einer Stelle zu finden sind.

3.4.6 Erweiterbarkeit

In einer Applikation die Redux verwendet ist die Vorgehensweise, wie neue Features hinzugefügt werden, exakt beschrieben, wobei die Reihenfolge folgender Punkte irrelevant ist:

- Der Programmierer definiert eine Action und den entsprechenden ActionCreator.
- Dem Reducer wird ein neuer Zweig beigefügt, der die Action erkennt und die neue Logik ausführt.
- Eine Container- und eine Presentational-Komponente werden erstellt, um die Action an den Store schicken zu können und gegebenenfalls die Daten wieder anzeigen zu können.

Durch den gegebenen Workflow ist es einfach die Applikation mit neuen Features zu erweitern. Weiters können bestehende Features, aus den gleichen Gründen die im Punkt Wartbarkeit gegeben sind, einfach modifiziert und ausgebaut werden. In einer Applikation ohne Redux hingegen, ist ein gleichwertiger Workflow nicht festgelegt. Die Logik und der Zustand können auf mehrere Komponenten verteilt sein, was wiederum bedeutet, dass Komponenten über mehrere Ecken miteinander kommunizieren müssen. Das Erweitern von React-Applikationen ohne Redux bringt einen größeren Aufwand mit sich, weil mehr Gedanken in den Aufbau und die Planung der Komponentenhierarchie gesteckt werden muss, damit Komponenten miteinander kommunizieren können.

Ein konkretes Problem für die Erweiterbarkeit in Applikation A ist, dass die Logik für das Hinzufügen, Aktualisieren und Entfernen von `Dishes` und die Logik für die Undo/Redo-Funktion sehr stark miteinander gekoppelt sind. Der Algorithmus muss neu geschrieben werden, wenn ein neues Feature hinzugefügt wird und folgende Punkte auf das neue Feature zutreffen:

- Das Feature verwaltet seinen eigenen Zustand.
- Das Feature wird außerhalb der `DishesEditor`-Komponente deklariert.
- Das neue Feature soll auch in die Undo-Historie aufgenommen.

4 Konklusion

Das erste Ziel dieser Arbeit, sich *“Einen Überblick darüber zu verschaffen, wie die Flux-Architektur, im Gegensatz zur MVC Architektur, mit Zustandsveränderungen umgeht”*, wurde im Flux’ Datenfluss Kapitel 2.3.2 bearbeitet. Dabei hat sich ergeben, dass Flux im Gegensatz zu MVC, den Datenfluss von Zuständsveränderungen einfachgerichtet haltet, um das Verständnis des Datenflusses einfach zu halten.

Die Ergebnisse vom zweiten Ziel *“Zu untersuchen wie Redux, eine konkrete Implementierung von Flux, in Verbindung mit React, durch funktionale Prinzipien, das Zustandsmanagement optimieren kann.”*, wurden in der Fallstudie (Kapitel 3) behandelt. Diese liefen darauf hinaus, dass die funktionalen Prinzipien, die Redux mit sich bringen, keinen besonderen Vorteil im Zustandsmanagement darstellt. React selbst folgt schon funktionalen Prinzipien wie Unveränderlichkeit und Puren Funktionen. Somit können die gleichen Strategien implementiert werden. Jedoch hat sich herausgstellt, dass in React-Applikationen mithilfe von Redux eine flexiblere Komponentenstruktur zusammengesetzt werden kann. Es bestehen weniger bis keine Datenabhängigkeiten zwischen den Komponenten, somit sind sie frei platzierbar.

Drittens, *“Zu eruieren, ob eine React-Applikation mit der Redux-Bibliothek leichter zu warten und zu erweitern ist.”*, wurde ebenfalls in der Fallstudie (Kapitel 3) bewertet und ergab, dass die Wartbarkeit von der Größe der Applikation abhängt. Bei dem Umfang der Applikationen aus der Fallstudie ist kein bemerkenswerter Unterschied zwischen beiden Applikation erkennbar, da in beiden Fällen der Zustand und die Logik gesammelt an einer Stelle zu finden sind. Hinsichtlich Erweiterbarkeit, wurde festgestellt, dass durch die Verwendung von Redux klare Richtlinien für das Erstellen von Features gelten und der Code somit auch bei Vergrößerung des Projekts einheitlich und übersichtlich bleibt. In Applikation A dagegen, ist die Logik für das Hinzufügen, Aktualisieren und Löschen von Dishes zu sehr mit der Undo/Redo-Logik gekoppelt. Dies macht es schwierig neue Features hinzuzufügen, die auch in die Undo-Historie mitaufgenommen werden sollen.

Der letzte Punkt, *“Und zu ermitteln, wie einfach oder schwer es ist, eine Undo/Redo-Funktion mit Redux zu implementieren.”*, welches auch in der Fallstudie (Kapitel 3) aufgeklärt wird, hat ergeben, dass kein Unterschied in der Einfachheit oder Schwierigkeit der Implementierung einer Undo/Redo-Funktion zwischen beiden Applikationen ist. Das liegt aber daran, dass in beiden Applikatione der gesamte Zustand an einer Stelle verwaltet wird. Wenn die Applikation aber größer wird und sich der Zustand auf mehrere Komponenten verteilt und trotzdem alle Interaktionen in die Undo-Historie mit aufgenommen werden sollen, wird der Algorithmus um ein Vielfaches komplexer als der angewandte Algorithmus in der Fallstudie.
Allgemein lässt sich sagen, dass die Redux-Bibliothek die Qualität des Codes steigert. Voral-

lem, wenn die Applikation größer ist und mehrere Komponenten hat, die miteinander kommunizieren und Daten austauschen müssen. Kleinere Projekte hingegen profitieren weniger von der Bibliothek, da doch sehr viel Boilerplate-Code geschrieben werden muss, um ein Feature umzusetzen.

Literaturverzeichnis

[1] ANGULARJS: *AngularJS*. [Online] Verfügbar unter: <https://angularjs.org/> [Zugang am 04.04.2018].

[2] ASHKENAS, J.: *Backbone.js*. [Online] Verfügbar unter: <http://backbonejs.org/> [Zugang am 12.02.2018].

[3] BURBECK, S.: *Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)*. 1987, 1992.

[4] CZAPLICKI, E.: *The Elm Architecture*. [Online] Verfügbar unter: <https://guide.elm-lang.org/architecture/> [Zugang am 04.04.2018].

[5] FACEBOOK INC.: *Flux - APPLICATION ARCHITECTURE FOR BUILDING USER INTERFACES*. [Online] Verfügbar unter: <https://facebook.github.io/flux/> [Zugang am 12.02.2018].

[6] FACEBOOK INC.: *React - A JavaScript library for building user interfaces*. [Online] Verfügbar unter: <https://reactjs.org/docs/> [Zugang am 12.02.2018].

[7] FEDOSEJEV, A.: *React.js Essentials*. Packt Publishing Ltd., 2015.

[8] FOWLER, M.: *Patterns of Enterprise Application Architecture*. Addison Wesley, Boston, 2002.

[9] GOOGLE: *Angular - One Framework. Mobile & desktop.*. [Online] Verfügbar unter: <https://angular.io/docs> [Zugang am 12.02.2018].

[10] OCCHINO, T., J. CHEN und P. HUNT: *Hacker Way: Rethinking Web App Development at Facebook*, 2014. [Online] Verfügbar unter: <https://facebook.github.io/flux/> [Zugang am 24.01.2018].

[11] REDUX: *Redux Documentation*. [Online] Verfügbar unter: <https://redux.js.org/> [Zugang am 24.01.2018].

[12] REENSKAUG, T.: *The original MVC reports*, 2007.

[13] STATEOFJS.COM: *Front-end Frameworks – Worldwide Usage*. [Online] Verfügbar unter: <https://stateofjs.com/2017/front-end/worldwide/> [Zugang am 24.01.2018].

[14] YOU, E.: *Vue.js - The Progressive JavaScript Framework*. [Online] Verfügbar unter: <https://vuejs.org/> [Zugang am 12.02.2018].

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

WWW	world wide web
JS	JavaScript
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
MV*	Model-View-Whatever
SPA	Single-Page-Applikation
UI	Userinterface
DOM	Document Object Model