HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Martin Gralka

Gegenüberstellung zweier Frameworks zum Testen von React-Anwendungen

2017

Fakultät: **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Gegenüberstellung zweier Frameworks zum Testen von React-Anwendungen

Autor:
Herr Martin Gralka

Studiengang:
Medieninformatik & Interaktives Entertainment

Seminargruppe:
MI13w1-B

Erstprüfer:
Prof. Dr.-Ing. Frank Zimmer

Zweitprüfer:
M.Eng. Johannes Christian Siehler

Einreichung:
Mittweida, 05.09.2017

Faculty: **Applied Computer Sciences & Biosciences**

BACHELOR THESIS

Comparison of two frameworks for testing React applications

author:
Mr. Martin Gralka

course of studies:
Media Informatics and Interactive Entertainment

seminar group:
MI13w1-B

first examiner:
Prof. Dr.-Ing. Frank Zimmer

second examiner:
M.Eng. Johannes Christian Siehler

submission:
Mittweida, 05.09.2017

Bibliografische Angaben

Gralka, Martin:

Gegenüberstellung zweier Frameworks zum Testen von React-Anwendungen

Comparison of two frameworks for testing React applications

125 Seiten, Hochschule Mittweida, University of Applied Sciences,
Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2017

Abstract

Die JavaScript-Bibliothek React ist inzwischen seit mehreren Jahren eine beliebte und weit verbreitete Technologie im Bereich der Frontend-Webentwicklung. In Kombination mit Redux ermöglicht React die Erstellung funktionaler Benutzeroberflächen selbst für komplexe Anwendungen. Es stellt sich die Frage, wie solche React-Anwendungen am besten zu testen sind. Zwei häufig genutzte Test-Frameworks für diesen Zweck sind Mocha und Jest. Diese werden in dieser Bachelorarbeit an Hand zuvor definierter Qualitätskriterien in mehreren Kategorien miteinander verglichen und so auf ihre Tauglichkeit für das Testen von React-Applikationen untersucht. Als Schwerpunkt der Gegenüberstellung wird eine Reihe von Tests mit beiden Frameworks für eine bereits bestehende App auf Basis von React und Redux implementiert. Am Ende steht die Beantwortung der Frage, welches der beiden Frameworks auf Grundlage der Untersuchungsergebnisse für das Testen von React-Anwendungen vorzuziehen ist.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

BDD	Behaviour-Driven Development
CI	Continuous Integration
CSS	Cascading Style Sheets
DOM	Document Object Model
dt.	Deutsch
engl.	Englisch
HTML	Hypertext Markup Language
JS	JavaScript
NPM	Node Package Manager
o.J.	ohne Jahr
SR	Shallow Rendering
SUT	System Under Test
SW	Software
TC	Test Case (Testfall)
TDD	Test-Driven Development
TF	Test-Framework
TO	Testobjekt
TS	Testsuite
UI	User Interface
VDOM	Virtual Document Object Model

Vorwort

Die vorliegende Bachelorarbeit entstand in Zusammenarbeit mit der *Sensape GmbH* in Leipzig. Ein besonderer Dank gilt Christian Siehler von Sensape, der nicht nur die Idee für das Thema dieser Arbeit lieferte, sondern mir als Zweitbetreuer auch bei allen Fragen helfend zur Seite stand und stets wertvolle Anregungen lieferte.

Des Weiteren gilt mein Dank meinem Erstbetreuer, Herrn Prof. Zimmer, der mich ebenfalls bei allen auftretenden Fragen in Bezug auf diese Arbeit mit gutem Rat unterstützte.

Abschließend möchte ich mich noch bei allen Familienmitgliedern und Freunden bedanken, die geholfen haben, diese Bachelorarbeit Korrektur zu lesen.

1 Einleitung

1.1 Motivation

Bei der Erstellung von großen Softwareprojekten zählt das Testen zu den wichtigsten Phasen im Entwicklungsprozess. Nur durch ausgiebiges Testen kann sichergestellt werden, dass eine Anwendung wie erwartet funktioniert und nicht voller ungewollter Programmfehler – sogenannter Bugs – ist, welche die Nutzererfahrung schmälern oder das Programm gar komplett unbrauchbar machen. Nichtsdestotrotz ist der Softwaretest aber auch der Schritt, der am häufigsten ausgelassen wird oder zumindest auf der Prioritätenliste hintenangestellt wird, frei nach dem Motto: „Lieber eine schlecht funktionierende Software, als gar keine Software“. Fehlende Zeit oder einfach mangelnde Motivation der Entwickler, Softwaretests zu schreiben, können eine solche Denkweise hervorrufen.

Auch in der Webentwicklung sind Softwaretests genauso essenziell wie in jedem anderen Teilbereich der Softwareentwicklung. Die Webentwicklung ist eine besonders schnelllebige Branche, in der fast täglich neue Technologien entwickelt werden und für Aufsehen in der Branche sorgen. Nur wenige dieser halten dem Hype jedoch stand und können sich auf lange Sicht durchsetzen. Eine Technologie, die sich bereits seit einigen Jahren großer Beliebtheit erfreut, ist die JavaScript-Bibliothek *React*. Dass dies keine bloße Behauptung ist, wird schnell deutlich, wenn man sich eine Liste der Namen von Unternehmen durchliest, welche React verwenden, darunter namhafte Firmen wie *eBay*, *Yahoo* und *Netflix*.[1] Der wohl populärste Vertreter ist jedoch das Großunternehmen *Facebook*, welches React selbst mitentwickelt hat und für all seine Dienste verwendet: *Instagram*, *WhatsApp* und natürlich das gleichnamige soziale Netzwerk *Facebook*.

Es stellt sich die Frage, wie solche React-Anwendungen, die häufig in Verbindung mit weiteren Webtechnologien verwendet werden, zu testen sind. Test-Frameworks zum Schreiben von automatisierten Tests von JavaScript-Anwendungen gibt es viele, die besonderen Anforderungen von React reduzieren die Auswahl jedoch stark. Die Community ist gespalten, hauptsächlich zwischen den beiden Test-Frameworks Mocha und Jest, welche beide Befürworter als auch Gegner haben. Die Frage, welches der beiden Frameworks sich besser zum Testen von React-Anwendungen eignet, ist auch für die Firma *Sensape* aus Leipzig von Interesse. Eine der wichtigsten Anwendungen des Startups, die sogenannte Second Screen App, basiert auf React. Da diese Anwendung mitunter auf Messen zum

[1] Siehe https://github.com/facebook/react/wiki/sites-using-react [Stand 02.09.2017]

Einsatz kommt und dort täglich von Hunderten von Menschen benutzt wird, ist es von äußerster Wichtigkeit, dass hier keine unvorhergesehenen Fehler auftreten.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die beiden Test-Frameworks Mocha und Jest umfassend zu vergleichen und die Frage zu beantworten, welches von beiden sich für die Anforderungen des Unternehmens Sensape besser zum Testen von React-Anwendungen eignet. Auf Grundlage der Ergebnisse dieser Arbeit soll zum Abschluss eine Empfehlung ausgesprochen werden, welches der beiden Test-Frameworks zukünftig von Sensape für diesen Zweck verwendet werden sollte.

Zum qualitativen Vergleich beider Frameworks sollen messbare Qualitätskriterien aufgestellt werden, die eine bewertbare Gegenüberstellung beider Frameworks ermöglichen. Für jedes Kriterium soll anschließend eine solche Gegenüberstellung erfolgen. Als Schwerpunkte der Arbeiten sollen beide Test-Frameworks in die bereits bestehende Second Screen App eingebunden werden (separat voneinander) und eine Vielzahl an Testfällen aufgestellt werden, die anschließend mit Mocha und Jest in Form von Komponententests implementiert werden. Diese Tests bilden die Grundlage für eine fundierte Untersuchung. Bei der Implementierung soll insbesondere auch das sogenannte *Snapshot Testing*, welches ein exklusives Feature von Jest ist, ausgiebig auf dessen Eignung für das Testen von React-Anwendungen geprüft werden, und die Vor- und Nachteile im Vergleich zu „konventionellen“ Tests herausgestellt werden.

1.3 Abgrenzung der Arbeit

Die vorliegende Arbeit beschränkt sich auf automatisierte Softwaretests in Form von Komponententests für das Frontend von React-Anwendungen. Weitere Teststufen, z.B. der Systemtest, gehören nicht zum Untersuchungsbereich der vorliegenden Bachelorarbeit, ebenso wenig wie das Testen von nichtfunktionalen Anforderungen (z.B. Performanz oder Sicherheit).

Bei der Erstellung der Testfälle sollen lediglich einige ausgewählte Komponenten der Second Screen App berücksichtigt werden. Es ist keinesfalls das Ziel, eine vollständige Abdeckung des Codes durchs die Tests zu erreichen. Sollten durch die implementierten Tests mögliche Programmfehler entdeckt werden, ist deren Behebung ebenso kein Bestandteil dieser Thesis.

Besonders betont werden soll zudem, dass die vorliegende Bachelorarbeit keinen Anspruch auf Allgemeingültigkeit zur Beantwortung der Frage erhebt, welches der beiden Test-Frameworks das „Bessere“ ist, da auch in dieser Arbeit nur ausgewählte Aspekte betrachtet werden können.

2 Grundlagen der Webentwicklung

2.1 Frontend

Allgemein bezeichnet das Frontend (ebenso: Front-End, Front End) bei einer Software denjenigen Bereich, welcher näher am Benutzer liegt. Bei einer Web-Anwendung[2] (WA) bezeichnet das Frontend den Teil, der clientseitig (in der Regel durch einen Webbrowser) interpretiert und ausgeführt wird. Alle modernen Browser enthalten Parser für *Hypertext Markup Language* (*HTML*), *Cascading Style Sheets* (*CSS*) und *JavaScript* (*JS*). Diese drei Standard-Technologien bilden somit die Grundlage für die Frontend-Entwicklung, da nur sie von Webbrowsern tatsächlich interpretiert werden können. In der Praxis werden jedoch auch häufig Web-Frameworks oder Bibliotheken an Stelle von oder in Kombination mit diesen drei Standard-Technologien verwendet.

HTML ist eine Textauszeichnungssprache, die verwendet wird, um „die Struktur und Semantik der Inhalte eines Web-Dokuments [zu] beschreib[en]“ (MDN, 2017). Alle Webseiten und Web-Anwendungen sind HTML-Dokumente, was HTML zur grundlegenden Technologie der Frontend-Entwicklung macht. CSS hingegen wird verwendet, um das Aussehen einzelner HTML-Elemente festzulegen. Wird eine HTML-Seite vom Browser geladen, analysiert dieser den Quellcode und erstellt daraus das sogenannte *Document Object Model* (*DOM*). Hierbei handelt es sich um ein Abbild des HTML-Dokuments in Form eines logischen Baums, wobei jedes HTML-Element einem Knoten im DOM entspricht. Abbildung 1 zeigt an Hand eines Beispiel-Dokuments den Zusammenhang zwischen HTML-Dokument und dem zugehörigen DOM, das vom Browser erzeugt wird.

[2] Hier: Oberbegriff für sämtliche Client-Server-Anwendungen, einschließlich Websites

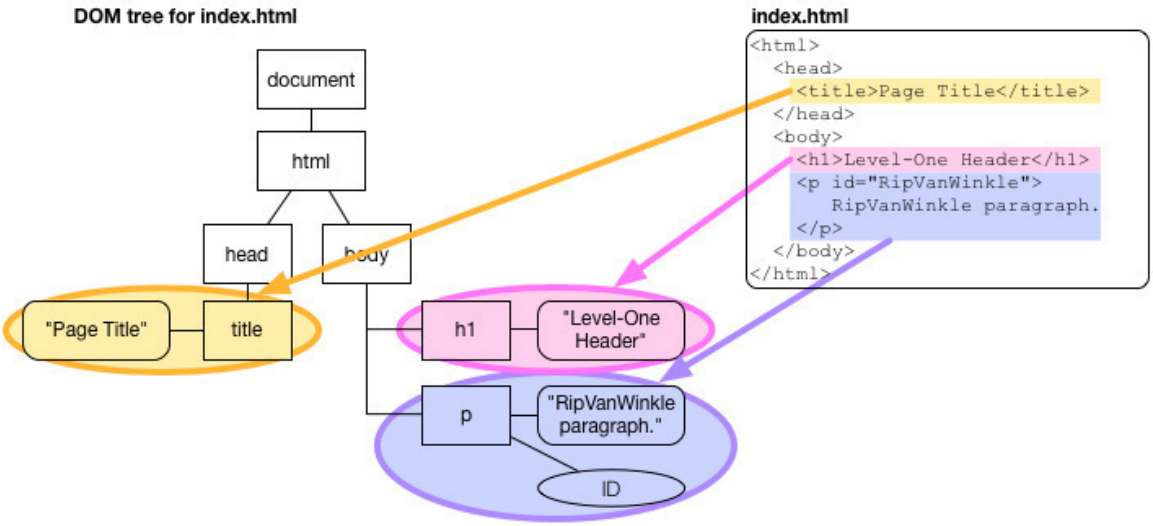

Abbildung 1: Zusammenhang zwischen HTML-Dokument und DOM

(Quelle: Dart (o.J. [ca. 2016]))

Auch CSS-Eigenschaften werden im DOM gespeichert. JS ermöglicht es, auf das DOM zuzugreifen und dieses, nachdem die Seite geladen hat, zu manipulieren, d.h. Knoten zu verändern, hinzuzufügen oder auch zu löschen. Dies wiederum resultiert in einer Aktualisierung der korrespondierenden HTML-Elemente und des zugehörigen CSS und somit einer Änderung der Benutzeroberfläche (*UI*; für engl.: *user interface*). In der Praxis geht dies häufig mit einer Benutzerinteraktion einher: Beispielsweise kann beim Drücken eines Buttons an einer anderen Stelle auf der Webseite ein Text (z.B. das Ergebnis einer Berechnung) angezeigt werden. JS ermöglicht somit über die DOM-Schnittstelle die Erstellung dynamischer und interaktiver WAs. Abbildung 2 stellt das zuvor erläuterte Prinzip schemenhaft dar.

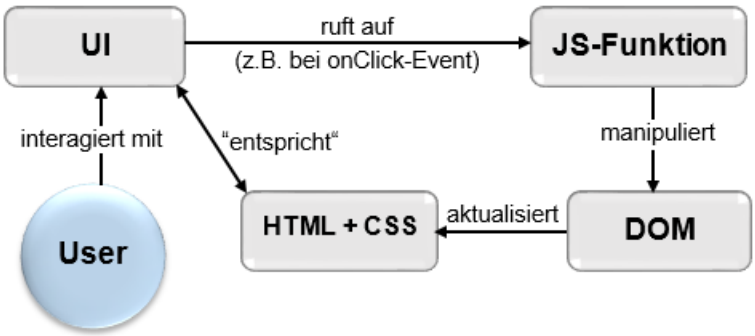

Abbildung 2: Modell Front-End

Die Frontend-Entwicklung kann somit als Praxis der Erstellung funktionaler Benutzeroberflächen, über welche Nutzer mit einer WA interagieren können, mittels HTML, CSS und JS (sowie darauf aufbauender Webtechnologien) betrachtet werden

2.2 Backend & Interaktion mit dem Frontend

Im Gegensatz zum Frontend bezeichnet das Backend (ebenso: Back-End, Back End) ganz allgemein denjenigen Bereich einer Software oder WA, der sich näher am System befindet und für den Nutzer selbst nicht sichtbar ist. Bei einer Client-Server-Anwendung bezeichnet das Backend den Server bzw. den Teil der Anwendung, der serverseitig ausgeführt wird.

Ein typisches Beispiel für einen Backend-Prozess sind Datenbank-Abfragen. Das Backend arbeitet dabei eng mit dem Frontend zusammen, ohne, dass der Nutzer etwas davon mitbekommt. Will ein Nutzer sich z.B. bei einer WA mit seinem Nutzerkonto anmelden, werden die eingegebenen Daten (z.B. Nutzername und Passwort) zunächst an den Server geschickt, wo die Validierung erfolgt. Dabei wird überprüft, ob die Anmeldedaten mit entsprechenden Einträgen in der Datenbank übereinstimmen. Dieser Vorgang spielt sich komplett im Backend ab. Das Ergebnis wird anschließend zurück an den Client (das Frontend) geschickt, wo je nach zurückgeliefertem Ergebnis entschieden wird, ob der Nutzer z.B. zur nächsten Seite weitergeleitet wird (bei erfolgreicher Anmeldung) oder eine Fehlermeldung angezeigt bekommt („Anmeldedaten falsch“).

Die Kommunikation zwischen Client und Server erfolgt dabei über das *Hypertext Transfer Protocol* (*HTTP*): Der Client sendet HTTP-*Requests* (Anfragen) an den Server, dieser verarbeitet die Anfragen (in einer bestimmten Programmier- bzw. Skriptsprache) und sendet als Antwort eine HTTP-*Response* in einem für den Client verständlichen Format zurück, z.B. HTML oder *JavaScript Object Notation* (*JSON*)[3]. Abbildung 3 veranschaulicht den kompletten zuvor beschriebenen Prozess, inklusive der Vorgänge im Backend, am Beispiel einer PHP-Ressource[4]. Innerhalb des Skripts wird serverseitig eine Datenbank-Abfrage durchgeführt; anschließend wandelt der PHP-Interpreter den PHP-Code in HTML um, das zurück an den Client gesendet wird.

[3] JSON: Ein Format zur Speicherung und zum Austausch von Daten, vergleichbar mit *XML*

[4] PHP: Akronym für *PHP Hypertext Preprocessor*, populäre Backend-Skriptsprache

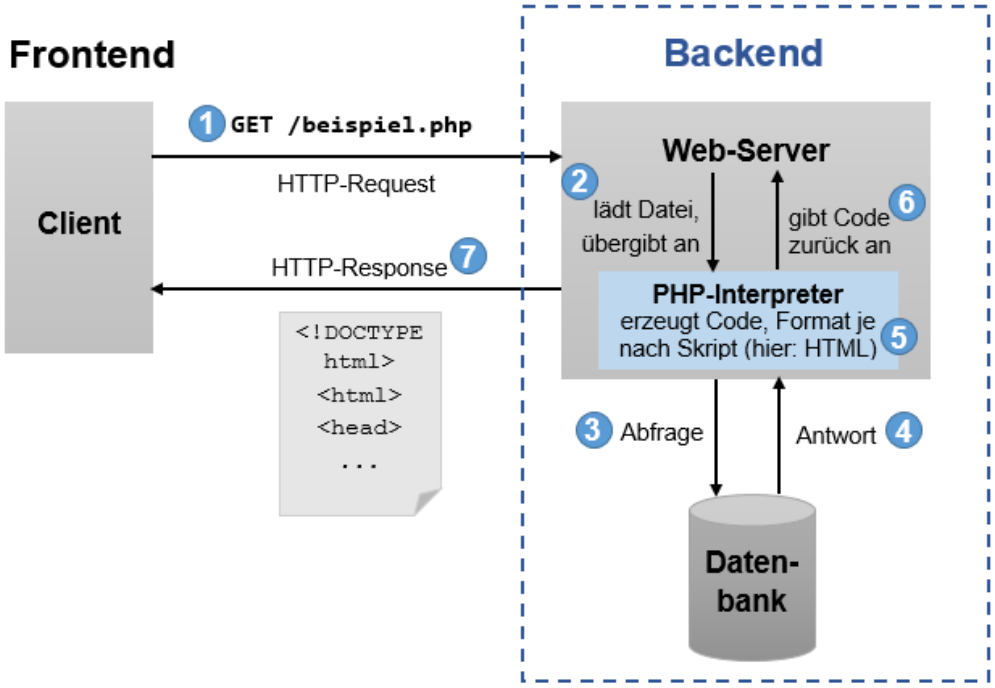

Abbildung 3: Zusammenspiel von Frontend und Backend
(Eigene Darstellung in Anlehnung an WIKIBOOKS (2011))

Weitere populäre Backend-Programmier- und Skriptsprachen neben PHP sind u.a. *Java*, *C#*, *Ruby*, *Python*, *Perl* und seit einigen Jahren auch JavaScript (durch die Entwicklung von Node.js, siehe Kapitel 2.3.3).

2.3 Grundlegende Webtechnologien dieser Arbeit

In diesem Kapitel sollen die wichtigsten Webtechnologien, auf denen die Second Screen App, basiert, erläutert werden. Da der Schwerpunkt dieser Arbeit auf dem Testen von React, *Redux* und JS (nach dem ECMAScript2015-Standard) liegt, werden diese Technologien ausführlicher beschrieben. Unterstützende Webtechnologien wie Babel oder Webpack werden hingegen nur kurz dargelegt, da ein tieferes Verständnis für diese Arbeit nicht zwingend erforderlich ist.

2.3.1 ECMAScript 2015

ECMAScript 2015 (*ES2015*) bezeichnet die 2015 erschienene Version von ECMAScript - der standardisierten Skriptsprache der Normierungsorganisation *European Computer Manufacturers Association* (*ECMA*), dessen bekannteste Implementierung JavaScript ist (vgl. NEUMANN, 2016). Ursprünglich als ECMAScript 6 veröffentlicht, spezifiziert ES2015 das

größte Update in der Geschichte von JavaScript, was bereits an der knapp 550 Seiten umfassenden Dokumentation ersichtlich wird.[5] Alle großen Browser (Edge, Firefox, Chrome, Safari) unterstützen in der aktuellsten Version ES2015-Syntax nahezu vollständig (>95%), wobei je nach Browser einige wenige Funktionen nur teilweise unterstützt werden (Stand: Juni 2017).[6] Auf Grund der hohen Browser-Kompatibilität kann ES2015 als der neue JS-Standard bezeichnet werden, womit es den Vorgänger ES 5.1 (veröffentlich im Juni 2011) ablöst. Es ist jedoch zu beachten, dass die Kompatibilität bei älteren Browser-Versionen geringer ausfällt. Ein Transpiler kann in diesem Fall Abhilfe schaffen (siehe Kapitel 2.3.5). Im Juni 2016 wurde zudem mit ES2016 bereits die nächste ECMAScript-Version veröffentlicht. Da die Neuerungen bei dieser Version jedoch verhältnismäßig gering ausfallen und zudem die Browser-Unterstützung noch sehr schwankt, soll in diesem Kapitel lediglich ES2015 behandelt werden. Zu den wichtigsten Neuerungen, die mit ES2015 Einzug in JS erhalten haben – sofern sie für diese Arbeit von Relevanz sind –, gehören:

- **Einführung von Klassen:** Erlaubt objektorientierte Programmierung auf Basis von Prototypen in JS. Man spricht daher auch von prototypenbasierte Programmierung, da keine „echten" Klassen verwendet werden, sondern lediglich Pseudoklassen (vgl. ACKERMANN, 2016: 691).

- **Neues Schlüsselwort `let`:** „Deklaration von Variablen im Gültigkeitsbereich des jeweils lokalen Blocks" (ACKERMANN, 2016: 757)

- **Arrow-Funktionen:** Neue, verkürzte Syntax zur Definition von Klassen, mit der Besonderheit, dass diese sich „auf den Kontext beziehen, in dem sie definiert wurden, nicht zwangsweise auf den Kontext, in dem sie ausgeführt werden" (ACKERMANN, 2016: 758)

- **Schlüsselwort `import`:** Anweisungen zum einfachen Einbinden von Funktionen oder Objekten aus externen Modulen oder Dateien in eine Datei (muss zuvor exportiert worden sein)

- **Schlüsselwort `export`:** Anweisung zum einfachen Exportieren von Funktionen oder Objekten aus einer Datei bzw. einem Modul (kann anschließend in eine andere Datei importiert werden)

[5] Siehe https://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf [Stand 17.06.2017]

[6] Siehe https://kangax.github.io/compat-table/es6/ [Stand 17.06.2017] (tabellarische Übersicht zur Browserunterstützung von ES2015)

Die genaue Syntax und Funktionsweise der einzelnen Neuerungen soll an dieser Stelle nicht beschrieben werden. Für detaillierte Erklärungen zu den oben genannten und weiteren ES2015-Neuerungen sei auf die JavaScript-Referenz von MDN[7], sowie das Werk von ACKERMANN (2016) verwiesen.

2.3.2 Node.js & Node Package Manager

Node.js ist eine Laufzeitumgebung, die es ermöglicht, JS außerhalb des Browsers, also serverseitig, auszuführen. Es basiert auf der V8-Engine von *Chrome*. Für Node.js stehen eine Vielzahl an Modulen, genannt Packages (deutsch: Pakete) zur Verfügung, darunter auch populäre Frontend-Frameworks und -Bibliotheken wie React, *JQuery* oder *Angular*. Zur Distribution und Installation dieser verfügt Node.js über eine eigene Paketverwaltung, den *Node Package Manager* (*NPM*). Mit rund 475.000 frei verfügbaren Packages ist NPM der größte Paketmanager der Welt (vgl. NPM, 2017).

Packages können mittels NPM einfach über das Terminal mit der Anweisung `npm install <package>` installiert werden, was einer lokalen Installation entspricht.[8] Soll beispielsweise React einem Projekt hinzugefügt werden, kann es von dessen Rootverzeichnis aus mit dem Befehl `npm install react` installiert werden. Das Package kann anschließend in jede JS-Datei importiert und somit genutzt werden.

Jedes Package verfügt über eine Konfigurationsdatei mit dem Namen `package.json`, welche sich im Rootverzeichnis des Packages befindet. Diese *JSON*-Datei beinhaltet sämtliche Informationen zu dem jeweiligen Package, darunter auch Abhängigkeiten zu anderen Packages, mindestens jedoch den Paketnamen und die Versionsnummer. Die Definition der Abhängigkeiten erfolgt über die Eigenschaften `dependencies` und `devDependencies`. Während über ersteres allgemeine Abhängigkeiten zu Paketen angegeben werden können, die das eigene Package zum Laufen benötigt, werden unter den `devDependencies` solche hinterlegt, die lediglich im Entwicklungsmodus benötigt werden, für das Funktionieren der Anwendung jedoch nicht erforderlich sind.

Während sich bei fremden Packages nicht um die `package.json` gekümmert werden muss, spielt sie vor allem bei eigenen Packages eine wichtige Rolle. Ein Package kann jedes private Node.js-Projekt mit einer validen `package.json`-Datei sein. Wird ein fremdes Paket mit dem Befehl `npm install <package> --save` installiert, wird dieses zusätzlich zur Installation auch in der Konfigurationsdatei des eigenen Packages als Abhängigkeit hinzugefügt (unter `dependencies`). Die alternative Flag `--save-dev` bewirkt, dass das entsprechende Paket stattdessen als Abhängigkeit in den `devDependencies` eingetragen wird. Der

[7] Siehe https://developer.mozilla.org/de/docs/Web/JavaScript/Reference [Stand 06.07.2017]

[8] Die Packages werden in einem Ordner „node_modules“ im aktuellen Verzeichnis installiert

Befehl `npm install`, ausgeführt ohne sonstige Parameter im Root-Verzeichnis eines Packages, installiert alle benötigten Abhängigkeiten, welche aus der `package.json` entnommen werden.

Der folgende Code (Listing 1) zeigt ein Beispiel für eine einfache `package.json`-Datei:

```
{
  "name": "Beispiel-Package",
  "version": "1.0.0",
  "description": "Ein Beispiel-Package",
  "scripts": {
    "start": "node index.js",
  },
  "dependencies": {
    "babel-polyfill": "^6.9.1",
    "react": "^15.2.1",
    "react-dom": "^15.2.1",
    "redux": "^3.5.2"
  },
  "devDependencies": {
    "webpack": "^1.13.1"
  }
}
```

Listing 1: Beispiel für eine package.json

Die Ausführung des Befehls `npm install` würde in dem Fall über NPM die Packages Babel-Polyfill, React, React-Dom und Redux als Dependency -, sowie Webpack als Dev-Dependency installieren. Der Wert hinter dem Doppelpunkt gibt jeweils die zu installierende Version an. Zudem ließe sich die Anwendung mit dem Befehl `npm start` ausführen, was das Skript `index.js` ausführen würde. Innerhalb des scripts-Objekts können zusätzlich auch noch weitere Befehle definiert werden.

2.3.3 React

React ist eine JavaScript-Bibliothek zur Erstellung von Benutzeroberflächen, die von Facebook entwickelt wurde und erstmals 2013 vorgestellt wurde. Seitdem wurde React als Open-Source-Projekt in enger Zusammenarbeit mit der Community stetig weiterentwickelt

und ist aktuell in der Version 15.6.1 (Stand: 02.07.2017) verfügbar.[9] React selbst fungiert lediglich als Darstellungsschicht einer Anwendung. Da die Bibliothek in JavaScript geschrieben ist und intern in eben jene Programmiersprache kompiliert wird, muss JavaScript auch für die Implementation der Logik von React-Anwendungen verwendet werden, wobei hierfür aber auch auf andere JS-Frameworks oder -Bibliotheken zurückgegriffen werden kann (vgl. ZEIGERMANN ET AL., 2016: 3). Es sei an dieser Stelle angemerkt, dass die folgenden Ausführungen nur einen groben Überblick über die fundamentalen Techniken und Konzepte von React geben, welche für ein grundlegendes Verständnis dieser Arbeit essenziell sind. Weitere wichtige Bestandteile von React, wie die Lifecycle-Methoden, werden in diesem Kapitel nicht behandelt, da dies den Umfang übersteigen würde. Für ein tieferes, weiterführendes Verständnis dieser Webtechnologie sei die offizielle React-Dokumentation[10] von Facebook empfohlen, sowie das Buch von ZEIGERMANN ET AL (2016).

Das Kernstück von React sind sogenannte *Komponenten*. Diese erlauben es, UIs in autarke, wiederverwendbare Teile aufzuteilen. Komponenten kapseln dabei Struktur (HTML), Aussehen (CSS) und Logik (JS). Eine React-Komponente kann auf verschiedene Weisen definiert werden. Im Folgenden wird die u.a. von FACEBOOK (2017a) empfohlene Variante mit ES2015-Klassensyntax verwendet, bei der eine Komponente als abgeleitete Klasse der abstrakten Klasse `React.Component`[11] definiert wird. Jede Komponente verfügt über mindestens eine spezielle Funktion: Die `render()`-Funktion. Innerhalb dieser wird die Struktur der Komponente mit HTML-Elementen festgelegt und mittels `return`-Anweisung zurückgeliefert. Das Vermischen von JavaScript mit HTML-Syntax wird dabei durch eine Technologie namens JSX ermöglicht, welche im Kapitel 2.3.5 noch erläutert wird.

Eine ganz schlichte Button-Komponente, die lediglich einen HTML-Button mit der Aufschrift „Hallo Welt!“ rendert, lässt sich beispielsweise in React mit folgendem Code (Listing 2) realisieren:

[9] Siehe https://github.com/facebook/react/releases [02.07.2017]

[10] Siehe https://facebook.github.io/react/docs/hello-world.html [Stand 18.06.2017]

[11] React muss zuvor per NPM installiert werden (siehe 2.3.2) und das Package `React` mit der Klasse `Component` importiert werden

```
// Button.js
import React from 'react';

class Button extends React.Component {
  render(){
    return <button type="button">Hallo Welt!</button>;
  }
}
```

Listing 2: Eine einfache React-Komponente

Über den Klassennamen, welcher der Bezeichnung der Komponente entspricht, kann die diese anschließend erzeugt werden. Das geschieht mit folgender allgemeinen, an HTML/XML angelehnten Syntax: `<ComponentName> Inhalt </ComponentName>` bzw. alternativ die verkürzte Schreibweise `<ComponentName />` bei Komponenten, die keinen Inhalt haben. Da dies bei der obigen Button-Komponente der Fall ist, würde hier der Aufruf von `<Button />` genügen, um die Komponente zu instanziieren.

Beim Erzeugen der Komponente wird die interne `render()`-Funktion aufgerufen, welche einen Knoten im DOM des Browsers erzeugt. Dieser repräsentiert das HTML-Element, das in der `return`-Anweisung zurückgegeben wird, in dem Fall `button`. Eine React-Komponente kann auch aus mehreren HTML-Elementen zusammengesetzt sein. In dem Fall ist es jedoch erforderlich, dass diese in ein umschließendes `div` eingebettet werden, da `return` immer genau ein Element zurückgeben muss.

Ein mit React erstelltes UI entsteht durch die Schachtelung mehrerer Komponenten ineinander. Der Aufruf bzw. die Instanziierung einer Komponente erfolgt innerhalb der sogenannten Eltern-Komponente (und deren `render()`-Funktion), welche wiederum eine Kind-Komponente einer anderen Komponente sein kann, usw. An oberster Stelle der daraus entstehenden Baumstruktur steht immer eine einzelne Wurzel-Komponente, häufig mit der Bezeichnung `App`, welche selbst keine Eltern-Komponente besitzt. Diese wird in ein spezifiziertes Element einer – für gewöhnlich leeren – HTML-Seite gerendert, wobei man hierfür klassischerweise ein `div`-Element wählt. React-Anwendungen fallen somit unter die Kategorie der Single Page Applications (SPAs)[12]. Abbildung 4 verdeutlicht die React-Komponentenarchitektur an einem beispielhaften UI mit 2 Dialogen (entspricht den Komponenten `HomePage` und `EmployeePage`). Die einzelnen Komponenten sind dabei farblich gekennzeichnet.

[12] SPA: Eine Web-Architektur, bei der es meist nur eine HTML-Datei gibt und alle benötigten Ressourcen entweder sofort geladen oder zur Laufzeit dynamisch nachgeladen werden (über AJAX)

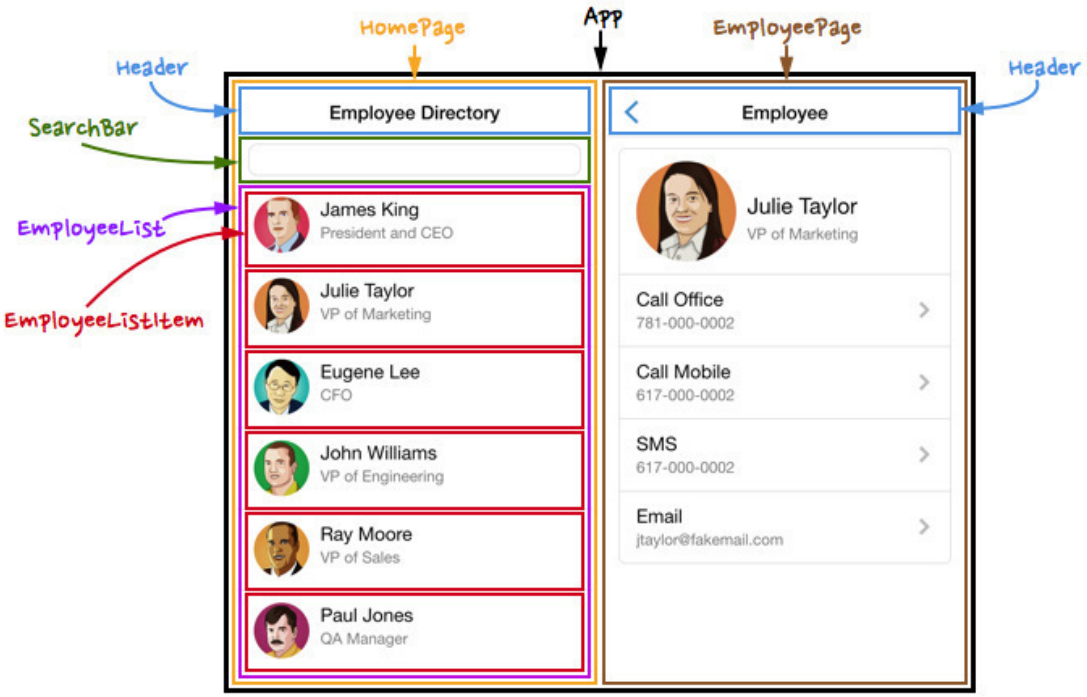

Abbildung 4: User Interface aus React-Komponenten
(Quelle: COENRAETS (2014))

Damit React-Komponenten nicht komplett statisch (wie die Button-Komponente, die immer einen Button mit derselben Aufschrift erzeugt) und somit wiederverwendbar sind, gibt es bei React sogenannte *Props* (kurz für *Properties,* dt.: Eigenschaften). Props sind Daten, im einfachsten Fall ein String oder ein numerischer Wert, die eine Komponente bei ihrer Instanziierung von ihrer Eltern-Komponente übergeben bekommt. Daten fließen bei React-Anwendungen somit immer nur in eine Richtung, was als unidirektionaler Datenfluss bezeichnet wird.

Bei dem UI aus Abbildung 4 gibt es z.B. eine Komponente `EmployeeListItem` (siehe rote Umrahmung). Innerhalb der Komponente ist die Struktur und das Aussehen definiert: Links ist ein kreisförmiges Bild fester Größe, rechts daneben ein Text in schwarzer Schrift, darunter ein weiterer Text in grauer, etwas kleinerer Schrift. Die tatsächlichen Daten – den Pfad zu der Bilddatei, den Name des Mitarbeiters und dessen Jobbezeichnung – bekommt die Komponente jedoch erst bei der Instanziierung als Prop übergeben. Auf diese Weise ist ein und dieselbe Komponente mehrfach nutzbar, da sie so allgemein wie möglich geschrieben ist.

Der Zugriff auf ein bestimmtes Prop erfolgt mit der Syntax `this.props`, gefolgt von einem Punkt und dem Namen des speziellen Props[13], und umgeben von geschweiften Klammern[14]. Das Prop ist dabei an der Stelle im Code einzufügen, an der später die Daten eingefügt werden sollen. Die `EmployeeListItem`-Komponente könnte z.B. wie folgt aufgebaut sein (Listing 3):

```
// EmployeeListItem.js
import * as style from '../style/EmployeeListItem.scss';
import React from 'react';

class EmployeeListItem extends React.Component {
  render(){
    return(
      <div className={style.container}>
        <img className={style.image} src={this.props.image} />
        <span className={style.name}> {this.props.name} </span>
        <span className={style.job}> {this.props.job} </span>
      </div>
    );
  }
}
```

Listing 3: Beispiel-Implementierung für die EmployeeListItem-Komponente

Das Styling erfolgt hierbei beispielhaft über eine importierte Sass-Datei[15] (`EmployeeListItem.scss`).

Beim Erzeugen einer `EmployeeListItem`-Komponente werden die Props in derselben Weise übergeben, wie Attribute in HTML/XML (der Attributname entspricht dabei der Bezeichnung des Props, der Wert entspricht den zu übergebenden Daten für dieses Prop).

Listing 4 zeigt, wie die zuvor geschriebene Komponente für den Mitarbeiter James King erzeugt werden könnte.

[13] Die Benennung ist mit Ausnahme einiger reservierter Schlüsselwörter frei wählbar

[14] Die geschweiften Klammern teilen dem Compiler mit, dass es sich bei dem eingeschlossenen Inhalt um reines JavaScript und nicht JSX handelt (siehe Kapitel 2.3.5).

[15] Sass: Ein Pre-Compiler für CSS

```
<EmployeeListItem image="/images/King.jpg"
name="James King" job="President and CEO" />
```

Listing 4: Erzeugen einer React-Komponente mit Props

Props werden verwendet, um das Aussehen oder Verhalten einer Komponente „von außen“ zu verändern, können jedoch innerhalb der Komponente nicht modifiziert werden. Aus diesem Grund gibt es bei React neben Props den sogenannten *State* (dt.: Zustand). Der State ist ein einfaches JS-Objekt, das den Zustand der Komponente zu einem bestimmten Zeitpunkt repräsentiert. Er beinhaltet Daten, die zum Beschreiben des Aussehens einer Komponente wichtig sind, ist jedoch veränderbar.

Der Zugriff auf eine State-Eigenschaft erfolgt analog zum Zugriff auf ein Prop mit `this.state.` gefolgt vom Namen der State-Eigenschaft. Die Änderung des States erfolgt mit der `setState()`-Funktion, welche als Argument ein JS-Objekt mit der zu ändernden Eigenschaft und dem neuen Wert übergeben bekommt. Dies geschieht meist bei einer Nutzerinteraktion: Ein Nutzer klickt z.B. auf einen Button, wodurch eine Funktion aufgerufen wird, die dessen State aktualisiert. Die Besonderheit ist dabei, dass die Komponente automatisch neu gerendert wird, wenn sich der State ändert und sich dies auf das Erscheinungsbild der Komponente auswirkt. Das Aussehen der Komponente und ihr State sind somit immer im Einklang, ohne, dass sich der Entwickler selbst darum kümmern muss.

Eine Besonderheit von React ist dabei, dass es ein sogenanntes virtuelles DOM (engl.: Virtual DOM, kurz: VDOM) verwendet. Hierbei handelt es sich um eine abstrakte Kopie des tatsächlichen Browser-DOMs in Form eines JavaScript-Objekts. Es wird für sämtliche Neuberechnungen verwendet, die sonst im DOM des Browsers durchgeführt werden würden, was auf Grund der höheren Komplexität deutlich länger dauert (vgl. CODEACADEMY, o.J. [ca. 2016]). Nur dann, wenn sich das VDOM geändert hat, wird auch das DOM des Browsers aktualisiert, was wiederum den Render-Vorgang anstößt. Die Auslagerung der Berechnungen auf das VDOM macht React-Anwendungen besonders schnell (vgl. LONG, 2016), was nur einer von vielen Vorteilen bei der Verwendung von React für das Frontend ist und die JS-Bibliothek vor allem für komplexe, dynamische Web-Anwendungen (mit vielen DOM-Berechnungen) besonders beliebt macht.

Weitere Vorteile von React sind laut ZEIGERMANN ET AL. (2016: 7) u.a. die einfache Erlernbarkeit, die problemlose Kombination mit anderen Frontend-Webtechnologien (React fungiert lediglich als Darstellungsschicht), sowie die große Anzahl an Frameworks und Tools für React, welche das Entwickeln deutlich vereinfachen.

2.3.4 Redux

Redux ist eine 2015 von Dan Abramov entwickelte Bibliothek für Front-End-Anwendungen, welche als Architekturmuster zur einfachen Verwaltung des States einer Applikation zum Einsatz kommt. Mit *State* sind dabei sämtliche Daten gemeint, die für die Applikation als

Ganzes oder zumindest mehr als eine Komponente wichtig sind, z.B. der aktuell aktive View oder Nutzereingaben (vgl. REDUX, 2016). Obwohl Redux mit nahezu jedem beliebigem JS-Framework (oder einfach schlichtem JS) verwendet werden kann, wird es am häufigsten in Kombination mit React gebraucht (vgl. BACHUK, 2016).

Im Gegensatz zu den klassischen Architekturmustern wie dem *Model-View-Control*-Muster (*MVC*), bei denen der Entwickler oft nur schwer nachvollziehen bzw. vorhersagen kann, wann und wieso sich der State (das Model) aktualisiert, macht Redux es sich zum Ziel, Änderungen des States vorhersehbar zu machen (vgl. REDUX, 2017).

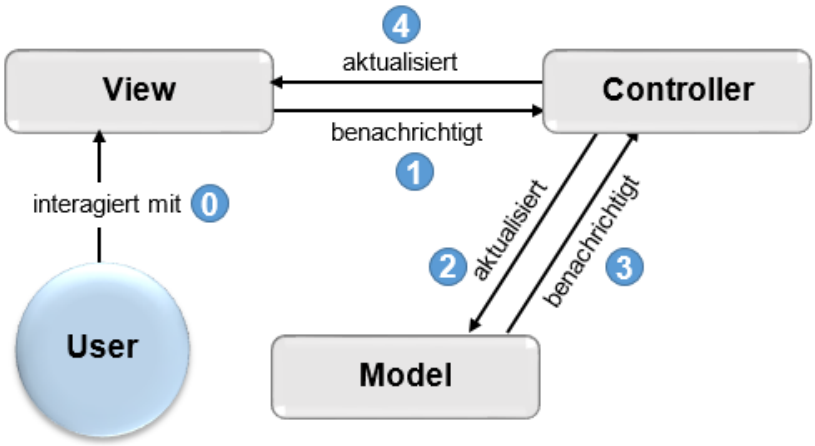

Abbildung 5: MVC-Entwurfsmuster mit bidirektionalem Datenfluss

Die meisten Implementierungen des MVC-Musters[16], wie etwa in Abbildung 5 dargestellt, setzen auf einen bidirektionalen Datenfluss. Der Nutzer interagiert über das UI (den View) mit der Anwendung, der View benachrichtigt den Controller, der daraufhin das Model aktualisiert. Anschließend fließen die Daten in umgekehrte Richtung wieder zurück: Das Model benachrichtigt den Controller über die Änderung, woraufhin dieser den View entsprechend der neuen Daten aus dem Model aktualisiert. Die Schwierigkeit liegt insbesondere im Debugging solcher Anwendungen (vgl. STARKE, 2015: 125). Bei mehreren Models, Views und/oder Controllern in einer Anwendung steigt die Komplexität zusätzlich: Aktualisiert sich das UI beispielsweise an einer falschen Stelle, kann es sehr mühselig sein, den Fehler im Code zu finden. Redux setzt daher auf einen strikt unidirektionalen Datenfluss, wie er auch von React verwendet wird (siehe Kapitel 2.3.3). Dies ist einer der Gründe, weshalb Redux eine beliebte Ergänzung für React-Anwendungen darstellt. Abbildung 6 zeigt modellhaft den Datenfluss bei Redux.

[16] Es gibt zahlreiche Variationen des MVC-Musters, die sich im Wesentlichen darin unterscheiden, welche Teile (Model, View, Controller) miteinander auf welche Weise miteinander kommunizieren.

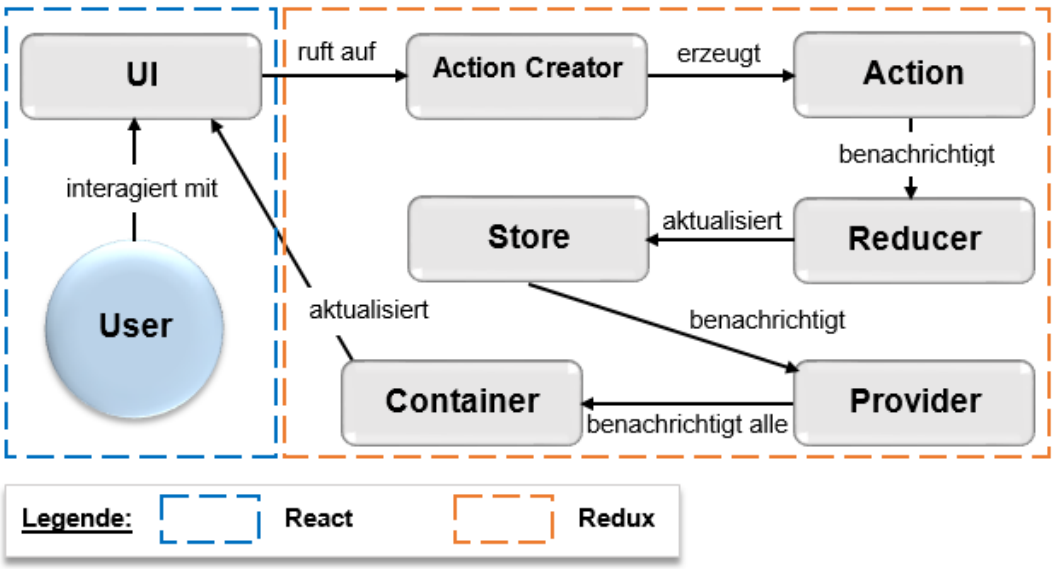

Abbildung 6: Unidirektionaler Datenfluss bei Redux

Die Kernidee von Redux ist, dass der gesamte (globale) State der Anwendung in einem einzigen JS-Objekt, dem sogenannten **Store**, gespeichert wird.[17] Wird der Store aktualisiert, d.h. ändert sich der Zustand der Applikation, resultiert dies in der Regel in einer Aktualisierung des UI.

Der globale State kann nur durch eine sogenannte **Action** geändert werden.[18] Eine Action ist einfaches JS-Objekt, welches eine gewünschte Änderung des States beschreibt. Es verfügt mindestens über eine Eigenschaft `type`, sowie weitere optionale, frei benennbare Eigenschaften (den sogenannten Payload, dt.: Nutzdaten), deren Werte als Daten an den Store gesendet werden sollen.

Beispiel für eine Action vom Typ `DO_SOMETHING` (der Typ wird laut Konvention in Uppercase-Schreibweise benannt) und einer zusätzlichen Eigenschaft `payload`:

```
{ type: "DO_SOMETHING", payload: "Aktualisiere Store" }
```

Listing 5: Beispiel für eine einfache Action unter Redux

Action Creator können bei der Erstellung von Actions helfen, wenngleich deren Gebrauch nicht zwingend ist. Hierbei handelt es sich um Funktionen, die eine Action zurückliefern.

---

[17] 1. Redux-Prinzip: „Single source of truth – The state of your application is stored in an object tree within a single store." (siehe http://redux.js.org/docs/introduction/ThreePrinciples.html [Stand 28.07.2017]

[18] 2. Redux-Prinzip: „State is read-only – The only way to change the state is to emit an action […]" (siehe http://redux.js.org/docs/introduction/ThreePrinciples.html [Stand 28.07.2017])

Führt der Nutzer eine Aktion durch, die eine Aktualisierung des globalen States zur Folge hat, etwa durch Klicken eines Buttons, wird zunächst der entsprechende Action Creator aufgerufen. Optionale Eigenschaften einer Action (wie `payload` bei Listing 5) werden dabei beim Aufruf der Funktion als Argument übergeben. Der Action Creator generiert anschließend die spezifische Action, welche automatisch an den Reducer gesendet[19] wird.

Während Actions beschreiben, mit welchen Daten der Store aktualisiert werden soll, ist es die Aufgabe der **Reducer** zu beschreiben, wie genau der Store modifiziert werden soll.[20] Ein Reducer ist eine reine Funktion[21], welche als Argument den aktuellen State und eine Action übergeben bekommt und als Ergebnis den neuen State zurückliefert. An Hand der Eigenschaft `type` entscheidet die Reducer-Funktion, wie der State für den jeweiligen Action-Typ zu modifizieren ist. Dabei ist die Idee, dass es nicht zwangsläufig nur einen einzigen Reducer gibt, der den gesamten globalen State verwaltet, sondern mehrere Reducer, die jeweils für einen unabhängigen, logischen Teil des States verantwortlich sind und auch nur diesen modifizieren.

Über eine spezielle `combineReducers()`-Funktion, welche alle Reducer der Anwendung als Argument übergeben bekommt, werden die die einzelnen „Teil-Zustände“ der einzelnen Reducer zusammengeführt. Dieser sogenannte Root-Reducer beinhaltet somit den globalen State der Applikation, woraus wiederum der Store erzeugt wird. Dies geschieht mit Hilfe der `createStore()`-Funktion. Eine vereinfachte Darstellung der Beziehung zwischen Reducer und Store findet sich in den Anlagen (Abbildung 18).

Sobald ein Reducer den (Teil-)State, für den er verantwortlich ist, modifiziert, wird die entsprechende Änderung automatisch auch im Store vorgenommen. Damit innerhalb der einzelnen React-Komponenten einer Anwendung auf den globalen State zugegriffen werden kann, muss der Store der Applikation zur Verfügung gestellt werden. Dies übernimmt der sogenannte **Provider**, der als eine Art Bindeglied zwischen dem Redux-Store und React angesehen werden kann. Komponenten, die auf den Applikations-State zugreifen können, bezeichnet man als **Container** bzw. Container-Komponenten. Hierbei handelt es sich um gewöhnliche React-Komponenten, mit der Besonderheit, dass diese über Änderungen des Stores informiert werden. Dies ist möglich, da ihnen der globale State (oder ein logischer Teil davon) als Prop zur Verfügung steht. Ändert sich der globale State, wird diese Information über den Provider an alle Container-Komponenten weitergereicht. Wird z.B. der Wert einer einzelnen Eigenschaft verändert, werden automatisch (zur Laufzeit) alle Container-

[19] Dies bezeichnet man als Dispatch (dt.: Versand).

[20] 3. Redux-Prinzip: „Changes are made with pure functions – To specify how the state tree is transformed by actions, you write pure reducers.” (siehe http://redux.js.org/docs/introduction/ThreePrinciples.html [Stand 28.07.2017])

[21] Reine Funktionen sind Funktionen, die bei gleichen Argumenten immer dasselbe Ergebnis berechnen und den State nicht direkt verändern.

Komponenten neu gerendert, die auf diese Eigenschaft als Prop zugreifen (Aktualisierung des UI).

Darüber hinaus können Container-Komponenten nicht nur Zugriff auf Teile des globalen States haben, sondern auch auf ausgewählte Action Creator, die ebenfalls als Props zur Verfügung stehen. Auf diese Weise können über das UI (z.B. beim Klicken eines Buttons) Actions erzeugt werden und somit der Store aktualisiert werden.

Die Vorteile der Verwendung von Redux in Verbindung mit React, die zuvor bereits erläutert wurden, zeigen sich erst bei größeren Applikationen: Dan Abramov, der Entwickler von Redux, betont, dass das Framework den Entwicklungsprozess bei kleineren Anwendungen unter Umständen nur unnötig verkompliziere, weswegen es durchaus legitim sei, hier nur den State der React-Komponenten anstelle eines globalen States zu verwenden (vgl. ABRAMOV, 2016).

2.3.5 JSX

JSX ist eine Syntaxerweiterung für JavaScript, die vor allem in Verbindung mit React zum Einsatz kommt.[22] JSX erlaubt es, JavaScript mit HTML-artiger Syntax zu vermischen. Obwohl React auch ohne JSX auskommt, wird die Nutzung empfohlen, da es vor allem die Lesbarkeit des Codes erhöht. Ohne JSX-Syntax könnte die `Button`-Komponente aus 2.3.1 auch mit einfachem JS mit der `createElement`-Methode definiert werden, wie es in Listing 6 gezeigt ist.

```
// innerhalb der render()-Funktion von Button.js – ohne JSX
return (
  React.createElement(
    "button",
    {type: "button", onClick: this.incrementCounter.bind(this)},
    this.state.counter
  );
);
```

Listing 6: Implementierung einer React-Komponente ohne JSX

Diese Variante macht es jedoch ungemein schwerer, nachzuvollziehen, was eigentlich dargestellt werden soll. Insbesondere dann, wenn eine Komponente aus mehreren Elementen und Kind-Komponenten besteht, geht die Übersichtlichkeit schnell verloren. Noch dazu schleichen sich bei der komplizierten Syntax schnell Fehler im Code ein. Mit JSX werden

[22] Siehe https://facebook.github.io/react/docs/introducing-jsx.html [21.05.2017]

Elemente einfach so definiert, wie es in einem HTML-Dokument getan wird. Die Technologie muss jedoch immer in Verbindung mit einem Transpiler (siehe Kapitel *2.3.6*) verwendet werden, welcher die JSX-Fragmente herausfiltert und in konformes JS übersetzt.

Der folgende Code (Listing 7) mit JSX-Syntax wird somit in den JS-Code aus Listing 6 kompiliert:

```
// innerhalb der render()-Funktion von Button.js - mit JSX
return (
  <button type="button" onClick={this.incrementCounter.bind(this)}>
    {this.state.counter}
  </button>
);
```

Listing 7: Implementierung einer React-Komponente mit JSX

Soll innerhalb eines JSX-Codes gewöhnliches JS verwendet werden, ist der entsprechende JS-Teil in geschweifte Klammern zu setzen, wie es zum Beispiel bei `{this.state.counter}` in dem Beispielcode getan wird. Der verwendete Transpiler weiß somit, dass es sich dabei nicht um JSX sondern normales JS handelt und somit keine Übersetzung stattfinden muss.

2.3.6 Transpiler - Babel

Transpiler (auch *Transcompiler* oder *Source-To-Source-Compiler*) sind spezielle Compiler, die Quellcode von einer Programmiersprache in eine andere Sprache umwandeln, also „übersetzen“. Babel ist ein Beispiel für einen beliebten JavaScript-Transpiler, der verwendet wird, um Quellcode in valides JS der Version ES5 zu übersetzen. Am häufigsten wird Babel eingesetzt, um ES2015-Code in die alte Version der Skriptsprache zu übersetzen. Obwohl die Browser-Kompatibilität für ES2015 bei modernen Browsern inzwischen sehr hoch ist (siehe Kapitel 2.3.4), werden noch nicht alle Funktionen des neuen Standards von allen Browsern, insbesondere älteren Versionen, unterstützt. Babel erlaubt es, die ES2015-Syntax bedenkenlos zu verwenden, indem es den geschriebenen Code beim Kompilieren in ES5-Code umwandelt, der von allen gängigen Browsern verstanden wird und somit keine Kompatibilitätsprobleme verursacht. Darüber hinaus wird Babel bei React-Anwendungen auch häufig als Transpiler für JSX eingesetzt. Babel kann also nicht nur ES2015 in ES5 übersetzen, sondern auch JSX-Syntax in ES5-Code umwandeln (siehe Kapitel 2.3.5).

2.3.7 Webpack

Webpack ist ein Modul-Bundler für JS-Projekte, der hauptsächlich dafür verwendet wird, um mehrere JS-Dateien oder sonstige Dateien, z.B. CSS-Dateien, zu wenigen, gebündelten Dateien – oftmals nur einer einzigen pro Dateiformat – zu kombinieren. Webpack ermöglicht es, beispielsweise den JS-Code logisch in mehrere Dateien aufzuteilen, jedoch lediglich eine einzelne Datei zu laden, wenn die Webanwendung im Browser ausgeführt wird. Dies nimmt vor allem dem Entwickler Arbeit ab, da bei großen Projekten nicht händisch mitunter Hunderte von Skripten in das Projekt eingebunden werden müssen. Darüber hinaus reduziert es auch die Fehleranfälligkeit, die ohne einen Bundler wie Webpack nicht nur durch das Vergessen des Einbindens einzelner Skripte ansteigt, sondern auch durch die falsche Reihenfolge der Einbindung.

3 Grundlagen des Testens von Software

3.1 Notwendigkeit und Ziele von Softwaretests

„Keine Software ist fehlerfrei."

(ZUSER ET AL., 2004: 356)

Empirische Untersuchungen haben ermittelt, dass ein Programm durchschnittlich auf 17,7 Fehler pro 1000 Zeilen „effektiven" Code (d.h. ausschließlich Kommentare und Leerzeilen) kommt (vgl. WITTE, 2016: 15). Im Bereich der Softwareentwicklung ist fehlerhafte SW immer mit hohen Kosten für das Softwareunternehmen verbunden (vgl. ZUSER ET AL., 2004: 356): Zusätzliche, nicht eingeplante Arbeitszeit muss dafür aufgebracht werden, Fehler zu lokalisieren und zu beheben, das Projektbudget ist jedoch festgelegt. Die Kosten sind dabei umso höher, je früher im SW-Entwicklungsprozess ein Fehler gemacht und je später er entdeckt wird (vgl. ZUSER ET AL., 2004: 356f.). Am höchsten wären die Einbußen demnach bei einem Fehler, der bereits auf die Phase der Anforderungsanalyse[23] zurückzuführen ist, während des gesamten Entwicklungsprozesses unentdeckt blieb, und erst nach der Inbetriebnahme auf Seiten des Kunden bemerkt. Neben wirtschaftlichen Schäden können insbesondere solche Fehler, die es über den Entwicklungsprozess hinaus in die finale SW schaffen, auch Imageschäden für das Softwareunternehmen zur Folge haben, wenn der Kunde unzufrieden mit der Produktqualität ist.

Softwaretests können hierbei präventiv wirken – vorausgesetzt, dass bereits frühzeitig mit dem Testen begonnen wird – und sind deshalb im Entwicklungsprozess von enormer Wichtigkeit. Es ist jedoch eine Fehlannahme, zu glauben, dass durch Testen die fehlerfreie Funktionsweise einer Software bestätigt bzw. hergestellt werden kann: MYERS (1979: 5) definiert Testen als das „Ausführen eines Programes mit der Absicht, Fehler zu finden"[24]. SW-Tests haben also vielmehr das Ziel, möglichst viele Programmfehler zu entdecken, die anschließend behoben werden müssen, und so die erwartungsgemäße Funktionsweise der SW zu gewährleisten. Es ist allerdings mit keinen Mitteln möglich zu beweisen, dass eine SW komplett fehlerfrei ist (vgl. LIGGESMEYER, 2009: 2) und demnach auch durch Tests nicht möglich, *alle* Fehler in einem Programm aufzudecken. Obwohl eine 100-prozentige Testabdeckung

[23] Jedes größere Softwareprojekt gliedert sich grob in die Phasen Anforderungsanalyse, (System-)Entwurf, Implementierung, Test & Betrieb

[24] In Originalsprache: „Testing is the process of executing a program with the intent of finding errors."

nicht möglich ist, können ordentlich geplante und implementierte Tests dazu beitragen, die Anzahl der Fehler in einer Software auf ein Minimum zu reduzieren und somit mitunter großen Schaden verhindern.

Das Testen spielt auch eine wichtige Rolle für Continuous Inegration (CI; dt.: kontinuierliche Integration) von Software-Projekten. CI bezeichnet einen automatisierten Prozess der agilen Softwareentwicklung, bei dem der Quellcode aller Entwickler an einem SW-Projekt zu einer einzigen lauffähigen Version zusammengeführt wird (vgl. MARCENKO, 2014: 25f.). Fast alle CI-Systeme erlauben das automatisierte Durchführen von Tests. Sofern nicht alle Tests erfolgreich durchlaufen werden, wird der Integrationsvorgang abgebrochen. Bei der Arbeit im Team ist dies von großer Bedeutung, da somit sichergestellt werden kann, dass kein Fehler, der von einem Entwickler (unbemerkt) in den Code programmiert wurde, in die neue Version der entsprechenden Software gelangt – zumindest keiner, der nicht zuvor bereits durch einen Test abgedeckt wurde. MARCENKO (2014: 44) betont jedoch, dass ausreichend Tests vorhanden sein müssen, damit automatisierte Tests in Verbindung mit CI auch nützlich sind.

3.2 Grundlegende Begriffe

Testobjekt (TO): Bezeichnung für die Komponente (z.B. ein Objekt oder eine Funktion) oder das (Sub-)System, welches getestet wird

System Under Test (SUT): Bezeichnung für das System / die Anwendung, die getestet wird. Alle Testobjekte sind Teil des SUT.

Testfall (TC; für engl.: test case): Bezeichnung für ein konkretes Szenario, das getestet wird. Ein Testfall legt fest, was (welches TO) unter welchen Bedingungen (welche TU) mit welchen Eingabewerten getestet wird, und welches Verhalten (Soll-Wert) als Reaktion auf den Eingabewert erwartet wird.

Testsuite: Bezeichnung für eine Gruppe von Testfällen, die zusammengehören. Eine Testsuite kann z.B. mehrere Testfälle für ein bestimmtes Testobjekt enthalten.

Testumgebung: Bezeichnung für den Zustand aller Hard- und Software-Komponenten, unter denen die Tests durchgeführt werden. Hierzu zählen u.a. die Hardware des Computers (auf welchem der Test ausgeführt wird), das verwendete Betriebssystem und installierte Bibliotheken. Die Testumgebung ist zu unterscheiden von der **Produktionsumgebung**, also der Umgebung, in der die Software später tatsächlich läuft. Als Leitsatz gilt, dass die Testumgebung von der Produktionsumgebung isoliert sein sollte, dieser jedoch möglichst nah sein sollte, da nur so aussagekräftige Ergebnisse produziert werden können (vgl. BRANDES, o.J. [ca. 2011]).

Black-Box-Test: Ein Testverfahren, bei dem das TO von außen betrachtet wird, wohingegen die innere Struktur (der Quellcode) ignoriert wird. Eingabewerte werden an Hand der

Spezifikation (z.B. laut Pflichtenheft) getestet und überprüft, ob das gelieferte Ergebnis (Ist-Wert) dem erwarteten Soll-Wert entspricht.

White-Box-Test: Ein Testverfahren, bei dem der Quellcode des Testobjekts bekannt ist und bewusst betrachtet wird, mit dem Ziel „aufgrund der Struktur alle möglichen Ausführungspfade zu überprüfen und mögliches Fehlverhalten bei einem der Pfade zu entdecken“ (ZUSER ET AL., 2004: 368).

Mock-Objekt: Bezeichnung für eine Objekt-Attrappe, die als Ersatz für ein reales Objekt dient, welches für das Testen einer Komponente benötigt wird, da der Test ansonsten nicht durchführbar wäre. Ein Mock-Objekt kommt immer dann zum Einsatz, wenn das reale Objekt noch nicht implementiert ist oder aus irgendeinem Grund für den Test nicht verwendet werden kann oder soll.

3.3 Teststufen nach dem V-Modell

Die Entwicklung von SW erfolgt klassischerweise in mehreren Phasen. Ein etabliertes klassisches Vorgehensmodell für den SW-Entwicklungsprozess ist das allgemeine V-Modell (siehe Abbildung 7). Dieses Modell definiert zum einen die einzelnen Entwicklungsphasen, beginnend mit der Anforderungsdefinition bis hin zur Implementierung. Zum anderen gibt das V-Modell aber auch phasenweise das Vorgehen zum Testen der entwickelten Software vor, beginnend mit den kleinsten Einheiten bis hin zum gesamten System und dem Abnahmetest mit dem Kunden. Dabei steht jeder Teststufe einer bestimmten Entwicklungsstufe gegenüber, gegen welche getestet wird. Die Tests selbst haben dabei einen validierenden Charakter, d.h. sie überprüfen, ob das jeweilige Testobjekt den Anforderungen der Spezifikation auf der jeweiligen Ebene entspricht. Die einzelnen Teststufen bauen aufeinander auf, d.h. die zweite Teststufe (Integrationstest) wird erst dann durchgeführt, wenn alle gefundenen Fehler in der vorherigen Teststufe (Komponententest) beseitigt wurden, usw.

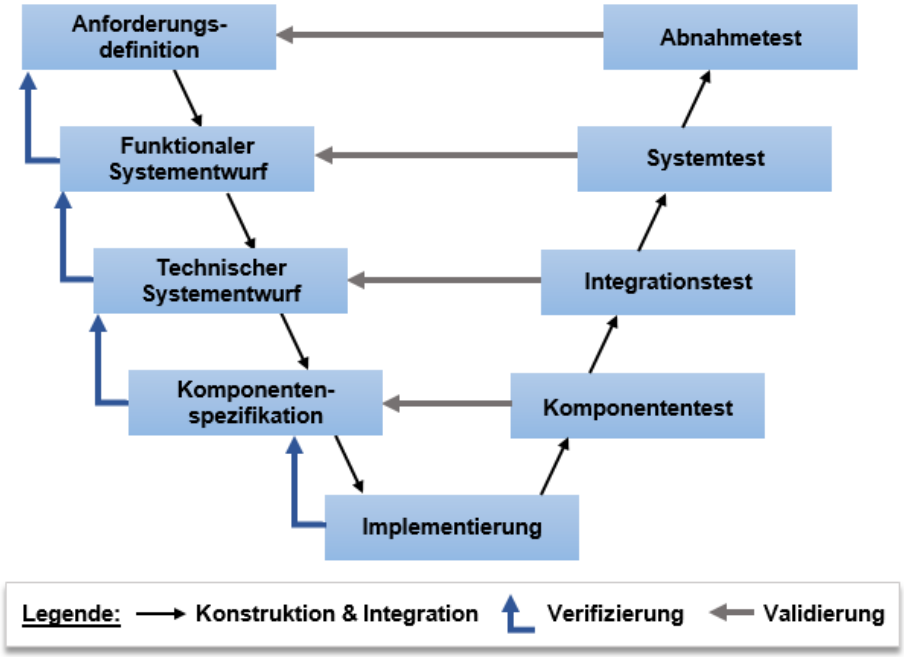

Abbildung 7: Das Allgemeine V-Modell
(Eigene Darstellung in Anlehnung an LINZ (2013))

Die einzelnen Teststufen des V-Modells sollen im Folgenden kurz erläutert werden, wobei der Schwerpunkt auf dem Komponententest liegt, der für diese Arbeit von Wichtigkeit ist. Zum Zwecke einer besseren Einordnung in den kompletten Testzyklus werden jedoch auch die weiteren Teststufen unter *3.3.2* zusammenfassend kurz beschrieben.

3.3.1 Komponententest

Der Komponententest (auch Modultest; engl.: *unit test*) ist ein Testverfahren auf der untersten Ebene (siehe V-Modell, Abbildung 7) zum Überprüfen der rechtmäßigen Funktionalität einzelner Softwarebausteine (genannt Komponenten; engl.: *units*) eines Programms. Laut SPILLNER ET AL. (2012: 44) prüft der Komponententest, „ob jeder einzelne Softwarebaustein für sich die Vorgaben seiner Spezifikation erfüllt." Die einzelnen, atomaren SW-Bestandteile werden bei diesem Testverfahren also isoliert betrachtet und auf eventuelle Fehler geprüft – Wechselwirkungen mit anderen Komponenten werden an dieser Stelle nicht betrachtet. Fällt ein SW-Baustein durch den Komponententest, ist der Fehler im SW-Design der Komponente selbst zu finden, nicht in komponentenexternen Einflüssen. Komponente ist dabei als abstrakter Begriff zu verstehen: Getestet werden können beispielsweise Klassen, Funktionen oder Methoden. Komponententests können dabei sowohl als Black-Box-, als auch als White-Box-Test durchgeführt werden.

3.3.2 Weitere Teststufen

Aufbauend auf den Komponententests folgen im nächsten Schritt Integrationstests, welche die erwartungsgemäße Kollaboration der zuvor isoliert betrachteten Komponenten miteinander prüfen. Integrationstests können somit Fehler in den Schnittstellen aufdecken.

Im anschließenden Systemtest wird das komplette System einschließlich aller Hardware-Komponenten gegen alle funktionalen und nichtfunktionalen Anforderungen getestet. Dieser wird als reiner Black-Box-Test durchgeführt.

Abschließend folgt der Abnahmetest, auch Akzeptanztest genannt, welcher stets in Beisein des Kunden durchgeführt wird und dem einzigen Zweck dient, diesen von der richtigen Funktionsweise der Software bzw. des Systems zu überzeugen (vgl. WITTE, 2016: 73).

3.4 Testen von Software und des Frontends – Der aktuelle Stand der Technik

Obwohl sich das Frontend bei einer WA aus den drei Teilen HTML, CSS und JS zusammensetzt, kann lediglich der JS-Code getestet (nach der Definition von MYERS aus 3.1) werden. Dies ist auf den einfachen Umstand zurückzuführen, dass es sich bei den beiden erstgenannten Technologien um keine Programmier- bzw. Skriptsprachen handelt. Tests werden also lediglich für den JS-Code geschrieben, unabhängig davon, ob reines JS oder ein Framework verwendet wird. Der Vollständigkeit halber sei jedoch angemerkt, dass auch das HTML und CSS auf Validität der Syntax geprüft werden sollte, wofür es spezielle Werkzeuge gibt (vgl. GRAVELLE, o.J. [ca. 2015]). Auch die Prüfung der Browserunterstützung, welche im Bereich der Frontend-Entwicklung von großer Bedeutung ist, wird in diesem Kapitel nicht behandelt, da auch dies nicht unter „Testen“ fällt. Im Folgenden sollen einige Kernkonzepte des „State of The Art“ beim Testen des Frontends bzw. allgemein von Software vorgestellt werden, und auch auf einige Besonderheiten beim Testen von React-Anwendungen eingegangen werden.

3.4.1 Test-Frameworks

Zur Automatisierung von Tests gibt es – wie auch in anderen Bereichen der Softwareentwicklung – spezielle Test-Frameworks. Solche Frameworks erlauben es, Code zum Testen des Programmcodes mit gezielten Testfällen zu schreiben, und diese Tests anschließend automatisiert durchzuführen und auszuwerten. Test-Frameworks dienen zum Durchführen von Komponententests, weswegen auch die Bezeichnung Unit-Test-Framework verwendet wird.

Klassischerweise werden die Tests in Form von Assertions (dt.: Zusicherungen) formuliert. SCHÄFER (2010: 294) definiert eine Assertion als „ein Sprachkonstrukt, das einen erwarteten Zustand einer Variablen prüft und bei Nichteinhaltung eine Exception, also eine Ausnahmebedingung signalisiert." Eine Assertion besteht in der Regel aus einem Selektor, der ein zu untersuchendes Element auswählt (im einfachsten Fall eine Variable) und einem sogenannten Matcher (von engl.: (to) match, dt.: übereinstimmen) welchem der Erwartungswert übergeben wird. Der Matcher legt fest, in welcher Relation der Erwartungswert zu dem selektierten Wert stehen soll, z.B. beide Werte sind identisch oder der Erwartungswert ist zahlenmäßig kleiner.

Nach jedem Durchlauf liefert das TF dem Tester eine Auflistung aller Testdurchläufe mit einem Hinweis, ob der jeweilige Test bestanden wurde oder nicht. Die Tests können dann wiederholt ausgeführt werden, z.B. nach dem Einbau eines neuen Features, wobei man hierbei von einem Regressionstest spricht. Es ist jedoch anzumerken, dass Test-Frameworks nur die unteren Teststufen abdecken, hauptsächlich Komponententests.

Laut einer Umfrage aus dem Jahr 2016 unter über 5000 JS-Entwicklern sind die beliebtesten und am häufigsten genutzten JS-Test-Frameworks Mocha und *Jasmine* (vgl. SHILMAN, 2016). Dabei ist jedoch zu bedenken, dass einige Test-Frameworks wiederum auf das Testen bestimmter JavaScript-Frameworks spezialisiert sind (z.B. Jest für React), und somit in der Umfrage einen niedrigeren Platz einnehmen, da sie somit nur für einen Bruchteil aller JS-Entwickler interessant sind.

3.4.2 Testgetriebene Entwicklung

Ein Trend, der sich nicht nur im Bereich der Web- bzw. Frontend-Entwicklung, sondern allgemein im Bereich der Softwaretechnik abzeichnet, ist die zunehmende Popularität von agilen Vorgehensmodellen. In diesem Zusammenhang ist beim Testen das Prinzip der testgetriebenen Entwicklung (TDD; für engl.: Test-Driven Development) von besonderer Wichtigkeit. Der Grundgedanke hinter diesem Ansatz ist, dass Komponententests bereits vor der eigentlichen Implementierung der Komponente geschrieben werden. Erst nach dem ersten Ausführen des Tests, welcher zwangsläufig beim ersten Durchführen fehlschlagen wird, wird die Komponente programmiert, „mit dem Ziel, die Anforderungen des Tests zu erfüllen, sodass dieser bei erneuter Ausführung besteht" (ACKERMANN, 2016: 944). Anschließend können noch Optimierungen an der Komponente vorgenommen werden (genannt Refactoring), wobei der zugehörige Komponententest jedes Mal aufs Neue zu bestehen ist. Der eben beschriebene TDD-Workflow ist in Abbildung 8 dargestellt.

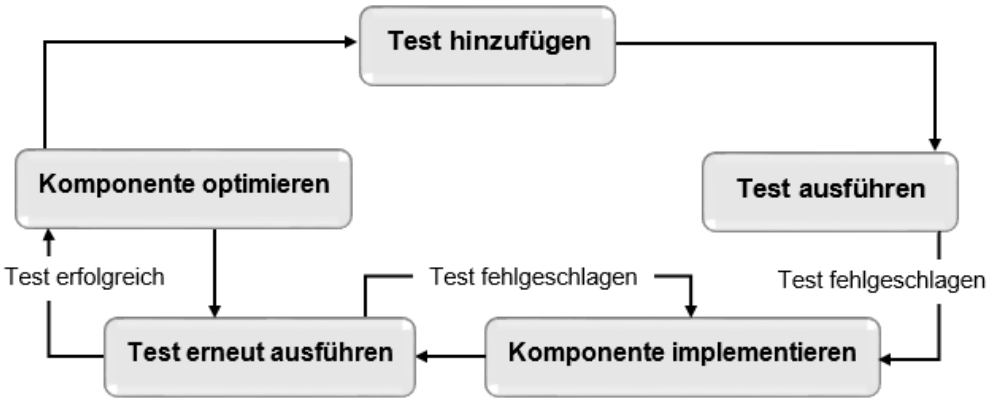

Abbildung 8: Workflow der testgetriebenen Entwicklung
(Eigene Darstellung nach ACKERMANN (2016: 945))

Die Frage nach dem Nutzen von TDD für den Entwicklungsprozess wird in der Branche kontrovers diskutiert. Diverse Studien kommen zu dem Ergebnis, dass TDD die Softwareentwicklung positiv beeinflusst. Eine Fünf-Jahres-Studie der *IBM*[25] in Zusammenarbeit mit der North Carolina State University zieht den Schluss, dass TDD zur Erhöhung der Qualität der SW bei der Entwicklung beitragen kann (vgl. SANCHEZ ET AL., 2007). Eine weitere Studie unter vier Entwicklerteams von IBM und Microsoft bestätigt dieses Ergebnis: Im konkreten Fall konnte die Fehlerrate zwischen 40% und 90% reduziert werden (in Relation zur Fehlerrate bei vergleichbaren Projekten) und selbst die Entwicklungszeit um bis zu 35% verringert werden (vgl. NAGAPPAN ET AL., 2008: 297f.).

Andere Studien hingegen – wie eine aus dem Jahr 2016 unter 21 Studenten im Aufbaustudium – kommen zu dem Ergebnis, dass TDD keinerlei messbare Vorteile im Vergleich zum klassischen „Test-Last"-Ansatz hat, weder was den Aufwand, noch die Qualität der SW oder die Produktivität der Entwickler anbelangt (vgl. FUCCI ET AL., 2016). Die gegensätzlichen Ergebnisse sind möglicherweise auf eine unterschiedliche Methodik und Erfahrung der Entwickler (Entwickler mit langjähriger Erfahrung im Gegensatz zu Studenten mit vergleichsweise wenig Erfahrung) zurückzuführen.

Auf Grundlage der angeführten Forschungsergebnisse kann die Frage, ob TDD den Entwicklungsprozess positiv beeinflusst, nicht eindeutig beantwortet werden. Es kann jedoch der Schluss gezogen werden, dass ein testgetriebenes Vorgehen den Entwicklungsprozess zumindest nicht negativ beeinflusst.

3.4.3 Verhaltensgetriebene Entwicklung

Das Konzept der verhaltensgetriebenen Entwicklung (BDD; für engl.: Behaviour-Driven Development) entstand als eine Erweiterung des TDD-Ansatzes, kann jedoch auch losgelöst

[25] *International Business Machines Corporation*, US-amerikanisches IT-Unternehmen

davon für einen klassischen Ansatz verwendet werden, bei dem die Tests erst nach der Implementierung einer Komponente geschrieben werden. Im Wesentlichen soll BDD die Frage beantworten, welche Funktionen bzw. – um den BDD-Terminus zu verwenden – welches Verhalten einer Komponente es zu testen gilt. Dan North, der als Entwickler dieser agilen Methode gilt, schlägt vor, alle Tests als Satz nach dem Muster „*The component should do this and that*“ zu formulieren, was eine der Kernideen des BDD ist (vgl. NORTH, 2006). Der Fokus wird damit auf die Untersuchung des Verhaltens einer Komponente gelegt. Das Wort „test“ soll auf Vorschlag von NORTH (2006) beim Schreiben von Tests hingegen komplett gestrichen werden.

Indem die Tests in Satzform beschrieben werden, klären sich viele Unklarheiten, die Entwickler beim Schreiben von Tests häufig haben, von allein – so etwa die Frage danach, wie viel Funktionalität ein einzelner Test prüfen sollte. Dan North trifft dazu folgende Aussage: „How much to test becomes moot – you can only describe so much behaviour in a single sentence.” (NORTH, 2006). Als Richtlinie gilt demnach, dass nur so viel Funktionalität in einem Test geprüft werden sollte, wie sich in einem einzelnen Satz beschreiben lässt.

3.4.4 Testen von React-Anwendungen

Beim Testen von React-Anwendungen gibt es einige Besonderheiten zu beachten. Zunächst stellt sich die Frage danach, *was* alles zu testen ist. Eine React-Anwendung besteht in der Regel nicht nur aus der Bibliothek React selbst, sondern einer Kombination von vielen Frameworks und Webtechnologien (siehe *2.3*). Nicht alle davon machen Sinn, getestet zu werden. Beispielsweise ist es überflüssig zu testen, ob die JSX-Syntax in valides JS übersetzt wird. Dies ist die Aufgabe des Transpilers, und eine nicht-valide Syntax würde vom Browser beim Parsen ohnehin bemerkt werden.

In jedem Fall sollten jedoch die React-Komponenten und auch Redux getestet werden, sofern dieses verwendet wird. Beim Testen von React-Komponenten sollte zumindest geprüft werden, ob die erzeugte Komponente das richtige, zu erwartende HTML rendert (vgl. SCOTT, 2017). Sofern es für die jeweilige Komponente relevant ist, empfiehlt SCOTT (2017) auch folgendes zu testen:

- die durch die Eltern-Komponente erhaltenen Props,
- den State der Komponente (sofern vorhanden), und
- das Verhalten bei einer Interaktion durch den Nutzer.

Anstatt React-Komponenten für jeden Test vollständig über das DOM zu rendern, gibt es auch die Möglichkeit des Shallow Renderings (SR). Dabei wird die jeweilige Komponente nur eine Ebene tief gerendert, d.h. es wird lediglich das Wurzel-Element oder die Wurzel-Komponente, die innerhalb der `return()`-Anweisung zurückgeliefert wird, gerendert. Zur Erklärung sei folgender Codeausschnitt (Listing 8) einer geschachtelten React-Komponente gegeben:

```
import React from 'react';
import InnerComponent from './InnerComponent';

class OuterComponent extends React.Component {
  render(){
    return(
      <div>
        <h1>Headline</h1>
        <InnerComponent />
      </div>
    );
  }
}
```

Listing 8: React-Komponente zur Demonstration des Shallow Rendering

Während beim normalen Rendern auch die innere Komponente `InnerComponent` beim Erzeugen von `OuterComponent` instanziiert werden würde, ist dies beim Shallow Rendering nicht der Fall. Die einzige Assertion, die für die innere Komponente gemacht werden kann, ist, dass diese vorhanden ist. Es können jedoch keine Assertions in Bezug auf die Struktur dieser Komponente, also deren inneres HTML, gemacht werden. Shallow Rendering benötigt kein DOM und ist somit schneller, und hat noch dazu den Vorteil, dass eine Komponente komplett isoliert betrachtet werden kann, ohne sich um das Verhalten der Kind-Komponente(n) kümmern zu müssen (vgl. FACEBOOK, 2017b). Zur Implementierung von Tests mittels Shallow Rendering empfiehlt FACEBOOK (2017b) die Shallow Rendering API von *Enzyme*[26].

Beim Testen des Redux-Codes sind vor allem die Action Creator und Reducer-Funktionen von Interesse.

[26] Enzyme: Testing Utiliy für React, siehe http://airbnb.io/enzyme/docs/api/shallow.html [Stand 03.09.2017]

4 Vorüberlegungen und Methodik

4.1 Verwendete Test-Frameworks

4.1.1 Mocha

Mocha[27] ist ein beliebtes und weit verbreitetes Open-Source JS-Test-Framework (siehe 3.4). Erstmals 2011 veröffentlicht, liegt das TF aktuell in der Version 3.5.0 vor (Stand: 2.8.2017). Mocha ist in erster Linie ein Framework zum Testen von reinem JS, das sich jedoch dadurch auszeichnet, dass es nahezu nach Belieben angepasst werden kann und somit z.B. auch zum Testen von React verwendet werden kann. Das Grundgerüst eines Tests mit Mocha hat folgende Syntax (Listing 9):

```
describe("Test object", () => {
  it("should be HELLO WORLD", function() {
    // Eigentlicher Test mittels Assertions
  });
});
```

Listing 9: Die grundlegende Struktur eines Tests mit Mocha

Alle Tests werden innerhalb der `describe()`-Funktion zusammengefasst, welche als erstes Argument den Namen des Testobjekts (z.B. Name einer zu testende Funktion) übergeben bekommen sollte. Dieser Name dient dabei lediglich der besseren Lesbarkeit für den Tester: Bei der Auswertung eines jeden Testdurchlaufs steht dieser an oberster Stelle, sodass sofort erkannt werden kann, welches Testobjekt in dem jeweiligen Testdurchlauf betrachtet wird. Es können auch mehrere `describe()`-Funktionen ineinander geschaltet werden, wobei die äußere Funktion dann z.B. eine Testsuite beschreibt und die Innere das spezifische Testobjekt.

Die `it()`-Funktion beinhaltet dann den eigentlichen Test und bekommt als erstes Argument die Testbeschreibung übergeben, welche in einem BDD-Stil formuliert sein sollte (siehe *3.4.3*). Bei der Auswertung des Testdurchlaufs im Terminal werden alle Beschreibungen der einzelnen Tests untereinander aufgelistet, jeweils mit einem symbolischen Hinweis, ob der entsprechende Test bestanden wurde oder nicht.

[27] Siehe https://mochajs.org/ [Stand 02.08.2017]

Für die Assertions verwendet Mocha standardmäßig das integrierte Node.js-Modul *assert*, erlaubt aber alternativ die Verwendung einer beliebigen Bibliothek für Assertions. *Chai*, *should.js* und *expect.js* sind Beispiele für bekannte Assertion-Bibliotheken, die sich im Wesentlichen in der Syntax der Assertions und dem Funktionsumfang unterscheiden. Für diese Arbeit wurde sich für Chai als Assertion-Bibliothek entschieden.

Chai verfügt nicht nur über eine größere Anzahl an Matchers, sondern hat darüber hinaus die Besonderheit, dass es nicht *eine* konkrete Syntax vorschreibt: Chai stellt dem Entwickler drei syntaktische Stile zum Schreiben der Tests zur Verfügung, die von der Funktionalität äquivalent sind, sich jedoch unterschiedlich lesen. Das folgende Listing (10) zeigt die gleiche Assertion, welche prüft, ob die Variable hello dem Wert `HELLO WORLD` entspricht, in den drei unterschiedlichen Stilen *should*, *expect* und *assert*:

```
var hello = "HELLO WORLD";

// Variante 1 - should-Syntax
chai.should();      // muss einmalig aufgerufen werden
hello.should.equal("HELLO WORLD");  // eigentlicher Test

// Variante 2 - expect-Syntax
var expect = chai.expect;   // einmalig
expect(hello).to.equal("HELLO WORLD");  // eigentlicher Test

// Variante 3 – assert-Syntax
var assert = chai.assert;   // einmalig
assert.equal(hello, "HELLO WORLD");     // eigentlicher Test
```

Listing 10: Die drei Syntax-Styles von Chai

Im Folgenden wird aus Gründen der persönlichen Präferenz die `expect`-Syntax verwendet.

Standardmäßig können mit Chai allein jedoch keine React-Komponenten getestet werden. Aus diesem Grund muss zusätzlich noch das bereits unter *3.4.4* erwähnte Test-Werkzeug Enzyme verwendet werden, welches genau das erlaubt. Enzyme bietet dem Entwickler drei Möglichkeiten, React-Komponenten zu testen:

- **Shallow Rendering** (siehe 3.4.4)
- **Full DOM Rendering:** Das komplette DOM wird gerendert, DOM-Manipulationen, z.B. durch simulierte Nutzerinteraktionen, können somit gut getestet werden.
- **Static Rendering API:** Die React-Komponente wird an Hand der statischen HTML-Struktur analysiert. Es wird kein vollständiges DOM erzeugt, weswegen sich diese Methode nicht zum Testen von DOM-Manipulationen eignet.

4.1.2 Jest

Jest[28] ist ein von Facebook entwickeltes Open-Source Test-Framework, das speziell zum Testen von React-Anwendungen dient, jedoch ebenso für reines JS verwendet werden kann. Jest ist seit 2014 auf dem Markt und wurde seitdem stetig weiterentwickelt. Inzwischen ist das TF in der Version 20.0.4 verfügbar (Stand 3.8.2017). Jest wirbt auf der eigenen Website damit, dass das Framework ohne irgendwelche Konfigurationen sofort zum Testen von React-Applikationen verwendet werden kann. Eine zusätzliche Bibliothek für Assertions wird nicht benötigt. Syntaktisch schreiben sich die Tests mit Jest nahezu identisch wie unter Mocha (in Kombination mit Chai). Der folgende Code (Listing 11) zeigt den Beispiel-Test aus 4.2.1 mit Jest:

```
var hello = "HELLO WORLD";

describe("Test object", () => {
  it("should be HELLO WORLD", () => {
    expect(hello).toBe("HELLO WORLD");  // eigentlicher Test
  });
});
```

Listing 11: Einfacher Beispieltest mit Jest

Wie auch bei Mocha bilden den äußeren Rahmen eines Tests mit Jest die Funktionen `describe()` und `it()`, welche analog verwendet werden. Alternativ kann anstelle von `it()` auch das Alias `test()` verwendet werden. Da letzteres jedoch nicht dem BDD-Stil entspricht (siehe 3.4.3) soll für diese Arbeit die `it()`-Syntax genutzt werden. Auch die Assertion selbst liest sich ähnlich: Jest verwendet hierfür eine Syntax, die nahezu identisch mit der expect-Syntax von Chai ist.

Eine Besonderheit von Jest ist das bereits in der Einleitung erwähnte Snapshot Testing, welches in Version 14 eingeführt wurde und vor allem das Testen von React-Komponenten erleichtern soll. Anstatt für jede Komponente manuell Assertions auf Grundlage des gerenderten HTMLs zu formulieren, wird bei dieser Art von Test einfach ein sogenannter Snapshot (dt.: Schnappschuss) generiert, welcher eine Repräsentation des HTML-Codes der erzeugten Komponente darstellt. Der Tester muss nach dem Erstellen eines neuen Snapshots manuell prüfen, ob dieser dem planmäßigen Ergebnis entspricht. Ist das nicht der Fall, weist dies auf einen Fehler hin, welcher behoben werden muss.

[28] Siehe https://facebook.github.io/jest/ [Stand 03.08.2017]

Beim jedem neuen Durchlauf desselben Tests wird ein neuer Snapshot erstellt und mit der vorherigen Version verglichen. Ist dieser mit dem vorherigen identisch, gilt der Test automatisch als bestanden. Unterscheiden sich die Snapshots, kann dies zwei Gründe haben: Die Komponente kann aktualisiert worden sein, wodurch sich erwartungsgemäß auch das erzeugte HTML-Markup ändert, oder die Komponente ist fehlerhaft. In beiden Fällen wird der Test zunächst fehlschlagen. Es liegt daraufhin an dem Tester, die Snaphot-Datei zu inspizieren und zu überprüfen, ob die Änderung, welche farblich gekennzeichnet wird, erwartungsgemäß ist oder nicht. Im Falle von ersterem kann der neue Snapshot übernommen werden, womit die alte Version überschrieben wird und derselbe Test nun bestanden wird. Andernfalls wird die geänderte Snapshot-Datei wieder verworfen und der zugrundeliegende Fehler muss vom Entwickler behoben werden.

Obwohl Jest auf das Testen von React spezialisiert ist, beschränkt sich der in dem Framework standardmäßig enthaltene Umfang dafür auf die Snapshot-Testing-Funktionalität. Um React-Komponenten auch auf die „klassische Art“ – d.h. mit einem kompletten DOM-Rendering oder per Shallow Rendering – testen zu können, muss auch Jest in Kombination mit Enzyme verwendet werden.

4.2 Qualitätskriterien zur Gegenüberstellung

Zum qualitativen Vergleich der beiden ausgewählten Test-Frameworks wurden folgende Qualitätskriterien aufgestellt (siehe Tabelle 1). Die Kriterien sind in absteigender Relevanz aufgelistet, d.h. je weiter oben ein Kriterium steht, desto höher wird es in der Auswertung gewichtet. Die obersten drei Kriterien beziehen sich allesamt auf die selbst geschriebenen Tests für die Second Screen App.

Kriterium	Beschreibung
Umsetzbarkeit	Lassen sich alle Tests, die durchgeführt werden sollen, mit dem jeweiligen TF implementieren?
Implementierung	Wie hoch ist die Komplexität bei der Implementierung der einzelnen Tests? Hierzu zählen Nachvollziehbarkeit der Syntax und die Menge an Code, die für die Implementierung eines Tests benötigt wird.
Automatisierungsgrad	Lassen sich die Tests komplett automatisiert durchführen, oder ist eine menschliche Verifizierung erforderlich? Ein hoher Automatisierungsgrad ist erstrebenswert, insbesondere für die spätere Einbindung der Tests in das CI-System von Sensape.
Tauglichkeit für TDD	Eignet sich das TF auch für die Durchführung von Tests nach dem TDD-Ansatz?
Verhalten im Fehlerfall:	Liefert das TF eine verständliche, nützliche Fehlermeldung, wenn ein Test nicht bestanden wird?
Installation und Konfiguration	Wie einfach lässt sich das TF dem Projekt hinzufügen? Sind viele Installationen und Konfigurationen notwendig, um eine funktionstüchtige Testumgebung aufzusetzen?

Tabelle 1: Qualitätskriterien zur Gegenüberstellung der Test-Frameworks

Nicht berücksichtigt werden soll der Faktor Performanz. Dies hat den Hintergrund, dass das langfristige Ziel ist, alle Tests bei Sensape automatisiert über das CI-System durchlaufen zu lassen. Der Untersuchung Performanz ist somit für diese Arbeit nicht von Relevanz.

4.3 System Under Test

Als SUT soll für diese Arbeit eine bereits bestehende Webapplikation der Firma Sensape, die sogenannte Second Screen App, verwendet werden. Diese Anwendung ist essenzieller Teil eines der wichtigsten Produkte von Sensape – der *Phantastic Photobox*[29]. Bei der Phantastic Photobox handelt es sich um eine Digital Signage-Stele, welche am häufigsten von Unternehmen für den Auftritt auf einer Messe gemietet wird. Die Phantastic Photobox verbindet dabei Augmented Reality mit künstlicher Intelligenz: Eine spezielle Software analysiert über eine eingebaute Kamera die Gesichter der Personen, die sich vor den großen

[29] Siehe https://phantastic-photobox.com/ [Stand 04.08.2017]

Bildschirm auf der Vorderseite der Photobox stellen. Je nach erfasstem Alter, Geschlecht und Gemütszustand werden die Personen auf dem großen Bildschirm in eine andere erweiterte Realität versetzt, indem z.B. virtuelle Sprechblasen über den Köpfen angezeigt werden, wobei der Inhalt genau auf das Alter und Geschlecht der Personen abgestimmt ist.

Abbildung 9: Die Phantastic Photobox im Einsatz
(Quelle: SENSAPE (2017))

Über einen Touchpoint auf dem großen Bildschirm kann eine Fotoaufnahme ausgelöst werden und das Szenario, wie es zu dem Zeitpunkt auf virtuellen Spiegel zu sehen ist, so in Form eines Bildes festgehalten werden. Dieses kann anschließend – sofern vom Nutzer gewünscht – direkt ausgedruckt werden. Hierzu verfügt die breite Stele seitlich über einen zweiten, kleineren Touch-Bildschirm, genannt Second Screen, auf welchem in einem Kiosk Browser die bereits erwähnte Second Screen App läuft. Die komplette Photobox mit dem integrierten Second Screen ist in Abbildung 9 zu sehen.

Die App erlaubt es dem Nutzer nicht nur, sein eben geschossenes Bild auszuwählen und zu drucken, sondern optional auch noch seine Kontaktdaten zu hinterlassen, die Erlaubnis zu geben, das Bild über Social Media zu teilen, und noch mehr. Der Nutzer wird dabei standardmäßig durch 5 Dialoge, sogenannte Views, geleitet, beginnend mit der Auswahl des Bildes und endend mit einem View, der den Nutzer darauf hinweist, dass das Bild nun gedruckt wird.

Aus technischer Sicht handelt es sich bei der Applikation um eine React-Anwendung auf Basis von Node.js. Das komplette UI der Applikation ist mit React, unter Verwendung der JSX-Syntax, umgesetzt worden: Jeder View der App ist eine React-Komponente, der sich

seinerseits aus mehreren Komponenten zusammensetzt. Sämtliche Daten, die nicht nur für einen bestimmten View, sondern die Applikation „als Ganzes“ von Wichtigkeit sind, werden mittels Redux in einem globalen State verwaltet. Dieser wiederum setzt sich aus insgesamt 4 Teil-Zuständen zusammen, die ihrerseits von jeweils einem Reducer verwaltet werden. So gibt es z.B. einen `dialog`-Reducer, der dafür verantwortlich ist, in seinem State den Namen des aktuellen und zurückliegenden Views zu speichern, und den State entsprechend zu aktualisieren, wenn der nächste View gerendert wird.

Darüber hinaus kommt Webpack als Modul-Bundler zum Einsatz, wodurch z.B. eine Trennung zwischen Entwicklungs- und Produktionsumgebung umgesetzt wird, sowie Babel als Transpiler zum Übersetzen des JSX- und ES2015-Codes in JS-Code nach dem ES5-Standard.

4.4 Wahl der Testfälle

Im Folgenden galt es, sich für einige ausgewählte Testobjekte des SUT zu entscheiden und dafür konkrete Testfälle aufzustellen. Die Schwierigkeit bestand darin, diese so zu wählen, dass zwei Bedingungen erfüllt werden: Einerseits sollte ein möglichst großes Spektrum abgedeckt werden, sodass beide Test-Frameworks in möglichst vielen Kategorien miteinander verglichen werden können. Andererseits sollten die Tests aber auch gehaltvoll sein, d.h. es sollte kein Test implementiert werden, der sich zwar für einen Vergleich gut eignet, jedoch keinen oder wenig Nutzen für das SUT bringt.

Auf Grundlage dieser Vorüberlegung und der unter 3.4.4 vorgestellten Richtlinien zum Testen von React-Anwendungen wurde zunächst ein erster, grober Plan für die Testfälle erstellt. Folgende Bereiche sollten abgedeckt werden:

- Test von mindestens 1 React-Komponente pro Testsuite
- Mindestens 1 Testsuite, die sich mit Jest komplett mit Snapshot Tests umsetzen lässt (zum Vergleich: Snapshot Test gegen klassischen Test)
- Mindestens 1 Testsuite, bei der alle React-Komponenten via Shallow Rendering getestet werden
- Mindestens 1 Testsuite, bei der alle React-Komponenten via Full DOM Rendering getestet werden
- Test von mindestens 1 Reducer-Funktion
- Mindestens 1 Test zur Integration von React und Redux (z.B. React-Komponente und Reducer)
- Mindestens 1 Test mit simulierter Nutzerinteraktion (Interaktionstest)

Aufbauend darauf wurden anschließend die Testobjekte ausgewählt und in logisch zusammengehörige Gruppen zu Testsuites zusammengefasst. Die überlegten Testfälle wurden präzisiert und in Tabellenform als Testfallspezifikation festgehalten. Diese befinden sich in den Anlagen unter dem Punkt *Testfallspezifikationen*. Insgesamt wurden 30 Testfälle, aufgeteilt auf 5 Testsuites, erstellt, die in Form von Tests für beide Test-Frameworks zu implementieren waren.

4.5 Testumgebung

Verwendete Hardware:

- Prozessor: *AMD FX-8320* (8-Kerne)
- Hauptspeicher: DDR3, 4GB

Verwendete Software:

- Betriebssystem: *Ubuntu 14.04.5 LTS*
- Second Screen App: Version 0.11.1
- Node.js: Version 7.10.0
- Mocha: Version 3.5.0
- Jest: Version 20.0.4

5 Vergleich der Test-Frameworks

In diesem Kapitel erfolgt die eigentliche Gegenüberstellung der beiden Test-Frameworks. Aus Gründen der Kontinuität wird stets erst das Vorgehen für Mocha und anschließend für Jest beschrieben, auch wenn dies nicht immer der tatsächlichen Reihenfolge entspricht, in der vorgegangen wurde.

Bei der Implementierung der Tests in den Kapiteln *5.2* bis *5.5* werden die importierten Dateien, die für die jeweilige Testsuite benötigt wurden, lediglich bei der Beschreibung zu Mocha aufgelistet. Sofern nicht explizit darauf hingewiesen wird, ist diese Liste für dieselbe Testsuite unter Jest identisch, mit Ausnahme der `expect`-Syntax von Chai, welche bei Jest nicht verwendet wird (siehe 4.1.2).

Des Weiteren wurde in beiden Testumgebungen ein Verzeichnis mit der Bezeichnung `constants` innerhalb des jeweiligen Test-Verzeichnisses erstellt, in welches sämtliche Mock-Objekte für die einzelnen Test-Dateien (jeweils in eine separate Datei mit derselben Bezeichnung wie die zugehörige Test-Datei) ausgelagert wurden. Dies dient dem einfachen Zweck, die Übersichtlichkeit der Test-Dateien zu erhöhen.

Es sei zudem angemerkt, dass lediglich drei der fünf implementierten Testsuites ausführlich erläutert werden, um den Umfang nicht zu übersteigen. Die Ergebnisse der anderen beiden Testgruppen werden an passender Stelle in der Auswertung mit eingebracht. Der komplette Quellcode aller 5 Testsuites für beide Frameworks kann in den Anlagen eingesehen werden.

5.1 Installation und Konfiguration

5.1.1 Mocha

Zum Aufsetzen von Mocha mussten zunächst folgende Pakete über NPM als Dev-Dependencies installiert werden (siehe Tabelle 2):

Packagename	Beschreibung
`mocha`	Das Test-Framework Mocha
`chai`	Die verwendete Assertion-Library; benötigt zum Schreiben von Assertions
`enzyme`	Test Utility; benötigt zum Rendern von React-Komponenten für die Tests (inkl. Shallow Rendering)

`jsdom`	Headless Browser, der ein künstliches DOM für Node.js emuliert. Es ist damit die Grundlage dafür, die Tests ohne einen richtigen Webbrowser durchzuführen
`react-test-renderer`	Renderer; wird von Enzyme benötigt
`babel-preset-airbnb`	Babel-Transpiler für Enzyme
`babel-preset-stage-2`	Transpiler zum Testen von ES2015-Code
`babel-register`	Compiler für die Tests
`redux-mock-store`	Künstlicher Redux-Store (Mock-Objekt); für Test von Container-Komponenten benötigt
`ignore-styles`	Verwendet, um sämtliche eingebundene Style-Dateien (hier: Sass-Dateien) zu ignorieren.[30]

Tabelle 2: Liste der benötigten Packages zum Aufsetzen von Mocha

Anschließend waren noch einige Konfigurationen vorzunehmen. Im Root-Verzeichnis musste zunächst eine Datei `.babelrc` angelegt werden. Hier mussten die Packages `babel-preset-airbnb` und `babel-preset-stage-2` konfiguriert werden, um diese für die Tests nutzen zu können (siehe Listing 12).

```
/* /.babelrc */
{ "presets": ["airbnb", "stage-2"] }
```

Listing 12: Inhalt der .babelrc-Datei für Mocha

Als nächstes wurde ein Verzeichnis `test` im Root-Verzeichnis angelegt, in welchem später die Tests abgelegt werden sollen. In `test` wurde zunächst eine Datei zur Konfiguration von JSDOM angelegt (`dom.js`). Hierzu wurde die Standard-Konfiguration aus AIRBNB (2017b) übernommen. Die komplette Konfigurationsdatei von JSDOM ist in Listing 13 zu sehen.

[30] Dieses Package musste nachinstalliert werden, da Mocha nicht mit den Sass-Dateien, welche für das Styling der Komponenten verwendet wurden, umgehen konnte.

```
/* /test/dom.js */

const { JSDOM } = require('jsdom');

const jsdom = new JSDOM('<!doctype html><html><body></body></html>');
const { window } = jsdom;

function copyProps(src, target) {
  const props = Object.getOwnPropertyNames(src)
    .filter(prop => typeof target[prop] === 'undefined')
    .map(prop => Object.getOwnPropertyDescriptor(src, prop));
  Object.defineProperties(target, props);
}

global.window = window;
global.document = window.document;
global.navigator = {
  userAgent: 'node.js',
};
copyProps(window, global);
```

Listing 13: Konfiguration von JSDOM zum Testen mit Mocha

Im Laufe der Implementierung der Tests musste die Datei noch einmal angepasst werden. Der Grund hierfür war folgende Fehlermeldung (siehe Abbildung 10), welche bei vereinzelten Tests auftrat:

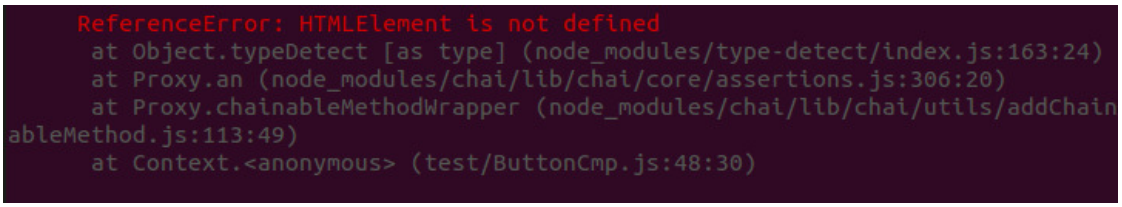

Abbildung 10: Fehlermeldung bei Test mit Mocha

Der Fehler konnte behoben werden, indem der Konfigurationsdatei von JSDOM folgende Zeile hinzugefügt wurde (Listing 14):

```
global.HTMLElement = window.HTMLElement;
```

Listing 14: Ergänzung in dom.js

Anschließend wurde im `test`-Verzeichnis eine Datei `example-test.js` erstellt, welche lediglich zum Prüfen dienen sollte, ob alles richtig aufgesetzt wurde. Dieser wurden zwei einfache Tests hinzugefügt: Zum einen ein Test zur Überprüfung, ob die Addition zweier Zahlen das richtige Ergebnis liefert, zum anderen ein simpler Test für eine React-Komponente.

Wird Mocha ohne zusätzliche Tools verwendet, reicht zum Ausführen der Tests der Aufruf von `mocha` über die Konsole. Das Programm sucht dann automatisch nach einem Verzeichnis mit dem Namen `test` und führt alle Tests in allen darin enthaltenen Dateien aus. Da im konkreten Fall noch einige Parameter übergeben werden mussten, ist das Kommando deutlich länger. Um dieses nicht jedes Mal von Hand eingeben zu müssen, wurde in der `package.json` innerhalb `scripts`-Objekts folgender Eintrag hinzugefügt (Listing 15):

```
"test": "mocha --compilers js:babel-register --require ignore-styles
--require ./test/dom.js --recursive"
```

Listing 15: Skript zum Ausführen der Tests unter Mocha

Über den Aufruf von `npm test` konnten somit nun beide Tests ausgeführt und erfolgreich durchlaufen werden, womit die Einrichtung von Mocha abgeschlossen war.

Abbildung 11 zeigt die zugehörige Konsolenausgabe mit der Testauswertung. Das Testergebnis wird in übersichtlicher Weise dargestellt: Alle durchlaufenen Tests werden untereinander aufgelistet; der grüne Haken vor der Beschreibung lässt sofort erkennen, dass der jeweilige Test erfolgreich war. Darunter wird zusätzlich noch die Anzahl der bestandenen (und ggf. nicht bestandenen) Tests angezeigt, so wie die für den kompletten Durchlauf aller Tests benötigte Zeit.

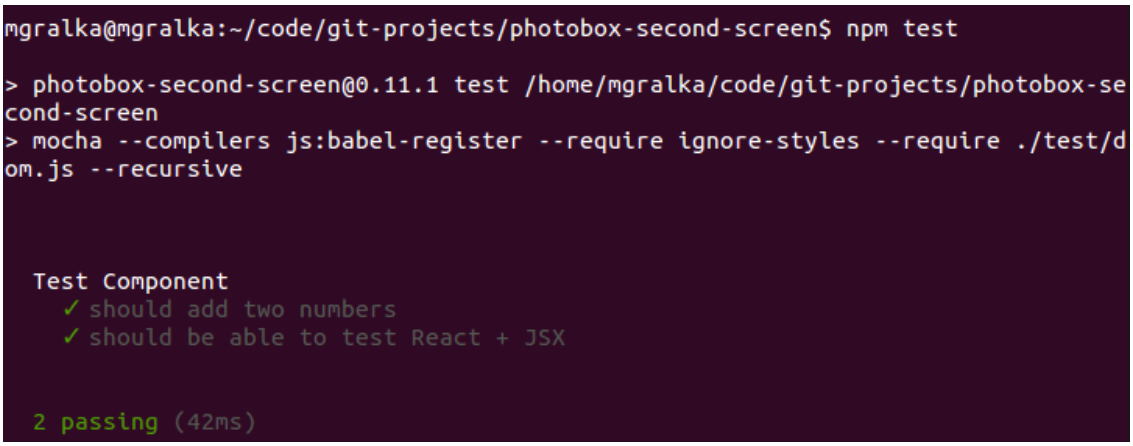

Abbildung 11: Konsolenausgabe Mocha nach erfolgreichem Testdurchlauf

5.1.2 Jest

Zur Installation von Jest mussten folgende Packages installiert werden (Tabelle 3):

Packagename	Beschreibung
`jest`	Das Test-Framework Jest
`enzyme`	Test Utility; benötigt zum Rendern von React-Komponenten für die Tests (inkl. Shallow Rendering)
`react-test-renderer`	Renderer; wird von Enzyme benötigt
`babel-jest`	Transpiler für Jest
`babel-preset-stage-2`	Transpiler zum Testen von ES2015-Code
`redux-mock-store`	Künstlicher Redux-Store (Mock-Objekt); für Test von Container-Komponenten benötigt
`Identity-obj-proxy`	Verwendet, um sämtliche eingebundene Style-Dateien (hier: Sass-Dateien) zu ignorieren (analog zu `ignore-styles` für Mocha)

Tabelle 3: Liste der benötigten Packages zum Aufsetzen von Jest

Anschließend wurde auch in dieser Testumgebung ein Verzeichnis für die Tests erstellt. Bei Jest trägt dieses immer die Bezeichnung `__tests__`. Alternativ erkennt Jest auch sämtliche Dateien mit den Suffixen `.test.js` oder `.spec.js` als Test-Dateien.

Vor dem Erstellen der ersten Tests mussten jedoch auch bei Jest noch einige Konfigurationen vorgenommen werden. Zunächst musste, wie bereits unter 5.1.1, eine `.babelrc`-Datei angelegt werden, welcher folgende Einträge hinzugefügt wurden (Listing 16):

```
/* /.babelrc */
{ "presets": ["es2015", "react", "stage-2"] }
```

Listing 16: Inhalt der .babelrc-Datei unter Jest

Anschließend musste Jest so angepasst werden, dass React-Komponenten ohne Berücksichtigung des Stylings getestet werden können. In der `package.json` kann das TF über die Eigenschaft `"jest"` konfiguriert werden. Zum Ignorieren von Sass-Dateien durch den `identity-obj-proxy` wurden die unter Listing 17 aufgeführten Zeilen hinzugefügt:

```
/* /package.json */
"jest": {
  "moduleNameMapper": {
    "\\.(scss)$": "identity-obj-proxy"
  }
}
```

Listing 17: Konfiguration von Jest in der package.json

(vgl. FACEBOOK, 2017c [Code für Sass angepasst])

Innerhalb des `jest`-Objekts wurde zusätzlich folgender Eintrag hinzugefügt (Listing 18):

```
"testPathIgnorePatterns": ["constants"]
```

Listing 18: Zusätzlicher Eintrag zur Konfiguration von Jest

Dies ermöglicht die Auslagerung der verwendeten Mock-Objekte in das Verzeichnis `constants`. Jest sucht in diesem Verzeichnis somit nicht nach Test-Dateien. [31]

Als nächstes wurde analog zu 5.1.1 ein Script namens `test` der `package.json` hinzugefügt, über welches alle Tests innerhalb des `__tests__`-Verzeichnisses automatisiert ausgeführt werden können (siehe Listing 19).

```
/* /package.json, innerhalb des script-Objekts */
"test": "jest --verbose"
```

Listing 19: Skript zum Ausführen der Tests unter Jest

Zum Überprüfen, ob das Test-Setup funktioniert, wurde zum Schluss wieder eine `example-test.js` in dem Test-Verzeichnis angelegt, welche mit zwei einfachen Tests gefüllt wurde. Hierzu wurden die Tests aus dem Mocha-Setup übernommen und lediglich die Syntax entsprechend angepasst. Über das Kommando `npm start` wurden diese anschließend ausgeführt. Eine Fehlermeldung blieb aus und beide Tests wurden bestanden, womit die Einrichtung von Jest abgeschlossen war.

Abbildung 12 zeigt einen Screenshot der Konsolenausgabe des ersten Testdurchlaufs mit Jest. Diese ist der Ausgabe von Mocha im Wesentlichen sehr ähnlich, zeigt jedoch noch einige zusätzliche Informationen an, wie etwa die benötigte Zeit für den Durchlauf jedes einzelnen Tests. Des Weiteren ist zu erkennen, dass die Snapshots getrennt aufgelistet

[31] Das Standardverhalten ist, dass Jest in jeder Datei in jedem Unterverzeichnis von `__test__` mindestens einen Test erwartet.

werden. Die Snapshot Testing-Funktion von Jest wurde an dieser Stelle noch nicht ausprobiert, weswegen in der Auswertung hinter `Snapshots` hier die Ausgabe `0 total` (noch keine Snapshots vorhanden) zu sehen ist.

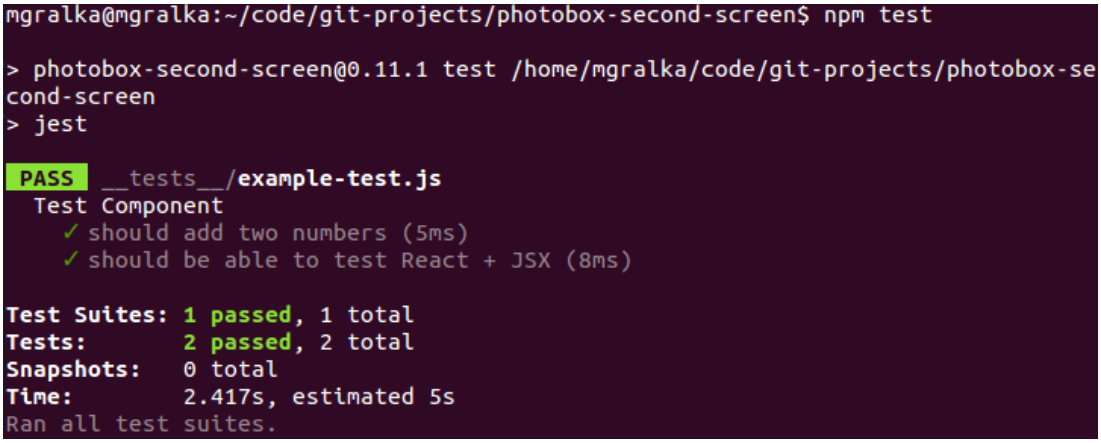

Abbildung 12: Konsolenausgabe Jest nach erfolgreichem Testdurchlauf

5.1.3 Auswertung

Für die Aufsetzung der beiden Test-Frameworks musste in beiden Fällen eine große Menge an Packages installiert werden: Bei Mocha insgesamt zehn, bei Jest immerhin nur sieben. Der größte Vorzug beim Aufsetzen von Jest ist, dass der Headless Browser (JSDOM) nicht zusätzlich installiert und konfiguriert werden muss, da dieser bereits integriert ist. Die richtige Konfiguration von JSDOM hat bei Mocha am längsten Zeit beansprucht und musste auch während der Implementierung des Tests nochmal angepasst werden.

Auch das Kommando zum Ausführen der Tests ist bei Jest deutlich kürzer: Hier genügt bereits das Kommando jest (die Flag `--verbose` aktiviert lediglich den Verbose-Modus für detailliertere Ausgaben auf der Kommandozeile). Bei Mocha müssen hingegen mehrere zusätzliche Parameter übergeben werden, um die Tests überhaupt ausführen zu können. Jest musste dafür in der `package.json` noch gesondert konfiguriert werden, womit sich dies in etwa aufhebt.

Zusammenfassend lässt sich dennoch das Fazit ziehen, dass Jest auf Grund der geringen Anzahl an benötigten Packages und dem bereits integrierten JSDOM einfacher aufzusetzen ist als Mocha. Insgesamt wurde für die Einrichtung von Mocha auch mehr als die doppelte Zeit benötigt, wobei sich dies dadurch etwas relativiert, dass Mocha zeitlich gesehen vor Jest aufgesetzt wurde.

5.2 Testsuite 1

5.2.1 Testobjekte

In der ersten Testsuite soll eine einzelne React-Komponente mit dem Namen `StpAppBar` getestet werden. Hierbei handelt es sich um eine Leiste, die in jedem der einzelnen Views oben im Bild angezeigt wird. Abbildung 13 zeigt die Komponente in 2 unterschiedlichen Views, hervorgehoben durch eine rote Umrahmung.

Abbildung 13: StpAppBar ohne und mit Icon

Auf dem `StartView`, dem ersten Dialog der App (siehe Abbildung 13, links), enthält diese lediglich einen Text. Diesen bekommt die Komponente bei der Instanziierung über das Prop `title` übergeben. In allen anderen Views, mit Ausnahme des letzten, enthält die Komponente ebenfalls einen übergebenen Text, zusätzlich aber noch ein Icon[32], über welches der Nutzer wieder zum `StartView` zurückgelangt. Damit die Komponente mit dem „Zurück"-Icon dargestellt wird, muss sie mit dem optionalen Prop `backicon={true}` erzeugt werden.

[32] Icon: ein Piktogramm

Das Prop `title` ist hingegen obligatorisch.[33] Wird die Komponente ohne dieses erzeugt, resultiert dies in einer Fehlermeldung durch den Compiler.

Die `StpAppBar`-Komponente ist im Inneren wiederum aus einer Komponente der UI-Bibliothek *React Toolbox*[34] zusammengebaut. React Toolbox stellt eine Sammlung von vorgefertigten, anpassbaren React-Komponenten im Material Design[35] zur Verfügung, und wird für mehrere Komponenten der Second Screen App verwendet. Die hier verwendete Toolbox-Komponente hat die Bezeichnung AppBar und rendert intern einen HTML-Header (`<header>`).

Die Icon-Komponente ist hingegen von Grund auf selbstgeschrieben und erzeugt ein SVG-Element[36].

5.2.2 Testziele und Testfälle

Durch die erste Testsuite soll gewährleistet werden, dass die Komponente stets den richtigen Text anzeigt, und das Icon in der Leiste angezeigt wird, wenn das Prop `backicon` mit dem richtigen Wert vorhanden ist.

Konkret sollen folgende Szenarien geprüft werden:

- Es soll kein Text in der Leiste angezeigt werden, wenn ein leerer String als `title` übergeben wird (/TC010/).
- Bekommt title einen String mit weniger als 30 Zeichen übergeben, soll dieser in der Leiste angezeigt werden (/TC020/).
- Bekommt `title` einen String mit mehr als 30 Zeichen übergeben, soll kein Text angezeigt werden (/TC030/). Der Titel wäre in dem Fall zu lang, um richtig angezeigt zu werden.
- Wird die Komponente ohne das Prop `backicon` instanziiert, soll die Leiste kein Icon enthalten (/TC040/).
- Wird die Komponente mit dem Prop `backicon={true}` instanziiert, soll die Leiste das „Zurück“-Icon enthalten (/TC050/).
- Wird die Komponente mit dem Prop `backicon={true}` instanziiert, soll auch hier der als title übergebene Text (Länge von <30 Zeichen) angezeigt werden (/TC060/).

Bei /TC030/ wird dabei ein Verhalten geprüft, dass zum derzeitigen Stand der Entwicklung noch nicht implementiert wurde. Aktuell ist es also auch möglich, einen Titel mit einer Länge

---

[33] In der Definition einer Komponente können ausgewählte Props als *required* festgelegt werden, womit sie bei der Erzeugung der Komponente zwingend vorhanden sein müssen.

[34] Siehe http://react-toolbox.com [Stand 25.08.2017]

[35] Material Design: Von Google entwickelte Design-Richtlinie für moderne UIs von Apps

[36] SVG: Scalable Vector Graphics, Format zur Beschreibung zweidimensionaler Vektorgrafiken

von mehr als 30 Zeichen anzugeben, was jedoch nicht erwünscht ist. Der zugehörige Test sollte somit nicht bestehen. Ansonsten wird erwartet, dass alle Tests erfolgreich bestanden werden.

Die komplette Testfallspezifikation befindet sich in den Anlagen (Tabelle 4).

5.2.3 Implementierung mit Mocha

Anfangs mussten für die Implementierung der Tests folgende Module importiert werden (Listing 20):

```
import React from 'react';
import StpAppBar from '../src/stp/components/elements/StpAppBar';
import {mount} from 'enzyme';  // Full DOM Rendering
import {expect} from 'chai';
```

Listing 20: Importe für Testsuite 1 unter Mocha

Zum Testen der `StpAppBar` musste ein Full DOM Rendering verwendet werden. Dies hat den Hintergrund, dass auf Elemente zugegriffen werden muss, die bei einem Shallow Rendering nicht vorhanden wären, da sie nicht beim Rendern der ersten Ebene enthalten sind. Hierfür wird die importierte Funktion `mount()` von Enzyme benötigt, welche zuvor importiert wurde (siehe Listing 20).

Listing 21 zeigt die Anweisung, um die zu untersuchende Komponente, hier mit der Beschriftung `Phantastic Photobox`, zu rendern. Zurückgeliefert wird ein Wrapper-Objekt, das in einer Konstanten gespeichert wird.

```
const wrapper = mount(<StpAppBar title='Phantastic Photobox' />);
```

Listing 21: Full DOM Rendering der StpAppBar-Komponente

Für /TC020/ wurde darauf aufbauend folgender Test geschrieben (Listing 22):

```
it('should display the title given as a prop', () => {
  const wrapper = mount(<StpAppBar title='Phantastic Photobox' />);
  expect(wrapper.find('header').text()).
  to.equal('Phantastic Photobox');
});
```

Listing 22: Implementierung von /TC020/ als Test mit Mocha

Dabei wird zunächst mit `find()` das inkludierte HTML-Element vom Typ `header` ausgewählt. Die `text()`-Methode ermittelt dann den Inhalt dieses Elements. An dieser Stelle sollte der als Prop übergebene `title` stehen. Das erwartete Verhalten wird mit dem Matcher `to.eql()` geprüft, welcher eine Gleichheitsprüfung mit dem als Argument übergebenen Erwartungswert durchführt.

Für die Testfälle /TC010/ und /TC030/ musste lediglich der `title` beim Rendern der Komponente, sowie der Erwartungswert laut den Vorgaben der Testfallspezifikation angepasst werden.

Bei den nächsten beiden Tests musste eine andere Assertion geschrieben. Für den Fall, dass das `backicon`-Prop vorhanden ist und den Wert `true` besitzt, wird erwartet, dass ein SVG-Element, das „Zurück"-Icon, gerendert wird. Dies kann mit der `exists()`-Methode geprüft werden, welche auf einen (vorhandenen oder auch nicht vorhandenen) DOM-Knoten angewandt wird. Dieser muss zuvor wieder über den `find()`-Selektor ausgewählt werden, wobei in dem Fall ein Element vom Typ `svg` gesucht wird. Die `exists()`-Methode liefert dann einen booleschen Wert zurück, der nur dann dem Wert `true` entspricht, wenn mindestens ein Element dieser Art im gerenderten DOM vorhanden ist. Listing 23 zeigt die Implementierung des Tests für /TC050/, bei dem die Komponente mit Icon gerendert wird.

```
it('should display the return icon if backicon prop is given',() => {
  const wrapper = mount(<StpAppBar title='Phantastic Photobox'
    backicon={true} />);
  expect(wrapper.find('svg').exists()).to.equal(true);
});
```

Listing 23: Implementierung von /TC050/ als mit Mocha

Für /TC040/ musste derselbe Test wieder entsprechend angepasst Wert, also das `backicon`-Prop entfernt und der Erwartungswert auf `false` geändert werden.

Der letzte TC der Testsuite ließ sich wieder analog zu /TC020/ implementieren, wobei die Komponente mit dem Icon gerendert werden musste.

5.2.4 Implementierung mit Jest

Für die Umsetzung mit Jest wurde sich bei dieser TS gegen die Verwendung der Snapshot Testing-Funktionalität entschieden. Diese Gegenüberstellung folgt unter *5.3*. Zunächst sollte herausgefunden werden, ob sich Jest auch zum Testen von React-Komponenten auf konventionellem Wege eignet. Hierzu wurde auch bei Jest das Full DOM Rendering von Enzyme verwendet. Dies hat zur Folge, dass das Rendern der Komponente über die `mount()`-Funktion für jeden einzelnen Test identisch ist zur Implementierung mit Mocha. Auch der Selektor verwendet die Enzyme API und ist somit stets gleich, sodass sich die Test-Pendants ausschließlich durch den verwendeten Matcher unterscheiden. Da in dieser

Testsuite ausschließlich auf strikte Gleichheit[37] geprüft werden, muss der Matcher für jeden Test durch `toBe()` ersetzt werden. Dies entspricht dem Jest-Äquivalent zu dem unter 5.2.3 verwendeten `to.eql()`.

So ließ sich /TC050/ beispielsweise wie folgt umsetzen (Listing 24):

```
it('should display the return icon if backicon prop is given',() => {
  const wrapper = mount(<StpAppBar title='Phantastic Photobox'
    backicon={true} />);
  expect(wrapper.find('svg').exists()).toBe(true);
});
```

Listing 24: Implementierung von /TC050/ als Test mit Jest

5.2.5 Auswertung

Die Testsuite ließ sich mit beiden Test-Frameworks umsetzen. Alle Tests haben einen maximalen Automatisierungsgrad, d.h. es ist keine menschliche Verifizierung der Ergebnisse erforderlich. Bedingt durch die Verwendung von Enzyme als „Hilfswerkzeug“ für beide Test-Frameworks ist die Implementierung der Tests für gleiche Testfälle sehr ähnlich. Das einzige Unterscheidungskriterium ist der Framework- bzw. Assertion-Bibliothek-spezifische Matcher. Aus Sicht des Autors sind diese als gleichwertig zu betrachten, da die daraus entstehende Assertion in beiden Fällen dem angestrebten BDD-Stil entspricht und somit gut verständlich ist.

Zusammenfassend kann somit gesagt werden, dass sich beide Frameworks gleichermaßen für diese Art von Test eignen.

5.3 Testsuite 2

5.3.1 Testobjekte

In der zweiten Testgruppe sollen zwei Testobjekte geprüft werden, zunächst isoliert und anschließend im Zusammenspiel als kleiner „Integrationstest“, wenngleich es sich auch hierbei tatsächlich nur um einen Komponententest handelt.

[37] Dies entspricht einem Vergleich mit dem Vergleichsoperator ===

Das erste TO ist eine React-Komponente mit dem Namen `ButtonCmp`. Hierbei handelt es sich, wie der Name bereits suggeriert, um einen Button, welcher in mehreren Views verwendet wird und somit eine wichtige Komponente der Second Screen App darstellt.

Wie bereits die Komponente aus /TS01/ verwendet auch der die `ButtonCmp`-Komponente einen vorgefertigten Button von React Toolbox. Der Button verfügt somit z.B. über eine Animationen, im Englischen als „Ripple Effect“ bezeichnet, wenn er gedrückt wird; durch die Verwendung der Fremdkomponente musste dieses Verhalten nicht erst aufwendig von Hand programmiert werden. Intern rendert der Button von React Toolbox jedoch auch einen gewöhnlichen HTML-`<button>`, nur dass dieser bereits komplett gestylt ist. Der folgende Code (Listing 25) zeigt die vollständige Implementierung von `ButtonCmp`:

```
import React from 'react';
import {Button} from 'react-toolbox/lib/button';

class ButtonCmp extends React.Component {

  render() {
    if(this.props.label === false) return null;
    return (<Button className={this.props.className}
    onClick={this.props.onClick} theme={this.props.theme}
    label={this.props.label} raised={this.props.raised}
    primary={this.props.primary} mini={this.props.mini}
    type={this.props.type} />
    );
  }
}
export default ButtonCmp;
```

Listing 25: Implementierung der Komponente ButtonCmp

In der ersten Zeile innerhalb der `render()`-Funktion wird der Wert des Props `label` geprüft. Hierbei handelt es sich um die Zeichenkette, welche als Aufschrift auf dem Button angezeigt wird. Wird `ButtonCmp` mit dem Wert `false` für das `label` instanziiert, soll der Button gar nicht erst gerendert werden. Dies hat folgenden Grund: Standardmäßig werden z.B. auf dem `ContactView` zwei Button-Komponenten gerendert. Je nach Wunsch des Kunden soll die App jedoch auch so zugeschnitten werden können, dass an dieser Stelle nur ein Button angezeigt wird. Diese Konfiguration erfolgt über Redux: Der Reducer `config` verwaltet in seinem State sämtliche anpassbare Daten der App. Dazu zählen sämtliche Texte, aber auch Farben oder die Reihenfolge, in welcher die einzelnen Views nacheinander angezeigt werden. Listing 26 zeigt den Initial-State des `config`-Reducers ausschnittsweise.

```
const initialState = {
  primaryColor: 'rgb(104,178,42)',
  // weitere Farben und Schriftart

  // strings beinhaltet alle konfigurierbaren Texte
  strings: {
    // Alle Strings des contactViews
    contactView: {
      title:'Dürfen wir...',
      subheader:'... Dich kontaktieren?',
      buttonLabelYes:'Na klar. Meldet Euch!',
      buttonLabelNo:'Weiter ohne Daten',
      // weitere Texte
    },
    // analog für alle weiteren Views
  },
  followups: {
    // beinhaltet Reihenfolge der Views
  }
}
```

Listing 26: Initial-State des Config-Reducers (Auszug)

Die beiden Buttons des „ContactViews“ haben z.B. als Label stets den aktuellen Wert von `buttonLabelYes` bzw. `buttonLabelNo` (innerhalb von `strings.contactView`). Um nun nur noch einen der beiden Buttons anzuzeigen, genügt es, z.B. den Wert von `strings.contactView.buttonLabelNo` auf `false` zu ändern. Dies geschieht über eine Action vom Typ `SET_CONFIG`, welche als Payload ein JS-Objekt mit den zu ändernden Eigenschaften übergeben bekommt. Um den zweiten Button verschwinden zu lassen, müsste das Payload wie folgt aussehen: `{strings: {contactView: {buttonLabelNo : false }}}`. Für die Aktualisierung des `config`-States ist anschließend der entsprechende Reducer zuständig. Der `config`-Reducer stellt das zweite Testobjekt dieser Testgruppe dar.

5.3.2 Testfälle

Um den Umfang einzugrenzen, soll bei der `ButtonCmp`-Komponente lediglich getestet werden, ob diese sich je nach übergebenem Wert für das Prop `label` richtig verhält. Das heißt, es soll gewährleistet werden, dass…

- der Button mit der richtigen Aufschrift gerendert wird, wenn er einen beliebigen String als `label` übergeben bekommt (/TC070/), und
- der Button nicht gerendert wird, wenn `label` den Wert `false` hat (/TC080/).

Beim Testen des Reducers soll geprüft werden, ob...

- der Initial-State zurückgeliefert wird, wenn er durch keine Action modifiziert wurde (/TC090/), und
- der State wie erwartet aktualisiert wird, wenn der Reducer eine Action vom Typ `SET_CONFIG` als Argument übergeben bekommt (/TC100/).

Da der Zugriff auf den realen State des `config`-Reducers nicht gewünscht ist, soll dieser durch ein Mock-Objekt imitiert werden, welches 1:1 dem tatsächlichen State entspricht.

Anschließend soll die Integration der beiden Testkomponenten geprüft werden. `ButtonCmp` soll hierzu mit dem Wert `config.strings.contactView.buttonLabelNo`, wobei `config` dem Mock-Objekt mit dem State entspricht, als Label erzeugt werden. Die konkreten Testfälle sind in der Testfallspezifikation definiert (siehe Anlagen, Tabelle 5).

5.3.3 Implementierung mit Mocha

Für die Umsetzung der Tests der zweiten Testgruppe mussten zunächst folgende Module importiert werden (Listing 27):

```
import React from 'react';
import {expect} from 'chai';
import {shallow} from 'enzyme';
// Testobjekte
import ButtonCmp from '../src/stp/components/elements/ButtonCmp';
import config from '../src/stp/reducers/config';
import * as types from '../src/stp/constants/ActionTypes';
// Mock-Objekt – entspricht Initial-State des config-Reducers
import {initialState} from './constants/ButtonCmp'
```

Listing 27: Importe für Testsuite 2 unter Mocha

Da es für das Testen der Button-Komponente nicht erforderlich ist, mehrere Ebenen tief zu rendern, sollen die entsprechenden Tests wieder mittels Shallow Rendering umgesetzt werden. Der erste Testfall (/TC070/) konnte wie folgt als Test umgesetzt werden (Listing 28):

```
const wrapper = shallow(<ButtonCmp label="Button label"
    type="button" />);
    expect(wrapper.props().label).to.equal("Button label");
});
```

Listing 28: Umsetzung von /TC070/ als Test mit Mocha

Der Wrapper liefert in diesem Fall die darunterliegende React Toolbox Button-Komponente zurück. Mit `props().label` wird der Wert des Props `label` selektiert. Der Matcher `to.equal` führt eine strenge Gleichheitsprüfung (`===`) durch und ermittelt so, ob der tatsächliche Wert derselbe ist, welcher `ButtonCmp` als `label` übergeben wurde. Dieser Test ist ausreichend, da darauf vertraut werden kann, dass der React Toolbox Button den richtigen HTML-Output mit dem Wert für `label` als Button-Aufschrift erzeugt.

Für den zweiten TC (/TC080/) musste ein komplett anderer Test geschrieben werden, da das erwartete Verhalten ist, dass gar nichts gerendert wird. Hier erwies sich die Methode `type()` als nützlich, die, angewandt auf den Wrapper, den Typ des darunterliegenden Knotens zurückliefert (z.B. `div` bei einem `<div>`-Element). In diesem Fall wird erwartet, dass der Typ `null` entspricht. Der entsprechende Test sieht wie folgt aus (Listing 29):

```
it('should not render anything', () => {
    const wrapper = shallow(<ButtonCmp label={false}
    type="button" />);
    expect(wrapper.type()).to.equal(null);
});
```

Listing 29: Umsetzung von /TC080/ als Test mit Mocha

Das Testen der Reducer-Funktion ist ziemlich unkompliziert, da diese immer ein Objekt zurückliefert. Bekommt die Funktion keine bekannte Action übergeben (hier: ein leeres Objekt), soll einfach der Initial-State zurückgeliefert werden, welcher als Mock-Objekt in die Test-Datei eingebunden wurde. Der geschriebene Test für /TC090/ ist in Listing 30 zu sehen.

```
it('should return the initial state', () => {
    expect(config(undefined, {})).to.eql(initialState);
});
```

Listing 30: Umsetzung von /TC090/ als Test mit Mocha

Zu beachten ist, dass diesmal der Matcher `to.eql` zu verwenden ist (im Gegensatz zu `to.equal`). Dieser führt einen Vergleich zweier Objekte in die Tiefe durch, prüft also, ob jede Eigenschaft aus dem einen Objekt in dem anderen vorhanden ist und auch den gleichen Wert hat.

Beim zweiten Reducer-Test wird erwartet, dass der ursprüngliche State als Ergebnis leicht modifiziert wird. Um dieses Verhalten zu prüfen, wurde zunächst eine Kopie des Initial-States erstellt (siehe Listing 31).

```
const modifiedState = JSON.parse(JSON.stringify(initialState));
```

Listing 31: Erstellen einer Kopie des Mock-Objekts mit dem Initial-State

Diese Anweisung erstellt ein Duplikat von `initialState`, erzeugt dabei jedoch keine Referenz, d.h. das neue Objekt ist von dem Original komplett unabhängig. Im Anschluss wurde die State-Kopie wie folgt modifiziert (Listing 32):

```
modifiedState.strings.contactView.buttonLabelYes = "Ja, bitte.";
modifiedState.strings.contactView.buttonLabelNo = "Nein, danke.";
```

Listing 32: Anpassungen an der duplizierten Version des Initial-States

Das Objekt `modifiedState` ist somit identisch mit dem State, der nach Aufruf der Reducer-Funktion von /TC100/ erwartet wird. Die Action, die der Reducer als Argument übergeben bekommen soll, wurde außerhalb der eigentlichen Tests definiert, da diese auch für die folgenden Assertions benötigt wird und somit lediglich einmal geschrieben werden musste (siehe Listing 33).

```
const payload = {strings: {contactView: {buttonLabelYes:'Ja, bitte.',
buttonLabelNo:'Nein, danke.'}}};
const expectedAction = {
    type: types.SET_CONFIG,
    payload
}
```

Listing 33: Erstellen der erwarteten Action für /TC100/

Die Assertion ist dann wieder ein simpler Vergleich zweier Objekte (Listing 34):

```
const newState = config(initialState, expectedAction);
expect(newState).to.be.an('object').and.eql(modifiedState);
```

Listing 34: Die Assertion für /TC100/ mit Mocha

Hier wurde zusätzlich noch eine Prüfung des Typs eingebaut. Tatsächlich ist dieser erste Teil des Matchers redundant (die Gleichheitsprüfung würde fehlschlagen, wenn nicht zwei Objekte miteinander verglichen würden), wurde hier jedoch eingebaut, um eine der Besonderheiten von Chai auszuprobieren, nämlich, dass mehrere Matcher miteinander verkettet werden können – hier durch das Schlüsselwort `and` , was einer logischen UND-Verknüpfung entspricht.

Die letzten drei Tests dieser Testsuite sind dann einfach eine Kombination der vorherigen Tests. Listing 35 zeigt stellvertretend die Implementierung des Tests für /TC120/.

```
it('should render the button with new label if global state is modi-
fied', () => {
    /* Ändere zwei Eigenschaften des config-States durch config-
    Reducer */
    const configState = config(initialState, expectedAction);
    const wrapper = shallow(<ButtonCmp
        label={configState.strings.contactView.buttonLabelNo}
        type="button" />);
    // Erwartet: Label hat den Wert 'Nein, danke.'
    expect(wrapper.props().label).to.equal("Nein, danke.");
});
```

Listing 35: Implementierung von /TC120/ unter Mocha

5.3.4 Implementierung mit Jest

Die Tests dieser Testsuite boten sich an, komplett mittels Snapshot Testing implementiert zu werden. Hierfür muss zusätzlich folgendes Modul importiert werden (Listing 36):

```
import renderer from 'react-test-renderer';
```

Listing 36: Importieren des React-Test-Renderers

Die Methode `renderer.create(<Component {...props} />)` rendert die Komponente mit den angegebenen Props als reines JavaScript-Objekt[38] und die `toJSON()`-Methode erzeugt daraus ein für den Menschen besser lesbares Format. Die Ausgabe kann in eine Variable gespeichert (laut Konvention mit der Bezeichnung `tree`) und dann mit dem zugehörigen Snapshot verglichen werden. Dies geschieht mit dem Matcher `toMatchSnapshot()`. Für den ersten TC sieht der zugehörige Test somit wie folgt aus (Listing 37):

```
it('should have the label given as a prop', () => {
    const tree = renderer.create(
        <ButtonCmp label="Button label" type="button" />
    ).toJSON();
    expect(tree).toMatchSnapshot();
});
```

Listing 37: Umsetzung von /TC070/ als Snapshot Test mit Jest

[38] Siehe https://www.npmjs.com/package/react-test-renderer [Stand 27.08.2017]

Beim erstmaligen Ausführen des Tests wird innerhalb des `__tests__`-Verzeichnisses ein neues Verzeichnis mit der Bezeichnung `__snapshots__` angelegt, sowie eine Snapshot-Datei. Für den Test aus Listing 37 wurde der in Abbildung 15 dargestellte Snapshot erzeugt.

```
exports[`Test Suite 2 Button should have the label given as a prop 1`] = `
<button
  className=""
  data-react-toolbox="button"
  disabled={false}
  href={undefined}
  onClick={undefined}
  onMouseDown={[Function]}
  onMouseLeave={[Function]}
  onMouseUp={[Function]}
  onTouchStart={[Function]}
  type="button"
>
  Button label
</button>
`;
```

Abbildung 14: Snapshot für Test der Button-Komponente

Standardmäßig gilt der Test beim ersten Ausführen automatisch als bestanden. Ob der Test tatsächlich richtig ist, musste anschließend jedoch für jeden Test manuell überprüft werden. Für den ersten Test liefert der Snapshot einen `<button>` mit dem Inhalt `Button label` zurück (siehe Abbildung 15, rote Markierung), was genau dem gewünschten Ergebnis entspricht.

Bei allen weiteren React-Tests wurde nach dem gleichen Prinzip vorgegangen. Die einzige „Schwierigkeit“ beim Schreiben der Tests besteht darin, die Komponente mit den richtigen Props zu rendern, wobei diese Überlegung bei „klassischen“ Tests genauso stattfinden muss. Ansonsten konnte einfach die Assertion `expect(tree).toMatchSnapshot()` für alle Tests übernommen werden, womit automatisch für jeden Test ein neuer Snapshot in der zugehörigen Datei hinzugefügt wurde.

Auch das Testen von Redux-Code (Reducer-Funktion) ließ sich als Snapshot Test umsetzen, wobei der Matcher `toMatchSnapshot()` hierbei einfach auf das zurückgelieferte JS-Objekt angewendet wird. Listing 38 zeigt beispielhaft den geschriebenen Code für /TC100/.

```
it('should update the state correctly if config is changed', () => {
    const newState = config(initialState, expectedAction);
    expect(newState).toMatchSnapshot();
});
```

Listing 38: Testen einer Reducer-Funktion mit einem Snapshot Test

Der erzeugte Snapshot entspricht dann einfach dem gesamten Objekt (`newState`), wobei anschließend wieder manuell überprüft werden muss, ob der State wie erwartet aktualisiert wurde.

5.3.5 Auswertung

In dieser Testsuite konnte das Snapshot Testing Feature von Jest ausgiebig auf den Prüfstand gestellt werden. Alle Tests waren mit beiden Test-Frameworks umsetzbar. Beim Testen von React ließen sich die Snapshot Tests vergleichsweise einfacher implementieren, da sich als Entwickler keine Gedanken darüber gemacht werden musste, eine passende Assertion für jeden Test zu formulieren. Im Gegenzug wirkte sich dies aber negativ auf den Automatisierungsgrad aus, da für jeden Snapshot eine manuelle Verifizierung erforderlich war. Es musste sich also im Vorfeld trotzdem überlegt werden, welche Ausgabe für jeden Snapshot Test erwartet wird. Bei dieser Testsuite überwiegte der Nutzen des Snapshot Testings den Extraaufwand, da jeder Snapshot so übersichtlich war, dass sofort erkannt werden konnte, ob die Ausgabe dem gewünschten Ergebnis entsprach.

Die Testsuite /TS04/ (siehe Anlagen, Quellcode) konnte dieses Ergebnis bestätigen: Auch hier ließ sich die ausgewählte Komponente gut mit Snapshot Testing prüfen. In beiden Fällen wurden Komponenten mit einer eher einfachen Struktur (keine vielfach verschachtelten HTML-Elemente oder Kind-Komponenten) getestet. Es kann somit geschlussfolgert werden, dass das Snapshot Testing Feature von Jest für diese Art von React-Komponenten ein durchaus brauchbares Werkzeug zum Testen ist und die Implementierung der Tests so sehr vereinfacht und beschleunigt, dass selbst die geringere Automatisierung zu vernachlässigen ist. Aus diesem Grund ist Jest in dieser Kategorie Mocha vorzuziehen.

Davon abzugrenzen ist das Testen von Redux. Im konkreten Fall wurde eine Reducer-Funktion getestet, welche einen langen State zurückliefert. Um ganz sicher zu gehen, dass dieser dem erwarteten Zustand entspricht, müsste eigentlich für jede Eigenschaft geprüft werden, ob diese vorhanden ist und deren Wert mit dem erwarteten Wert übereinstimmt. Dies ist wenig effizient und erhöht die Wahrscheinlichkeit, dass ein potenzieller Fehler gar nicht erst entdeckt wird. Aus diesem Grund ist für Reducer-Funktionen eine Implementierung mit einer klassischen Assertion wie unter Mocha vorzuziehen.

5.4 Testsuite 3

5.4.1 Testobjekte

Gegenstand der dritten Testsuite sollen der `CorporateView` und die `VisibleView`-Komponente sein. Der `CorporateView` ist standardmäßig der vorletzte View, welcher dem Nutzer die Möglichkeit gibt, die AGBs (Allgemeine Geschäftsbedingungen) des Unternehmens zu lesen. Um das selektierte Bild im nächsten Schritt zu drucken und alle eingegebenen Daten

an das Backend zur Weiterverarbeitung zu senden, müssen die AGBs zwingend akzeptiert werden. Dies geschieht über eine Checkbox (dt.: Kontrollkästchen), welche standardmäßig abgewählt, d.h. nicht abgehakt, ist. Durch einen Klick auf die Checkbox, kann diese markiert, also „abgehakt“, werden. Nur dann, wenn die AGB-Checkbox markiert ist, der Nutzer also ausdrücklich den AGBs zustimmt, soll es möglich sein, durch Klick auf den Button zum nächsten View zu gelangen und dort sein Bild zu drucken.

Abbildung 16 zeigt den `CorporateView`. Die AGB-Checkbox und der Button sind rot umrahmt.

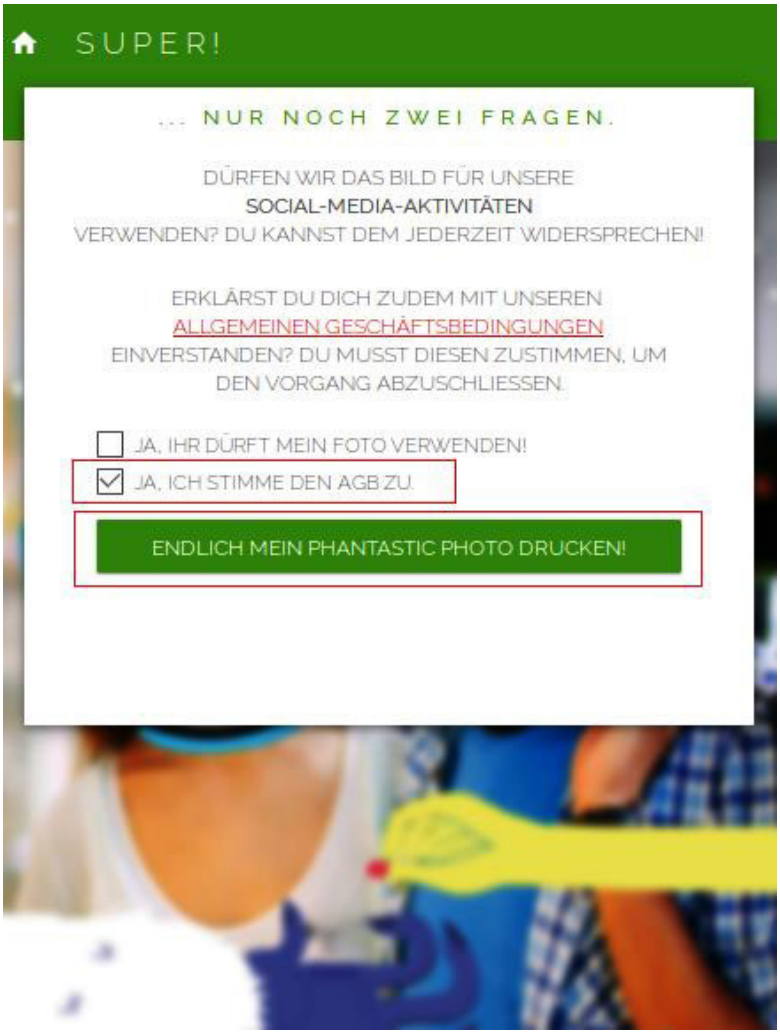

Abbildung 15: Der “CorporateView” der Second Screen App

Die Checkbox ist selbst eine React-Komponente, welche in ihrer `render()`-Funktion wieder eine `Checkbox`-Komponente von React Toolbox zurückliefert. Diese wiederum rendert ein HTML-Element vom Typ `<input>`. Die Checkbox verfügt über ein Prop `checked`, welches die Werte `true` (markiert) oder `false` (abgewählt) annehmen kann. Auf dem `CorporateView` hat die AGB-Checkbox als Wert hierfür `this.state.acceptTerms`. Der Wert entspricht also der Eigenschaft `acceptTerms` des States des CorporateViews, der logischer-

weise auch nur diese beiden booleschen Werte annehmen kann. Beim erstmaligen Rendern des Views hat `this.state.acceptTerms` standardmäßig den Wert `false`. Beim Klicken auf die Checkbox wird eine Funktion aufgerufen, welche den entsprechenden Wert im State umkehrt, was zur Folge hat, dass sich automatisch auch die Checkbox-Komponente aktualisiert.

Akzeptiert der User die AGBs und klickt anschließend auf den Button, soll die Eigenschaft `acceptTerms` mit dem Wert `true` zu den Kundendaten hinzugefügt werden. Des Weiteren soll der nächste View gerendert werden. Hierzu existiert eine Action vom Typ `SHOW_NEXT_DIALOG`.

5.4.2 Testfälle

Bei dieser Testsuite soll der Schwerpunkt auf der simulierten Nutzerinteraktion liegen. Die ersten drei Testfälle sollen prüfen, ob die Eigenschaft `acceptTerms` des States den richtigen Wert hat, wenn:

- der View gerendert wurde und noch keine Nutzerinteraktion stattgefunden hat (/TC140/),
- der Nutzer einmal auf die AGB-Checkbox geklickt hat (/TC150/), und
- der Nutzer zwei Mal auf die AGB-Checkbox geklickt hat (/TC160/).

Anschließend soll geprüft werden, ob die Checkbox und `this.state.acceptTerms` „synchron“ sind: Bei keiner Nutzerinteraktion wird erwartet, dass die Checkbox abgewählt ist (/TC170/); nach einem Klick auf die Checkbox soll diese hingegen markiert sein (/TC180/). Werden diese ersten fünf Tests ohne Fehler durchlaufen, kann angenommen werden, dass sich die Checkbox und `this.state.acceptTerms` auch für jede beliebige andere Menge an Klicks erwartungsgemäß verhalten.

Zum Abschluss soll noch geprüft werden, ob:

- weder die Nutzerdaten hinzugefügt noch der nächste View ausgewählt werden, wenn der Button angeklickt wird, ohne dass die AGB-Checkbox zuvor markiert wurde (/TC190/), und
- die Kundendaten (`acceptTerms:true`) zum Store hinzugefügt werden (/TC200/) und der nächste View ausgewählt wird (/TC210/), wenn die AGBs akzeptiert wurden und der Nutzer auf den Button drückt.

Die Testfallspezifikation befindet sich in den Anlagen (Tabelle 6).

5.4.3 Implementierung mit Mocha

Für die Implementierung der dritten Testsuite mussten folgende Importe eingefügt werden (Listing 39):

```
import React from 'react';
import {Provider} from 'react-redux';
import {mount} from 'enzyme';   // für Full DOM-Rendering
import {expect} from 'chai';
// Die Testobjekte
import CorporateView from '../src/stp/components/views/CorporateView';
import VisibleView from '../src/stp/containers/VisibleView';
// Mock-Objekte
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';    // Middleware
import {globalState} from './constants/CorporateView'
```

Listing 39: Importe für Testsuite 3 unter Mocha

Das `globalState`-Objekt imitiert dabei den globalen State, welcher für alle Tests dieser Suite zur Verfügung stehen muss.

Für die ersten fünf Tests, welche lediglich den `CorporateView` als separate Komponente ohne Anbindung an den Store benötigen, muss der View mit der bereits unter /TS01/ verwendeten `mount()`-Funktion gerendert werden. Dies geschieht mit folgender Anweisung (Listing 40):

```
const wrapper = mount(<CorporateView {...globalState} />);
```

Listing 40: Full DOM Rendering der CorporateView-Komponente

Der Zusatz `{...props}` ist eine ES2015-Syntax, die als *Spread-Operator* bezeichnet wird. Hierüber werden alle Eigenschaften des importierten JS-Objekts der View-Komponente als Prop zur Verfügung gestellt. In dem Fall hat die gerenderte Komponente damit auf den kompletten globalen State als Props Zugriff, was eine Anbindung an den Store imitiert.

Um, wie für die ersten drei Tests erforderlich, auf den State der Komponente zuzugreifen, kann die Methode `state()` verwendet werden, welche ein Objekt mit allen Eigenschaften des States zurückliefert. Über den Punktoperator kann dann wie gewohnt auf eine bestimmte Eigenschaft des Objektes zugegriffen werden. Der selektierte Wert kann dann über den bereits bekannten Gleichheits-Matcher mit dem Soll-Wert verglichen werden. Der erste Test der Suite (/TC140/) ließ sich somit wie folgt umsetzen (Listing 41):

```
it('should be false by default', () => {
    const wrapper = mount(<CorporateView {...globalState} />);
    expect(wrapper.state().acceptTerms).to.equal(false);
});
```

Listing 41: Implementierung von /TC140/ mit Mocha

Für den zweiten Test war zusätzlich die Simulation einer Nutzerinteraktion erforderlich. Hierzu musste zunächst das HTML-Element selektiert werden, auf das die Aktion später angewendet werden soll. Dies ließ sich mit der `find()`-Methode realisieren, welche ein Array aller Knoten zurückliefert, die mit dem als Argument übergebenen Selektor übereinstimmen. Hat dieses eine Länge von > 1, d.h. existieren mehrere DOM-Knoten, die mit der Auswahl übereinstimmen, kann mittels `at()` ein bestimmter ausgewählt werden. Dies geschieht, indem der Methode der Index des gewünschten Elements in Klammern übergeben wird. Auf dem CorporateView werden standardmäßig zwei Checkboxen angezeigt (siehe Abbildung 16), wobei für den Test explizit die Zweite von Interesse ist. Dies entspricht der Checkbox zur Bestätigung der AGBs. Folgender Selektor wurde für /TC210/ geschrieben (Listing 42):

```
wrapper.find('Checkbox').at(1).childAt(0)
```

Listing 42: Selektor zur Auswahl der Checkbox

Dabei selektiert `wrapper.find('Checkbox').at(1)` die zweite Checkbox, und `childAt(0)` davon wiederum das erste Kind-Element. Hiermit wird das eigentliche HTML-Element vom Typ `<input>` ausgewählt. Um auf dieses ein Klick-Ereignis (`'click'`) zu simulieren, wird einfach die Methode `simulate()`, welche ebenfalls Teil der Enzyme API für Full DOM Rendering ist, mit dem gewünschten Ereignis als Argument auf das selektierte Element angewendet. Listing 43 zeigt die komplette Implementierung des Tests für /TC150/.

```
it('should be true after click on Checkbox', () => {
    const wrapper = mount(<CorporateView {...globalState} />);
    // Simuliere Klick auf Checkbox (zuvor unchecked)
    wrapper.find('Checkbox').at(1).childAt(0).simulate('click');
    expect(wrapper.state().acceptTerms).to.equal(true);
});
```

Listing 43: Implementierung von /TC150/ mit Mocha

Für /TC160/ musste die Klick-Simulation einfach zweimal hintereinander ausgeführt werden und der Erwartungswert auf `false` geändert werden.

Die Testfälle /TC170/ und /TC180/ erforderten den Zugriff auf das Attribut `checked` des HTML-Elements `<input>`, da sich an dessen Wert erkennen lässt, ob die Checkbox markiert ist oder nicht. Hierzu muss das entsprechende Element wie unter Listing 43 ausgewählt werden. Über die Methode `prop()` kann dann der Wert des als Argument übergebenen Attributs ermittelt werden.

```
describe('Checkbox', () => {
    it('should be unchecked by default', () => {
        const wrapper = mount(<CorporateView {...globalState} />);
        expect(wrapper.find('Checkbox').at(1).
        childAt(0).prop('checked')).to.equal(false);
    });
});
```

Listing 44: Implementierung von /TC170/ mit Mocha

Listing 44 zeigt die Implementierung des Tests für /TC170/. Bei /TC180/ musste zusätzlich noch ein Klick auf die Checkbox simuliert werden.

Für die letzten drei Tests war es erforderlich, einen kompletten Store zu imitieren. Dieser sollte denselben globalen State wie die vorherigen Tests beinhalten. Da der echte Store der Applikation nicht für Tests verwendet werden soll, musste wieder auf ein Mock-Objekt zurückgegriffen werden. Hierzu dient die Funktion `configureStore()` aus dem Package `redux-mock-store`, welches unter 4.1.1 installiert wurde und über die Importe in die Test-Datei eingebunden wurde. Diese ermöglicht die Erstellung eines Mock-Stores, der sich ähnlich wie der echte Store verhält, jedoch seinen State nie aktualisiert. Das heißt, dass beim Versenden einer Action die Reducer-Funktion nicht wirklich aufgerufen wird, der Store aber trotzdem registriert, dass eine Action verschickt wurde.

Der Mock-Store musste zunächst für jeden der folgenden drei Tests wie folgt erzeugt werden (Listing 45):

```
// außerhalb der äußersten describe()-Funktion (einmalig)
const middlewares = [thunk];
const mockStore = configureStore(middlewares);

// innerhalb der Tests
const store = mockStore(globalState)
```

Listing 45: Erstellung eines Mock-Stores

Die `configureStore()`-Funktion bekommt zunächst ein Array mit Middlewares[39] übergeben. In dem Fall wird die Middleware *Thunk* eingebunden, welche auch von dem realen Store benötigt wird. Die Container-Komponente `VisibleView` muss dann über einen Provider mit dem Store verbunden werden. Listing 46 zeigt die Implementierung der `mount()`-Funktion für die letzten drei Tests:

```
const wrapper = mount(<Provider store={store}>
    <VisibleView />
</Provider>);
```

Listing 46: Full DOM Rendering für eine Container-Komponente

`VisibleView` rendert dann automatisch den View, welcher im globalen State des Stores unter `config.dialog.next` angegeben ist: Standardmäßig ist dies beim ersten Mal der `StartView`, für diesen Test wurde `globalState` (das importierte Mock-Objekt) jedoch so modifiziert, dass der `CorporateView` dargestellt wird. Über `store.getActions()` konnten nun alle versendeten Actions ermittelt werden. Wird diese Methode direkt nach dem Rendern der Komponente (siehe Listing 46) ausgeführt und sich die Ausgabe mit dem Befehl console.log() über die Konsole angezeigt, wird ein Array mit genau einer Action vom Typ `SERVER/GET_CONFIG` angezeigt (siehe Abbildung 17). Hierbei handelt es sich um eine Action, die automatisch nach dem Rendern eines jeden Views an den Server gesendet wird, was erklärt, weshalb hier bereits ein Eintrag vorhanden ist.

```
[ { type: 'SERVER/GET_CONFIG' } ]
```

Abbildung 16: Ausgabe der Actions über die Konsole

Hat der Nutzer die AGB-Checkbox aktiviert und klickt anschließend auf den „Weiter“-Button, wird erwartet, dass zwei weitere Actions hinzugefügt werden. Für die Testfälle /TC200/ und /TC210/ mussten die beiden Ereignisse (Klick auf Checkbox und anschließender Klick auf den Button) zunächst wieder simuliert werden. Um zu überprüfen, ob die beiden erwarteten Actions anschließend in dem Array vorhanden sind, wurde dieses mit der `forEach()`-Methode durchlaufen. Im Fall von /TC200/ wurde hier z.B. geprüft, ob in dem Array eine Action vom Typ `ADD_CUSTOMER_DATA` vorhanden ist und der zugehörige Payload die Eigenschaft `acceptTerms` besitzt. Der Wert dieser Eigenschaft wurde dann in der Assertion mit dem erwarteten Wert verglichen, in dem Fall `true`. Listing 47 zeigt die komplette Implementierung des zugehörigen Tests für /TC200/:

[39] Eine Middleware ist bei Redux eine zusätzliche Instanz, die zwischen dem Versenden einer Action und dem Erreichen des Reducers in Aktion tritt.

```
it('should add customer data on button click if checkbox is checked',
() => {
    const store = mockStore(globalState);
    const wrapper = mount(<Provider store={store}>
        <VisibleView />
    </Provider>);
    // Simuliere Klick auf Checkbox
    wrapper.find('Checkbox').at(1).childAt(0).simulate('click');
    // Simuliere anschließend Klick auf Button
    wrapper.find('button').at(1).simulate('click');
    // Speichere Array mit versendeten Actions in Variable
    const arr = store.getActions();
    // Zweites Array zur Speicherung der Übereinstimmungen
    let actions = [];
    arr.forEach(function(item){
        if (item.type === 'ADD_CUSTOMER_DATA'){
            actions.push(item.payload.acceptTerms);
        }
    });
    expect(actions).to.include(true);
});
```

Listing 47: Implementierung von /TC200/ mit Mocha

Der Matcher `to.include()` prüft dabei, ob der als Argument übergebene Wert in dem jeweiligen Array enthalten ist.

Für die Testfälle /TC190/ und /TC210/ ist die Implementierung ähnlich, wobei die Funktion der `forEach()`-Methode und die Assertion jeweils angepasst werden musste bzw. im Falle von /TC190/ der Klick auf die Checkbox fehlt.

5.4.4 Implementierung mit Jest

Die Implementierung der ersten fünf Tests mit Jest war nahezu identisch mit der von Mocha. Der Versuch, diese Tests als Snapshot Test umzusetzen, scheiterte. Durch die ausgegebene Fehlermeldung konnte herausgefunden werden, dass hierfür die Methode `findDOMNode()` aus dem Package `react-dom` verantwortlich ist, welche im `CorporateView` mehrfach verwendet wird und offenbar nicht mit Jests Snapshot Testing kompatibel ist. Somit wurde für die Implementierung ebenfalls das Full DOM Rendering von Enzyme verwendet, womit die ersten fünf Tests mit Ausnahme der Framework-spezifischen Matcher identisch sind.

Für die Testfälle /TC190/ bis /TC210/ konnte das Snapshot Testing allerdings sinnvoll eingesetzt werden. Zwar musste auch hier wieder das Full DOM Rendering zum Rendern der Komponente genutzt werden, allerdings wurde anschließend einfach ein Snapshot des Arrays mit den versendeten Actions erstellt. Dieses muss somit nicht für jeden Test erst durchlaufen und untersucht werden, womit der benötigte Quellcode deutlich reduziert werden konnte. Der größte Vorteil dieser Methode ist, dass die Testfälle /TC200/ und /TC210/ gleich in einem einzelnen Test kombiniert werden konnten. Die Implementierung dieses Tests ist in Listing 48 dargestellt.

```
it('should add customer data and move to next dialog on button click
if checkbox is checked', () => {
    const store = mockStore(globalState);
    const wrapper = mount(<Provider store={store}>
        <VisibleView />
    </Provider>);
    // Simuliere Klick auf Checkbox
    wrapper.find('Checkbox').at(1).childAt(0).simulate('click');
    // Simuliere anschließend Klick auf Button
    wrapper.find('button').at(1).simulate('click');
    // store.getActions() liefert alle dispatchten Actions
    const actions = store.getActions();
    expect(actions).toMatchSnapshot();
});
```

Listing 48: Kombination aus Full DOM Rendering und Snapshot Test

Als Ergebnis wurde der folgende Snapshot generiert (siehe Listing 49):

```
exports[`Testsuite 3 ViewSelector should add customer data and move to
next dialog on button click if checkbox is checked 1`] = `
Array [
  Object {
    "type": "SERVER/GET_CONFIG",
  },
  Object {
    "payload": Object {
      "acceptTerms": true,
      "allowCorporateSharing": undefined,
    },
    "type": "ADD_CUSTOMER_DATA",
  },
  Object {
    "next": "GOODBYE_DIALOG",
    "recent": "CORPORATE_DIALOG",
    "type": "SHOW_NEXT_DIALOG",
  },
]
`;
```

Listing 49: Erzeugter Snapshot für /TC170/ und /TC180/

Es ist sofort erkennbar, dass das Array die beiden erwarteten Actions mit den richtigen Werten, die hier zum Zweck der Veranschaulichung rot hervorgehoben wurden, enthält. Die manuelle Verifizierung, die wie immer der größte Nachteil bei Snapshot Tests ist, beansprucht in diesem Fall also nicht viel Zeit.

5.4.5 Auswertung

Bei der dritten Testsuite zeigte sich, dass das Snapshot Testing Feature von Jest nicht nur zum Testen von React-Komponenten geeignet ist. Indem die Funktionalität auf ein Array angewandt wurde, konnte ein kompletter Test gespart werden und die Länge des Codes auf ein Dreiviertel dessen reduziert werden, was unter Mocha geschrieben werden musste. Bei der Implementierung der Testsuite 5 (siehe Anlagen, Quellcode) konnte dieses Ergebnis bestätigt werden, wo ebenfalls ein Array als Snapshot getestet wurde und somit mehrere Zeilen Code eingespart werden konnten. Es ist jedoch zu beachten, dass diese Methode nur dann sinnvoll ist, wenn das Array eine überschaubare Länge hat.

Gleichzeitig zeigte diese Testsuite die Grenzen der Snapshot Testing-Funktionalität auf, da es nicht möglich war, eine Komponente zu testen, bei welcher mittels `findDOMNode()` auf

das DOM zugegriffen wird, was bei der Second Screen App z.B. bei allen View-Komponenten der Fall ist. Bei der Implementierung von /TS05/ konnte dieses Verhalten bestätigt werden. Andererseits erscheint es ohnehin wenig nützlich, eine View-Komponente, die aus vielen Sub-Komponenten zusammengesetzt ist, auf diese Weise zu testen, da der Snapshot auf Grund der Menge des HTML-Codes schlecht überschaubar wäre.

Alles in allem rechtfertigen die dargestellten Ergebnisse die Aussage, dass die Implementierung der dritten Testsuite mit Jest deutlich effizienter war. Nur dann, wenn die gerenderte Komponente selbst – wie bei den ersten fünf Tests der Testsuite – getestet werden soll, nehmen sich beide Frameworks nicht viel.

5.5 Weitere Vergleichskriterien

5.5.1 Tauglichkeit für testgetriebene Entwicklung

Beide Test-Frameworks erlauben in Verbindung mit Enzyme, React-Komponenten nach dem TDD-Ansatz zu testen. Hierzu muss die zu testende Komponente vorerst zumindest insofern implementiert werden, dass diese ohne Fehler rendert, ohne jedoch die konkrete Struktur und Funktionalität zu spezifizieren. Die Komponente kann somit in die zugehörige Test-Datei eingebunden werden und wie eine bereits bestehende Komponente in den Tests verwendet werden, nur dass diese bis zur richtigen Implementierung alle fehlschlagen werden.

Davon abzugrenzen ist jedoch Jests Snapshot Testing, welches sich nicht für TDD eignet. Dies würde erfordern, dass der erwartete Snapshot im Vorfeld händisch erstellt wird, was theoretisch zwar möglich wäre, jedoch komplett der Idee des Snapshot Testings widerspricht, da dieser automatisch generiert werden sollte. Nicht zuletzt wäre es äußerst zeitaufwendig und fehleranfällig, sich für eine noch nicht bestehende Komponente zu überlegen, welches HTML diese erzeugen soll. Aus diesen Gründen ist von der Verwendung der Snapshot Testing-Funktionalität in Verbindung mit einem TDD-Ansatz abzuraten. Da bei beiden Test-Frameworks jedoch, wie bereits beschrieben, Enzyme hierfür verwendet werden kann, eignen sich beide gleichermaßen für die testgetriebene Entwicklung bei React-Komponenten.

5.5.2 Verhalten im Fehlerfall

Um das Verhalten der Test-Frameworks im Fehlerfall zu testen, wurden einige der Testobjekte so manipuliert, dass die zugehörigen Tests einen Fehler werfen. So wurde etwa im `CorporateView` der Anfangswert der Eigenschaft `acceptTerms` des States von `false` auf `true` geändert. Dies sollte zur Folge haben, dass kein einziger Test der Testsuite 3 besteht,

während alle sonstigen Tests davon nicht betroffen sind, was auch bei beiden Test-Frameworks der Fall war. Bei beiden Test-Frameworks wurden die fehlgeschlagenen Tests deutlich kenntlich gemacht – bei Mocha durch rote Schrift, bei Jest durch ein rotes „X“ vor der Beschreibung –, sodass sofort erkennbar war, welche Tests fehlschlugen (siehe Anlagen, Abbildungen 21 und 22). Zudem wurde bei beiden Test-Frameworks kenntlich gemacht, wodurch der Fehler für jeden Test verursacht wurde. Bei Jest ist dies sehr übersichtlich. Im konkreten Fall wurde folgende Fehlerausgabe geliefert (siehe Abbildung 18):

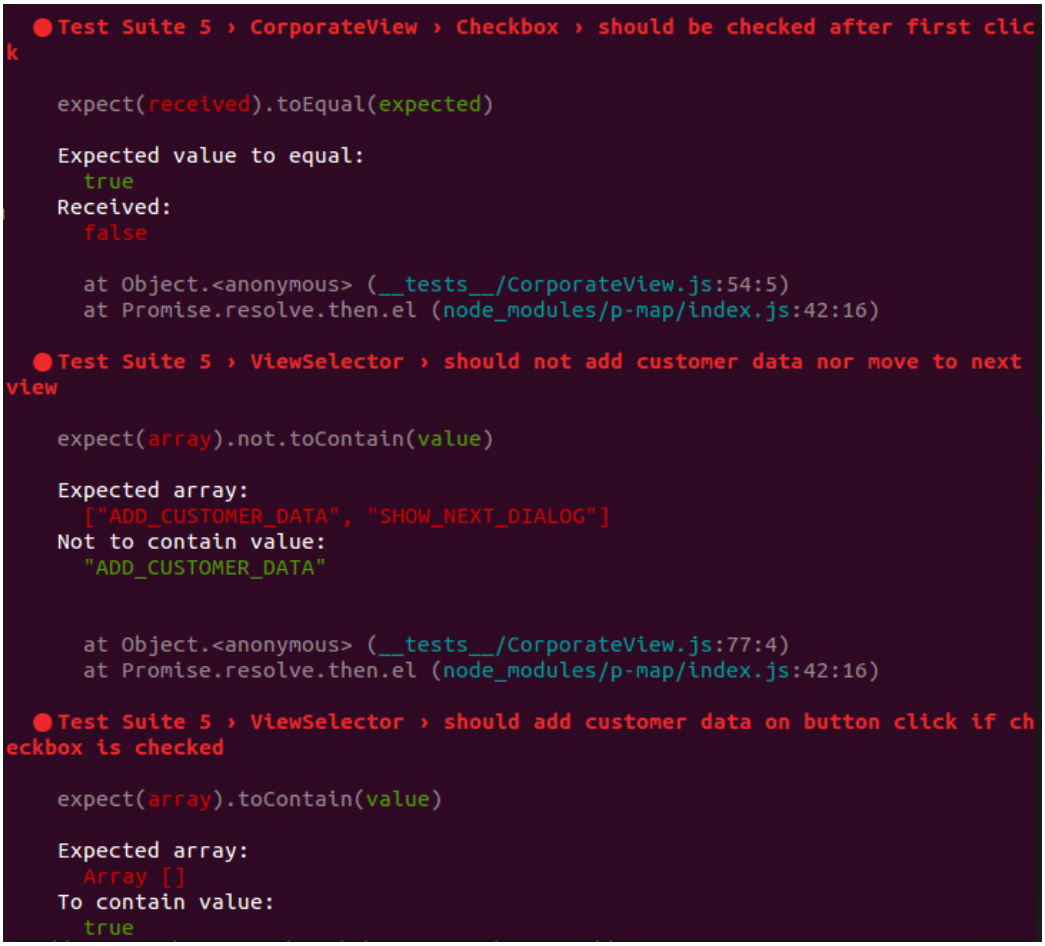

Abbildung 17: Ausgabe der gefundenen Fehler mit Jest (Ausschnitt)

Die Fehlerausgabe für jeden Fehler erfolgt nach einem übersichtlichen Muster. Zunächst wird mit roter Schrift der konkrete Test aufgelistet, bei dem der auftrat, darunter die Assertion, welche fehlgeschlagen ist. Anschließend werden der erhaltene Wert (rote Schrift) und erwartete Wert (grüne Schrift) untereinander aufgeführt, jeweils mit einer kurzen Überschrift, die Aufschluss darüber gibt, was den Fehler verursacht hat. Bei dem zweiten aufgeführten Test aus Abbildung 18 wird z.B. sofort klar, dass ein Array mit zwei Einträgen, darunter dem String `"ADD_CUSTOMER_DATA"`, erhalten wurde, obwohl dieser Wert laut der Assertion nicht vorhanden sein dürfte.

Im Vergleich dazu zeigt Abbildung 19 die Fehlerausgabe für dieselben Tests mit Mocha.

```
  5) Test Suite 5 CorporateView Checkbox should be checked after first click:

      AssertionError: expected false to equal true
      + expected - actual

      -false
      +true

      at Context.<anonymous> (test/CorporateView.js:50:36)

  6) Test Suite 5 CorporateView ViewSelector should not add customer data nor mo
ve to next view:
     AssertionError: SHOW_NEXT_DIALOG: expected [ Array(2) ] to not include 'ADD
_CUSTOMER_DATA'
      at Context.<anonymous> (test/CorporateView.js:70:28)

  7) Test Suite 5 CorporateView ViewSelector should add customer data on button
click if checkbox is checked:
     AssertionError: expected [] to include true
      at Context.<anonymous> (test/CorporateView.js:89:24)

  8) Test Suite 5 CorporateView ViewSelector should move to next dialog on butto
n click if checkbox is checked:
     AssertionError: expected [] to include 'CORPORATE_DIALOG'
      at Context.<anonymous> (test/CorporateView.js:108:24)
```

Abbildung 18: Ausgabe der gefundenen Fehler mit Mocha (Ausschnitt)

Die Ausgabe wirkt allgemein weniger übersichtlich: Bei Assertions mit einem Test auf Gleichheit, wie z.B. Test 5 (siehe Abbildung 19), werden der tatsächliche Wert (in roter Schrift) und der Erwartungswert (grüne Schrift) einfach untereinander aufgelistet. Dem tatsächlich erhaltenen Wert wird dabei stets ein „-“ vorangestellt, dem Soll-Wert hingegen ein „+“. Insbesondere das Minus kann dabei stellenweise für kurzzeitige Verwirrung sorgen, nämlich dann, wenn der tatsächliche Wert eine Zahl ist. Durch das fehlende Leerzeichen kann somit eine positive Zahl bei Unachtsamkeit oder Unwissenheit für eine Negative gehalten werden, was es wiederum erschweren kann, die Fehlerursache zu finden.

Bei den weiteren Tests werden Erwartungswert und Ist-Wert gar nicht erst farblich voneinander getrennt. Es wird lediglich eine kurze Beschreibung des Fehlers in roter Schrift geliefert. Bei dem Array aus Test 6 aus Abbildung 19 werden im Gegensatz zu dem Äquivalent von Jest die beiden enthaltenen Einträge auch nicht angezeigt, sondern lediglich durch die eckigen Klammern und den Inhalt `Array(2)` kenntlich gemacht, dass es sich um ein Array mit der Länge 2 handelt. Die Fehlermeldungen sind zwar trotzdem aufschlussreich (wenn auch kurz), jedoch wirkt die Jest-Variante sauberer und besser lesbar (u.a. auch durch die durchweg durchgezogene farbliche Trennung der beiden Werte), wodurch der Tester schneller Rückschlüsse auf die Fehlerursache ziehen kann.

Alles in allem kann festgehalten werden, dass Jest im Fehlerfall verständlichere und besser lesbare Ausgaben liefert als Mocha.

6 Fazit

Zielsetzung der vorliegenden Arbeit war die Untersuchung der beiden Test-Frameworks Mocha und Jest in Bezug auf deren Tauglichkeit für den Einsatz zum Testen von React-Applikationen bei der Firma Sensape. Um dieses Ziel zu erreichen, wurden beide Frameworks an Hand definierter Qualitätskriterien miteinander verglichen. Der Schwerpunkt war dabei die Implementierung einer Reihe von Komponententests mit beiden Test-Frameworks für eine bereits existierende React-Anwendung von Sensape.

Jest konnte insbesondere durch die integrierte Snapshot Testing-Funktionalität überzeugen, welche vor allem das Schreiben von Tests für React-Komponenten mit überschaubarer Komplexität um einiges erleichtert, jedoch auch für „Nicht-React-Komponenten“ sinnvoll eingesetzt werden kann. So fiel die zum Schreiben der Snapshot Tests benötigte Menge an Quellcode, sowie die dafür benötigte Zeit, mitunter deutlich geringer aus als bei Mocha, wo alle Tests mit klassischen Assertions geschrieben werden mussten.

Gleichzeitig haben die Untersuchungen jedoch gezeigt, dass auch Snapshot Testing keine Allzwecklösung ist: Nicht alle Komponenten lassen sich mit dieser Methode testen, und stellenweise bietet es sich auch nicht an. Das Testen von Redux mittels Snapshots brachte bei der ausgewählten Reducer-Funktion z.B. eher Nachteile mit, als dass es nützlich war. Allgemein ist der größte Nachteil von Snapshot Tests der niedrigere Automatisierungsgrad, der dadurch entsteht, dass alle Snapshots manuell auf Richtigkeit geprüft werden müssen. Hier ist stets ein Abwägen erforderlich, ob die Testkomponente für eine Implementierung als Snapshot Test sinnvoll ist. Auch eignet sich die Funktionalität nicht für die testgetriebene Entwicklung. Da es durch die Verwendung des Frameworks in Kombination mit Enzyme jedoch genauso möglich ist, alle Tests als klassische Assertions im BDD-Stil zu schreiben, steht Jest dem konkurrierenden Mocha in dieser Hinsicht in Nichts nach.

Mocha konnte in Verbindung mit Chai durch eine flexiblere Formulierung von Assertions durch eine Verkettung der Matcher aufwarten, in welchem Punkt es der integrierten Assertion-Bibliothek von Jest überlegen ist. Dieser Pluspunkt wird jedoch dadurch aufgehoben, dass Jest deutlich bessere Fehlermeldungen liefert, wenn ein Test fehlschlägt. Auch die Installation und Konfiguration von Jest und die damit verbundene Integration in eine bestehende Anwendung war bei Jest mit weniger Komplikationen verbunden. Der Autor zieht somit aus der Untersuchung das Fazit, dass Jest in Verbindung mit Enzyme für das Einsatzgebiet „Testen des Frontends bei React-Anwendungen“ die bessere Wahl ist.

7 Ausblick

Sofern sich das Unternehmen Sensape dafür entscheidet, die Empfehlung, Jest zukünftig für das Testen seiner Reakt-Anwendungen zu verwenden, anzunehmen, besteht der nächste Schritt in der Einbindung der bereits geschriebenen Tests, die bei dieser Arbeit entstanden sind, in die aktuelle Entwicklungsversion der Second Screen App. Diese liegt zum Zeitpunkt der Fertigstellung dieser Arbeit bereits in der Vesion 0.12.3 vor.

Im nächsten Schritt soll die App dann um noch weitere Tests erweitert werden, um so eine möglichst hohe Testabdeckung zu erzielen.

Das langfristige Ziel ist die Integration der gesamten Tests für die Second Screen App in das von Sensape verwendete Continuous Integration System.

Die Ergebnisse dieser Arbeit liefern zudem wertvolle Erkenntnisse, welche auch für den Einsatz bei weiteren React-Anwendungen, die von Sensape entwickelt werden, von Nutzen sind, da sie eine Richtlinie vorgeben, wie solche Applikationen effizient zu testen sind.

Literatur

Abramov, Dan (2017): *You Might Not Need Redux*. URL: https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367 [Stand 29.07.2017]

Ackermann, Philip (2016): *JavaScript. Das umfassende Handbuch*, Bonn.

Bachuk, Alex (28.06.2016): *Redux · An Introduction*. URL: https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/ [Stand 02.07.2017]

Brandes, Christian (o.J. [ca. 2011]): *Konzeption von produktionsnahen Testumgebungen*. URL: http://pi.informatik.uni-siegen.de/stt/31_1/03_Technische_Beitraege/Brandes_Testumgebungen.pdf [Stand 24.06.2017]

Codeacademy (o.J. [ca. 2016]): *React: The Virtual DOM. Fighting Wasteful DOM Manipulation*. URL: https://www.codecademy.com/articles/react-virtual-dom [Stand 29.07.2017]

Coenraets, Christophe (2014): *Sample Mobile Application with React and Cordova*. URL: http://coenraets.org/blog/2014/12/sample-mobile-application-with-react-and-cordova/ [Stand 03.09.2017]

Dart (o.J. [ca. 2016]): *Connect Dart & HTML*. URL: https://webdev.dartlang.org/tutorials/low-level-html/connect-dart-html [Stand 03.09.2017]

Facebook (2017a): *React.Component*. URL: https://facebook.github.io/react/docs/react-component.html [Stand 18.06.2017]

Facebook (2017b): *Shallow Renderer*. URL: https://facebook.github.io/react/docs/shallow-renderer.html [Stand 03.09.2017]

Facebook (2017c): Using With Webpack. URL: https://facebook.github.io/jest/docs/webpack.html [Stand 13.08.2017]

Fucci, Davide u.a. (2016): *An External Replication on the Effects of Test-driven Development Using a Multi-Site Blind Analysis Approach*. URL: http://people.brunel.ac.uk/~csstmms/FucciEtAl_ESEM2016.pdf [Stand 23.07.2017]

Gravelle, Rob (o.J. [ca. 2015]): *Guidelines for Testing Front-end Web Components.* URL: http://www.htmlgoodies.com/beyond/webmaster/guidelines-for-testing-front-end-web-components.html [Stand 14.07.2017]

Liggesmeyer, Peter (2009): *Software-Qualität. Testen, Analysieren und Verifizieren von Software*, 2. Auflage, Heidelberg.

Linz, Tilo (2013): *Testen in Scrum-Projekten. Leitfaden für Softwarequalität in der agilen Welt*. URL: https://download.e-bookshelf.de/download/0003/8316/92/L-X-0003831692-0002232028.XHTML/index.xhtml [Stand 03.09.2017]

Long, James (2014): *Removing User Interface Complexity, or Why React is Awesome*. URL: http://jlongster.com/Removing-User-Interface-Complexity,-or-Why-React-is-Awesome [Stand 29.07.2017]

Marcenko, Vadim (2014): *Aufbau einer Continuous Integration-Plattform für agile Web-Projekte. Theoretische Grundlagen und Beispielprojekt*, Saarbrücken.

MDN (2017): *HTML*. URL: https://developer.mozilla.org/de/docs/Web/HTML [Stand 16.06.2017]

Mocha (2017): *Arrow Functions*. URL: https://mochajs.org/#arrow-functions [Stand 03.08.2017]

Myers, Glenford J. (1979): *The Art of Software Testing*, New York.

Nagappan, Nachiappan u.a. (2008): „*Realizing quality improvement through test driven development: results and experiences of four industrial teams*“. In: *Empirical Software Engineering*, Jg. 13, Nr. 3, S. 289-302, URL: https://github.com/tpn/pdfs/blob/master/Realizing%20Quality%20Improvement%20Through%20Test%20Driven%20Development%20-%20Results%20and%20Experiences%20of%20Four%20Industrial%20Teams%20(nagappan_tdd).pdf [Stand 23.07.2017]

Neumann, Alexander (2015): *Sprachspezifikation ECMAScript 2015 ist fertig*. URL: https://www.heise.de/developer/meldung/Sprachspezifikation-ECMAScript-2015-ist-fertig-2715826.html [Stand 02.07.2017]

North, Dan (2006): *Introducing BDD*. URL: https://dannorth.net/introducing-bdd/ [Stand 03.09.2017]

NPM (2017): *npm*. URL: https://www.npmjs.com/ [Stand 29.07.2017]

Redux (2016): *Motivation*. URL: http://redux.js.org/docs/introduction/Motivation.html [Stand 30.06.2017]

Redux (2017): *Organizing State*. URL: http://redux.js.org/docs/faq/OrganizingState.html [Stand 30.06.2017]

Sanchez, Julio Cesar/Williams, Laurie/Maximilien, E. Michael (2007): *A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.6319&rep=rep1&type=pdf [Stand 23.07.2017]

Schäfer, Werner (2010): *Softwareentwicklung. Einstieg für Anspruchsvolle*, München.

Scott, Stephen (2017): *The Right Way to Test React Components*. URL: https://medium.freecodecamp.org/the-right-way-to-test-react-components-548a4736ab22 [Stand 03.09.2017]

Sensape (2017): *Phantastic Photobox*. URL: https://phantastic-photobox.com/ [Stand 03.09.2017]

Shilman, Michael (2016): *Testing Frameworks*. URL: https://stateofjs.com/2016/testing/ [Stand 04.09.2017]

Sneed, Harry M./Baumgartner, Manfred/Seidl, Richard (2009): *Der Systemtest. Von den Anforderungen zum Qualitätsnachweis*, 2., aktualisierte und erweiterte Auflage, München.

Spillner, Andreas/Linz, Tilo (2012): *Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*, 5., überarbeitete und aktualisierte Auflage, Heidelberg.

Starke, Gernot (2015): *Effektive Softwarearchitekturen. Ein praktischer Leitfaden*, 7., überarbeitete Auflage, München.

Wikibooks (2011): *PHP praxisorientiert lernen: Wie PHP funktioniert*. URL: https://de.wikibooks.org/wiki/PHP_praxisorientiert_lernen:_Wie_PHP_funktioniert [Stand 03.09.2017]

Witte, Frank (2016): *Testmanagement und Softwaretest*. Theoretische Grundlagen und praktische Umsetzung, Wiesbaden.

Wrobel, Gunnar (2015): *JavaScript Tools. Besserer Code durch eine professionelle Programmierumgebung*, München.

Zeigermann, Oliver/Hartmann, Nils (2016): *React. Die praktische Einführung in React, React Router und Redux*, Heidelberg.

Zuser, Wolfgang/Grechenig, Thomas/Köhle, Monika (2004): *Software Engineering. mit UML und dem Unified Process*, 2., überarbeitete Auflage, München.

Anlagen

Abbildungen

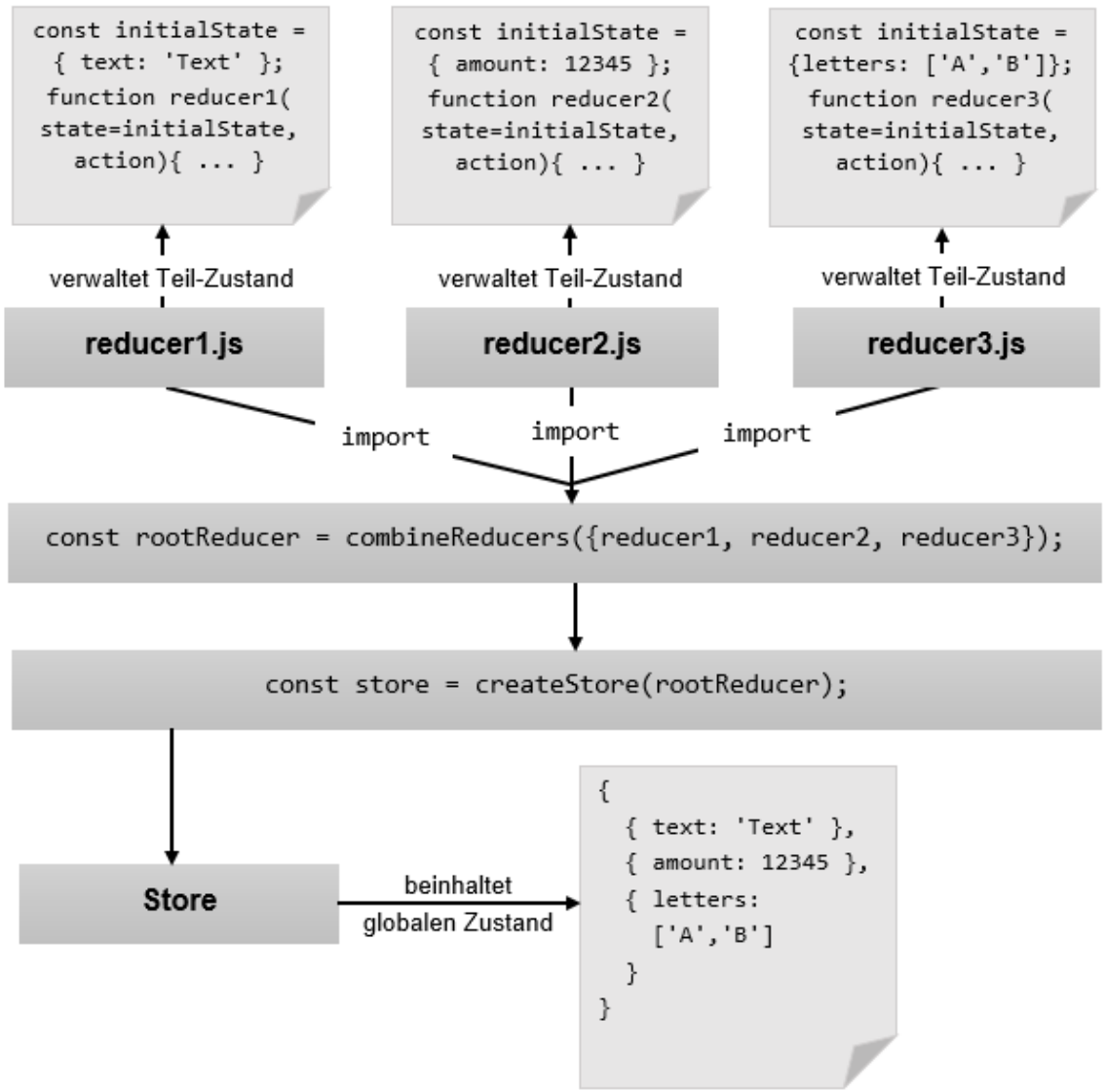

Abbildung 19: Modell zur Beziehung zwischen Store und Reducer

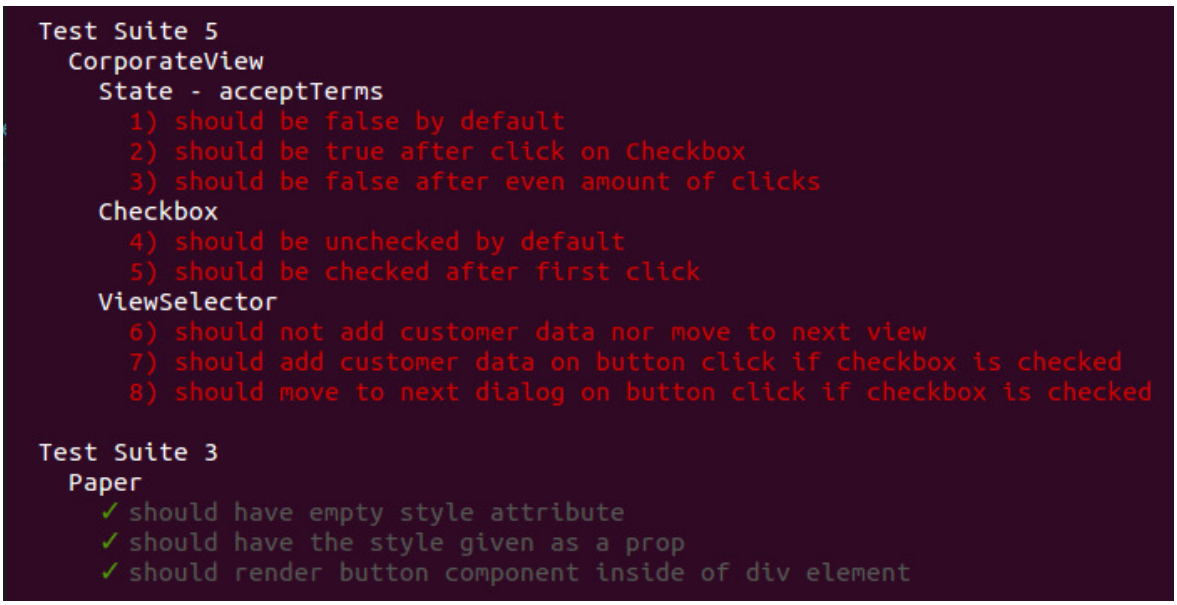

Abbildung 20: Fehlgeschlagene Tests bei Mocha

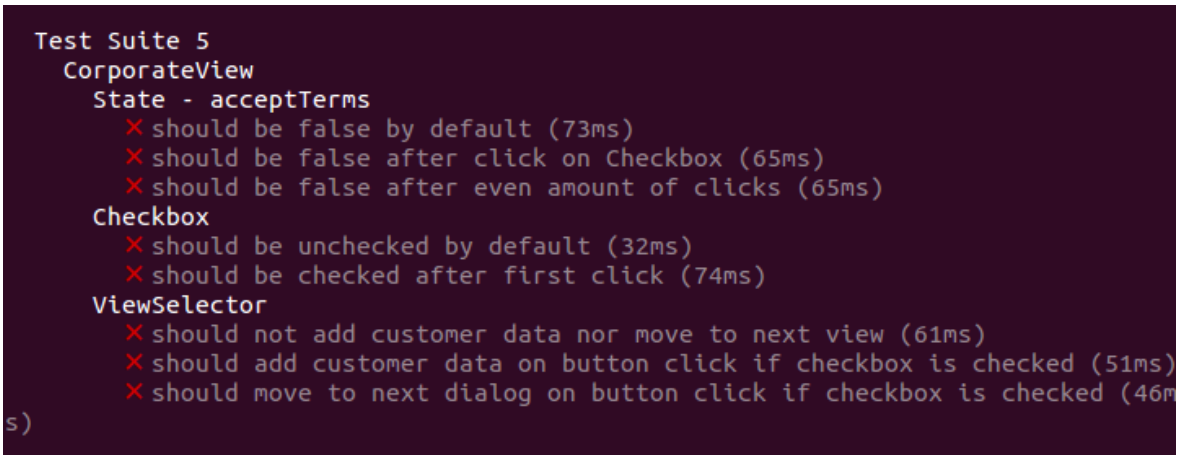

Abbildung 21: Fehlgeschlagene Tests bei Jest

Testfallspezifikationen

Test-Suite-ID	/TS01/	
Testobjekt	StpAppBar-Komponente	
TC-ID	**Eingabewert & Konditionen**	**Erwartetes Ergebnis**
/TC010/	`<StpAppBar title='' />`	`StpAppBar` wird ohne Text gerendert.
/TC020/	`<StpAppBar title='Phantastic Photobox' />`	`StpAppBar` wird mit dem Text `Phantastic Photobox` gerendert.
/TC030/	`<StpAppBar title='Phantastic Photobox is Phantastic' />`	`StpAppBar` wird ohne Text gerendert.
/TC040/	Wie /TC020/	`StpAppBar` wird ohne „Zurück“-Icon gerendert.
/TC050/	`<StpAppBar title='Phantastic Photobox' backicon={true} />`	`StpAppBar` wird mit “Zurück”-Icon gerendert.
/TC050/	Wie /TC050/	`StpAppBar` (mit Icon) wird mit dem Text `Phantastic Photobox` gerendert.

Tabelle 4: Testfallspezifikation für /TS01/

Test-Suite-ID	/TS02/	
Testobjekt	ButtonCmp (Button-Komponente)	
TC-ID	**Eingabewert & Konditionen**	**Erwartetes Ergebnis**
/TC070/	`<ButtonCmp label="Button label" ... />`	Button-Komponente wird gerendert mit Aufschrift `Button label`

/TC080/	`<ButtonCmp label={false} ... />`	Button-Komponente wird nicht gerendert
Testobjekt	config-Reducer	
/TC090/	Aufruf der Reducer-Funktion ohne Action	Der zurückgelieferte State entspricht dem Initial-State

/TC100/	Aufruf mit Action vom Typ `SET_CONFIG` mit geänderten Werten für `buttonLabelYes` (`"Ja, bitte."`) und `buttonLabelNo` (`"Nein, danke."`)	Der zurückgelieferte State entspricht dem Initial-State, mit zwei Änderungen:
		`strings.contactView. buttonLabelYes: "Ja, bitte."`
		`strings.contactView. buttonLabelNo: "Weiter ohne Daten"`

Testobjekt	ButtonCmp + configReducer	
/TC110/	`<ButtonCmp label={config.strings.contactView. buttonLabelNo} ... />` Config-State nicht modifiziert (entspricht dem Initial-State von `config`)	Button-Komponente wird gerendert mit Aufschrift `Weiter ohne Daten`
/TC120/	`<ButtonCmp label={config.strings.contactView. buttonLabelNo} ... />` Config-State modifiziert wie bei /TC040/	Button-Komponente wird gerendert mit Aufschrift `Nein, danke.`
/TC130/	`<ButtonCmp label={config.strings.contactView. buttonLabelNo} ... />` Config-State durch Action vom Typ `SET_CONFIG` aktualisiert, geänderter Wert für `buttonLabelNo : false`	Button-Komponente wird nicht gerendert

Tabelle 5: Testfallspezifikation für /TC02/

Testsuite-ID	/TS03/	
Testobjekt	CorporateView: `this.state.acceptTerms`	
TC-ID	Eingabewert & Konditionen	Erwartetes Ergebnis
/TC140/	`<CorporateView ... />` als Full DOM Rendering - `CorporateView` hat Zugriff auf den globalen State als Props	`acceptTerms` hat den Wert `false`
/TC150/	Analog zu /TC200/ - Szenario: Klick auf die AGB-Check-box	`acceptTerms` hat den Wert `true`
/TC160/	Analog zu /TC200/ - Szenario: Zwei aufeinanderfolgende Klicks auf die AGB-Checkbox	`acceptTerms` hat den Wert `false`
Testobjekt	CorporateView: CheckboxCmp	
/TC170/	Analog zu /TC200/	Die Checkbox für die Zustim-mung zu den AGBs ist abge-wählt.
/TC180/	Analog zu /TC210/	Die Checkbox ist markiert.
Testobjekt	VisibleView	
/TC190/	Analog zu /TC190/ (ohne Szenario) mit folgendem Zusatz: - `dialog` hat folgenden Wert: `{ recent : "SHARING_DIALOG", next : "CORPORATE_DIALOG" }` (wählt den `CorporateView` aus) - `customer` hat folgenden Wert:	Es werden weder die Kunden-daten in den globalen State übernommen noch der nächste View angezeigt (Keine Actions vom Typ `ADD_CUSTOMER_DATA` oder `SHOW_NEXT_DIALOG` versendet)

	`{ src : "/image.jpg" }` - Szenario: Klick auf Button	
/TC200/	Analog zu /TC250/ (ohne Szenario) - Szenario: Klick auf AGB-Checkbox, anschließend Klick auf Button	Es werden die Kundendaten in den globalen State übernommen (Action vom Typ `ADD_CUSTOMER_DATA` versendet, `payload.acceptTerms` ist vorhanden und hat den Wert `true`)
/TC210/	Analog zu /TC260/	Der globale State aktualisiert den aktuell aktiven View (Eine Action vom Typ `SHOW_NEXT_DIALOG` wird versendet, die Eigenschaft `recent` vom Payload hat dabei den Wert `CORPORATE_DIALOG`)

Tabelle 6: Testfallspezifikation für /TS03/

Testsuite-ID	/TS04/	
Testobjekt	Paper	
TC-ID	**Eingabewert & Konditionen**	**Erwartetes Ergebnis**
/TC220/	`<Paper />`	Attribut `style` des darunterliegenden `<div>`-Elements hat den Wert `{}` (leeres Objekt, entspricht keinem Style)
/TC230/	`<Paper style={{height: '900px'}} />`	Attribut `style` des darunterliegenden `<div>`-Elements hat den Wert `{height:'900px'}` (entspricht einer Höhe von 900px)

/TC240/	`<Paper>`
	`<ButtonCmp type='button' label='Button label' />`
	`</Paper>`
		Die ButtonCmp wird als Kind-Element gerendert, d.h. es existiert ein Element von Typ `button`.

Tabelle 7: Testfallspezifikation für /TS04/

Test-Suite-ID	/TS05/	
Testobjekt	StartView	
TC-ID	**Eingabewert & Konditionen**	**Erwartetes Ergebnis**
/TC250/	`<StartView ... />` als Full DOM Rendering - `StartView` hat Zugriff auf globalen State als Props - Eigenschaft `customer` beinhaltet genau einen Eintrag	Es wird genau 1 Bild auf dem `StartView` angezeigt.
/TC260/	Analog zu /TC140/, aber `customer` um drei weitere Einträge erweitert (insgesamt somit 4 Einträge vorhanden)	Es werden genau 4 Bilder auf dem `StartView` angezeigt.
Testobjekt	Action-Creator addCustomerData()	
/TC270/	`addCustomerData({src : "/image.jpg"})`	`{ type: "ADD_CUSTOMER_DATA, payload : {src:"/image.jpg"}}`
Testobjekt	customer-Reducer	
/TC280/	Aufruf der Reducer-Funktion ohne Action	Der zurückgelieferte State entspricht einem leeren Objekt (Initial-State des customer-Reducers)
/TC290/	Aufruf mit Action-Creator von /TC160/	Neuer `customer`-State: `{src : "/image.jpg"}`
Testobjekt	VisibleView	

/TC300/	`<VisibleView ... />` mit Anbindung an Store - Store-Objekt entspricht Kombination der Initial-States aller 4 Reducer, mit folgender Anpassung: - `photos` hat folgenden Wert: `[{id : 1, src : "/image.jpg", date : new Date()}]` (1 Eintrag, d.h. 1 Bild wird angezeigt) - Szenario: Klick auf das angezeigte Bild	Action vom Typ `ADD_CUSTOMER_DATA` wird versendet, als Payload wird der Pfad des ausgewählten Bildes (`src`) als Wert der Eigenschaft `src` übergeben.

Tabelle 8: Testfallspezifikation für /TS05/

Quellcode

```
import React from 'react';
import StpAppBar from '../src/stp/components/elements/StpAppBar';
import {mount} from 'enzyme';
import {expect} from 'chai';


describe('Test Suite 1', () => {
      describe('StpAppBar', () => {
            it('should have no text if title is empty string', () => {
            const wrapper = mount(<StpAppBar title='' />);
            expect(wrapper.find('header').text()).to.eql('');
            }),
          it('should display the title given as a prop', () => {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox'
            />);
            expect(wrapper.find('header').text()).to.eql('Phantastic Pho
            tobox');
          }),
          it('should not display no text if title is too long', () => {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox
            is Phantastic' />);
            expect(wrapper.find('header').text()).to.eql('');
          }),
          it('should display no icon if no backicon prop is given', () =>
            {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox'
            />);
            expect(wrapper.find('svg').exists()).to.eql(false);
          }),
          it('should display the return icon if backicon prop is given',
          () => {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox'
            backicon={true} />);
            expect(wrapper.find('svg').exists()).to.eql(true);
          }),
           it('should display the correct title if there is a backicon',
           () => {
            const wrapper = mount(<StpAppBar title='Bar with icon'
            backicon={true} />);
            expect(wrapper.find('header').text()).to.eql('Bar with
            icon');
          });
      });
});
```

Quellcode 1: Komplette Implementierung von /TS01/ mit Mocha

```
import React from 'react';
import StpAppBar from '../src/stp/components/elements/StpAppBar';
import renderer from 'react-test-renderer';
import {mount} from 'enzyme';
```

```
describe('Test Suite 1', () => {
      describe('StpAppBar', () => {
            it('should have no text if title is empty string', () => {
            const wrapper = mount(<StpAppBar title='' />);
            expect(wrapper.find('header').text()).toBe('');
            }),
          it('should display the title given as a prop', () => {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox'
            />);
            expect(wrapper.find('header').text()).toBe('Phantastic Photo
            box');
          }),
          it('should not display no text if title is too long', () => {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox
            is Phantastic' />);
            expect(wrapper.find('header').text()).toBe('');
          }),
          it('should display no icon if no backicon prop is given', () =>
          {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox'
            />);
            expect(wrapper.find('svg').exists()).toBe(false);
          }),
          it('should display the return icon if backicon prop is given',
          () => {
            const wrapper = mount(<StpAppBar title='Phantastic Photobox'
            backicon={true} />);
            expect(wrapper.find('svg').exists()).toBe(true);
          }),
           it('should display the correct title if there is a backicon',
           () => {
            const wrapper = mount(<StpAppBar title='Bar with icon'
            backicon={true} />);
            expect(wrapper.find('header').text()).toBe('Bar with icon');
          });
      });
});
```

Quellcode 2: Komplette Implementierung von /TS01/ mit Jest

```
import * as dialogFilters from '../../src/stp/constants/DialogFilters';

export const initialState = {
  primaryColor: 'rgb(104,178,42)',
  grayColor: 'rgb(189,189,189)',
  warningColor: '#e53020',
  timeout: '180000',
  customFont: false,
  strings: {

    contactView: {
      title:'Dürfen wir...',
      subheader:'... Dich kontaktieren?',
      buttonLabelYes:'Na klar. Meldet Euch!',
      buttonLabelNo:'Weiter ohne Daten',
      lblName:'Name',
      lblPreName: 'Vorname',
```

```
      lblSurName: 'Nachname',
      lblTel:'Telefon',
      lblMail:'Email'
    },
    corporateView: {
      title:'Super!',
      subheader:'... Nur noch zwei Fragen.',
      question:'Dürfen wir das Bild für unsere <br/><b>Social-Media-Akti
      vitäten</b><br/>verwenden? Du kannst dem jederzeit widersprechen!',
      agbBefore: 'Erklärst du dich zudem mit unseren',
      agb: '</br>Allgemeinen Geschäftsbedingungen</br>',
      agbAfter: 'einverstanden? Du musst diesen zustimmen, um den Vorgang
      abzuschließen.',
      agbCheckbox: 'Ja, ich stimme den AGB zu.',
      agbTitle:'Eligibility Requirements and Terms & Conditions:',
      agbText:"<h3>1. Images (Files and Usage)</h3><p>Paragraph
      1</p><p>Paragraph 2</p><p>Paragraph 3</p><h3>2. Cancellation     of
      Images</h3><p>Paragraph 1</p><p>Paragraph 2</p>",
      corporateCheckbox: 'Ja, ihr dürft mein Foto verwenden!',
      buttonLabelYes:'Endlich Mein Phantastic Photo drucken!',
      buttonLabelNo:'Nein, Danke.'
    },
    goodbyeView: {
      title:'Super!',
      subheader:'... Es geht los!',
      printing:'Alles klar, wir drucken dein Foto.<br/>Viel Spaß damit!'
    },
    sharingView: {
      title:'Sollen wir...',
      subheader:'... Dir das Bild schicken?',
      buttonLabelYes:'JA! Schickt mir das Bild',
      buttonLabelNo:'Nein, Danke.',
      lblMail:'Email'
    },
    startView: {
      title:'Phantastic Photobox',
      subheader:'Wähle Dein Bild',
      anotherPic:'Oder mache ein neues <b>Phantastisches Photo!</b>'
    },
    errorMessages: {
      name:'Bitte gib deinen Namen ein',
      preName:'Bitte gib deinen Vornamen ein',
      surName:'Bitte gib deinen Nachnamen ein',
      phone:'Bitte gib deine Telefonnummer ein',
      email:'Bitte gib deine E-Mail-Adresse ein'
    }
  },
  followups: {
    START_DIALOG : dialogFilters.CONTACT_DIALOG,
    CONTACT_DIALOG : dialogFilters.SHARING_DIALOG,
    SHARING_DIALOG : dialogFilters.CORPORATE_DIALOG,
    CORPORATE_DIALOG : dialogFilters.GOODBYE_DIALOG,
    GOODBYE_DIALOG : dialogFilters.START_DIALOG
  }
};
```

Quellcode 3: Mock-Objekt für /TS02/ (für beide Test-Frameworks)

```
import React from 'react';
import ButtonCmp from '../src/stp/components/elements/ButtonCmp';
import { expect } from 'chai';
import {shallow} from 'enzyme';
import renderer from 'react-test-renderer';
import config from '../src/stp/reducers/config';
import * as types from '../src/stp/constants/ActionTypes';
import {initialState} from './constants/ButtonCmp'

const payload = {strings: {contactView: {buttonLabelYes:'Ja, bitte.',
buttonLabelNo:'Nein, danke.'}}};
const expectedAction = {
    type: types.SET_CONFIG,
    payload
}

describe('Test Suite 2', () => {
    describe('ButtonCmp', () => {
        it('should have the label given as a prop"', () => {
            // Erzeuge ButtonCmp mit Aufschrift "Button label"
            const wrapper = shallow(<ButtonCmp label="Button label"
                type="button" />);
            expect(wrapper.props().label).to.equal("Button label");
        }),
        it('should not render anything if no label is given', () => {
            const wrapper = shallow(<ButtonCmp label={false}
                type="button" />);
            // type() liefert den Typ des aktuellen Knotens des Wrappers
            zurück
            // Kein Knoten entspricht dem Wert null
            // Erwartet: wrapper.type() ist null ()
            expect(wrapper.type()).to.equal(null);
        });
    }),
    describe('config reducer', () => {
        it('should return the initial state if it receives no action', ()
         => {
            // Rufe config-Reducer ohne Action auf
            // Erwartet: config-State entspricht dem Initial-State
            expect(config(undefined, {})).to.eql(initialState);
        });
        it('should update the state correctly if config is changed', ()
        => {
            // initialState-Objekt kopieren
            // ohne Referenz auf Original-Objekt
            const modifiedState = JSON.parse(JSON.stringify(initial
            State));
            modifiedState.strings.contactView.buttonLabelYes = "Ja,
            bitte.";
            modifiedState.strings.contactView.buttonLabelNo = "Nein,
            danke.";
            // Ändere zwei Eigenschaften des config-States durch config-
            Reducer
            const newState = config(initialState, expectedAction);
            // Erwartet: ButtonLabelYes und ButtonLabelNo anders, Rest
            unverändert
            expect(newState).to.be.an('object').and.eql(modifiedState);
        });
    }),
    describe('integration tests', () => {
```

```
        it('should render button with the default label if global state
        is not modified', () => {
            // Rufe config-Reducer ohne Action auf -->
            // configState sollte dem Initial-State entsprechen
            const configState = config(undefined, {});
            const wrapper = shallow(<ButtonCmp
                label={configState.strings.contactView.buttonLabelNo}
                type="button" />);
            expect(wrapper.props().label).to.equal("Weiter ohne Daten");
        }),
        it('should render the button with new label if global state is
        modified', () => {
            // Ändere zwei Eigenschaften des config-States durch config-
            Reducer
            const configState = config(initialState, expectedAction);
            const wrapper = shallow(<ButtonCmp
                label={configState.strings.contactView.buttonLabelNo}
                type="button" />);
            // Erwartet: Label hat den Wert 'Nein, danke.'
            expect(wrapper.props().label).to.equal("Nein, danke.");
        }),
        it('should not render anything if no button label is given after
        changing global state', () => {
            // Ändere Eigenschaft strings.contactView.buttonLabelNo auf
            Wert false
            const newAction = {type: types.SET_CONFIG, payload :
                {strings: {contactView: { buttonLabelNo: false}}}};
            const configState = config(initialState, newAction);
            const wrapper = shallow(<ButtonCmp
                label={configState.strings.contactView.buttonLabelNo}
                type="button" />);
            // Erwartet: Der Button wird nicht gerendert (return null)
            expect(wrapper.type()).to.equal(null);
        });
    });
});
```

Quellcode 4: Komplette Implementierung von /TS02/ mit Mocha

```
import React from 'react';
import ButtonCmp from '../src/stp/components/elements/ButtonCmp';
import renderer from 'react-test-renderer';
import {setConfig} from '../src/stp/actions/config';
import config from '../src/stp/reducers/config';
import * as types from '../src/stp/constants/ActionTypes';
import {initialState} from './constants/ButtonCmp';

const payload = {strings: {contactView: {buttonLabelYes:'Ja, bitte.',
buttonLabelNo:'Nein, danke.'}}};
const expectedAction = {
    type: types.SET_CONFIG,
    payload
}

describe('Test Suite 2', () => {
    describe('Button', () => {
        it('should have the label given as a prop', () => {
            // Snapshot Test
```

```
            // Erzeuge ButtonCmp mit Aufschrift "Button label"
            const tree = renderer.create(
                <ButtonCmp label="Button label" type="button" />
            ).toJSON();
            // Erzeuge Snapshot und vergleiche diesen mit tree
            expect(tree).toMatchSnapshot();
        }),
        it('should not render anything if no label is given', () => {
            // Snapshot Test
            // Erzeuge ButtonCmp mit Aufschrift die den Wert false hat
            const tree = renderer.create(
                <ButtonCmp label={false} type="button" />
            ).toJSON();
            // Erzeuge Snapshot und vergleiche diesen mit tree
            // Erwartet: return null
            expect(tree).toMatchSnapshot();
        });
    }),
    describe('config reducer', () => {
        it('should return the initial state if it receives no action', ()
            => {
            // Rufe config-Reducer ohne Action auf
            // Erwartet: config-State entspricht dem Initial-State
            expect(config(undefined, {})).toEqual(initialState);
        }),
        it('should update the state correctly if config is changed', ()
        => {
            // Snapshot Test
            // Ändere zwei Eigenschaften des config-States durch config-
            Reducer
            const newState = config(initialState, expectedAction);
            // Erzeuge Snapshot und vergleiche diesen mit newState
            // Erwartet: ButtonLabelYes und ButtonLabelNo anders, Rest
            unverändert
            expect(newState).toMatchSnapshot();
        });
    }),
    describe('integration tests', () => {
        it('should render button with the default label if global state
        is not modified', () => {
            // Snapshot Test
            // Rufe config-Reducer ohne Action auf -->
            // configState sollte dem Initial-State entsprechen
            const configState = config(undefined, {});
            const tree = renderer.create(
                <ButtonCmp
                label={configState.strings.contactView.buttonLabelNo}
                type="button" />
            ).toJSON();
            // Erzeuge Snapshot und vergleiche diesen mit tree
            // Erwartet: Label hat Wert 'Weiter ohne Daten.'
            expect(tree).toMatchSnapshot();
        }),
        it('should render the button with new label if global state is
        modified', () => {
            // Snapshot Test
            // Ändere zwei Eigenschaften des config-States durch config-
            Reducer
            const configState = config(initialState, expectedAction);
            const tree = renderer.create(
                <ButtonCmp
```

```
                label={configState.strings.contactView.buttonLabelNo}
                type="button" />
            ).toJSON();
            // Erzeuge Snapshot und vergleiche diesen mit tree
            // Erwartet: Label hat den Wert 'Nein, danke.'
            expect(tree).toMatchSnapshot();
        }),
        it('should not render anything if no button label is given after
        changing global state', () => {
            // Snapshot Test
            // Ändere Eigenschaft strings.contactView.buttonLabelNo auf
            Wert false
            const newAction = {type: types.SET_CONFIG, payload :
                {strings: {contactView: { buttonLabelNo: false}}}};
            const configState = config(initialState, newAction);
            const tree = renderer.create(
                <ButtonCmp label={configState.strings.contactView.but
                tonLabelNo} type="button" />
            ).toJSON();
            // Erzeuge Snapshot und vergleiche diesen mit tree
            // Erwartet: Der Button wird nicht gerendert (return null)
            expect(tree).toMatchSnapshot();
        });
    });
});
```

Quellcode 5: Komplette Implementierung von /TS02/ mit Jest

```
import * as dialogFilters from '../../src/stp/constants/DialogFilters';

export const globalState = {
        customer: {
            src: "/image.jpg"
        },
        config: {
        primaryColor: 'rgb(104,178,42)',
        grayColor: 'rgb(189,189,189)',
        warningColor: '#e53020',
        timeout: '180000',
        customFont: false,
        strings: {
            contactView: {
              title:'Dürfen wir...',
              subheader:'... Dich kontaktieren?',
              buttonLabelYes:'Na klar. Meldet Euch!',
              buttonLabelNo:'Weiter ohne Daten',
              lblName:'Name',
              lblPreName: 'Vorname',
              lblSurName: 'Nachname',
              lblTel:'Telefon',
              lblMail:'Email'
            },
            corporateView: {
              title:'Super!',
              subheader:'... Nur noch zwei Fragen.',
              question:'Dürfen wir das Bild für unsere <br/><b>Social-Me
              dia-Aktivitäten</b><br/>verwenden? Du kannst dem jederzeit
              widersprechen!',
```

```
            agbBefore: 'Erklärst du dich zudem mit unseren',
            agb: '</br>Allgemeinen Geschäftsbedingungen</br>',
            agbAfter: 'einverstanden? Du musst diesen zustimmen, um den
            Vorgang abzuschließen.',
            agbCheckbox: 'Ja, ich stimme den AGB zu.',
            agbTitle:'Eligibility Requirements and Terms & Condi
            tions:',
            agbText:'AGB-Text',
            corporateCheckbox: 'Ja, ihr dürft mein Foto verwenden!',
            buttonLabelYes:'Endlich Mein Phantastic Photo drucken!',
            buttonLabelNo:'Nein, Danke.'
          },
          goodbyeView: {
            title:'Super!',
            subheader:'... Es geht los!',
            printing:'Alles klar, wir drucken dein Foto.<br/>Viel Spaß
            damit!'
          },
          sharingView: {
            title:'Sollen wir...',
            subheader:'... Dir das Bild schicken?',
            buttonLabelYes:'JA! Schickt mir das Bild',
            buttonLabelNo:'Nein, Danke.',
            lblMail:'Email'
          },
          startView: {
            title:'Phantastic Photobox',
            subheader:'Wähle Dein Bild',
            anotherPic:'Oder mache ein neues <b>Phantastisches
            Photo!</b>'
          },
          errorMessages: {
            name:'Bitte gib deinen Namen ein',
            preName:'Bitte gib deinen Vornamen ein',
            surName:'Bitte gib deinen Nachnamen ein',
            phone:'Bitte gib deine Telefonnummer ein',
            email:'Bitte gib deine E-Mail-Adresse ein'
          }
      },
      followups: {
          START_DIALOG : dialogFilters.CONTACT_DIALOG,
          CONTACT_DIALOG : dialogFilters.SHARING_DIALOG,
          SHARING_DIALOG : dialogFilters.CORPORATE_DIALOG,
          CORPORATE_DIALOG : dialogFilters.GOODBYE_DIALOG,
          GOODBYE_DIALOG : dialogFilters.START_DIALOG
      }
  },
  dialog: {
      recent: dialogFilters.SHARING_DIALOG,
      next: dialogFilters.CORPORATE_DIALOG,
  },
  photos: [{
      id: 1,
      src: '/image.jpg',
      date: new Date()
  }]
};
```

Quellcode 6: Mock-Objekt für /TS03/ (für beide Test-Frameworks)

```
import React from 'react';
import CorporateView from '../src/stp/components/views/CorporateView';
import VisibleView from '../src/stp/containers/VisibleView';
import {mount} from 'enzyme';
import { expect } from 'chai';
import renderer from 'react-test-renderer';
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import {globalState} from './constants/CorporateView'
import { Provider } from 'react-redux';

const middlewares = [thunk];
const mockStore = configureStore(middlewares);

describe('Test Suite 3', () => {
    describe('CorporateView', () => {
        describe('State - acceptTerms', () => {
            it('should be false by default', () => {
                const wrapper = mount(<CorporateView {...globalState}
            />);
                expect(wrapper.state().acceptTerms).to.equal(false);
            }),
            it('should be true after click on Checkbox', () => {
                const wrapper = mount(<CorporateView {...globalState}
            />);
                // Simuliere Klick auf Checkbox (zuvor unchecked)
                wrapper.find('Checkbox').at(1).childAt(0).simu
                late('click');
                expect(wrapper.state().acceptTerms).to.equal(true);
            }),
            it('should be false after even amount of clicks', () => {
                // Selektiere das zweite input Element in dem View
                const wrapper = mount(<CorporateView {...globalState}
                />);
                // Simuliere zwei Klicks auf Checkbox
                wrapper.find('Checkbox').at(1).childAt(0).simu
                late('click');
                wrapper.find('Checkbox').at(1).childAt(0).simu
                late('click');
                // acceptTerms sollte nun wieder false sein
                expect(wrapper.state().acceptTerms).to.equal(false);
            });
        }),
        describe('Checkbox', () => {
            it('should be unchecked by default', () => {
                const wrapper = mount(<CorporateView {...globalState}
                />);
                // Selektiere die zweite Checkbox in dem View
                // Überprüfe, ob das Prop checked den Wert false hat
                expect(wrapper.find('Checkbox').at(1).
                childAt(0).prop('checked')).to.equal(false);
            }),
            it('should be checked after first click', () => {
                const wrapper = mount(<CorporateView {...globalState}
                />);
                wrapper.find('input').at(1).simulate('click');
                expect(wrapper.find('Checkbox').at(1).
                childAt(0).prop('checked')).to.equal(true);
            });
        });
    }),
```

```
describe('ViewSelector', () => {
     it('should not add customer data nor move to next view', ()
     => {
        const store = mockStore(globalState);
        const wrapper = mount(<Provider store={store}>
            <VisibleView />
        </Provider>);
        // Simuliere Klick auf Button
        wrapper.find('button').at(1).simulate('click');
        // store.getActions() liefert alle dispatchten Actions
        als Array
        const arr = store.getActions();
        let actions = [];
        arr.forEach(function(item){
            if (item.type === 'ADD_CUSTOMER_DATA' ||
                item.type === 'SHOW_NEXT_DIALOG'){
                    actions.push(item.type);
                }
        });
        expect(actions).to.not.include('ADD_CUSTOMER_DATA',
        'SHOW_NEXT_DIALOG');
    });
    it('should add customer data on button click if checkbox is
    checked', () => {
        const store = mockStore(globalState);
        const wrapper = mount(<Provider store={store}>
            <VisibleView />
        </Provider>);
        // Simuliere Klick auf Checkbox
        wrapper.find('Checkbox').at(1).childAt(0).
        simulate('click');
        // Simuliere anschließend Klick auf Button
        wrapper.find('button').at(1).simulate('click');
        // store.getActions() liefert alle dispatchten Actions
        const arr = store.getActions();
        let actions = [];
        arr.forEach(function(item){
            if (item.type === 'ADD_CUSTOMER_DATA'){
                    actions.push(item.payload.acceptTerms);
                }
        });
        expect(actions).to.include(true);
    });
    it('should move to next dialog on button click if checkbox is
    checked', () => {
        const store = mockStore(globalState);
        const wrapper = mount(<Provider store={store}>
            <VisibleView />
        </Provider>);
        // Simuliere Klick auf Checkbox
        wrapper.find('Checkbox').at(1).childAt(0).
        simulate('click');
        // Simuliere anschließend Klick auf Button
        wrapper.find('button').at(1).simulate('click');
        // store.getActions() liefert alle dispatchten Actions
        const arr = store.getActions();
        let actions = [];
        arr.forEach(function(item){
            if (item.type === 'SHOW_NEXT_DIALOG'){
                    actions.push(item.recent);
                }
```

```
                });
                expect(actions).to.include('CORPORATE_DIALOG');
            });
        });
});
```

Quellcode 7: Komplette Implementierung von /TS03/ mit Mocha

```
import React from 'react';
import CorporateView from '../src/stp/components/views/CorporateView';
import VisibleView from '../src/stp/containers/VisibleView';
import {mount} from 'enzyme';
import renderer from 'react-test-renderer';
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import { Provider } from 'react-redux';
import * as types from '../src/stp/constants/ActionTypes';
import {globalState} from './constants/CorporateView';

const middlewares = [thunk];
const mockStore = configureStore(middlewares);

describe('Testsuite 3', () => {
    describe('CorporateView', () => {
        describe('State - acceptTerms', () => {
            it('should be false by default', () => {
            const wrapper = mount(<CorporateView {...globalState} />);
            expect(wrapper.state().acceptTerms).toBe(false);
            }),
            it('should be false after click on Checkbox', () => {
                const wrapper = mount(<CorporateView {...globalState}
                />);
                // Simuliere Klick auf Checkbox (zuvor unchecked)
                wrapper.find('Checkbox').at(1).childAt(0).
                simulate('click');
                expect(wrapper.state().acceptTerms).toBe(true);
            }),
            it('should be false after even amount of clicks', () => {
                // Selektiere das zweite input Element in dem View
                const wrapper = mount(<CorporateView {...globalState}
                />);
                // Simuliere zwei Klicks auf Checkbox
                wrapper.find('Checkbox').at(1).childAt(0).
                simulate('click');
                wrapper.find('Checkbox').at(1).childAt(0).
                simulate('click');
                // acceptTerms sollte nun wieder false sein
                expect(wrapper.state().acceptTerms).toBe(false);
            });
        }),
        describe('Checkbox', () => {
            it('should be unchecked by default', () => {
                const wrapper = mount(<CorporateView {...globalState}
                />);
                // Selektiere die zweite Checkbox in dem View
                // Überprüfe, ob das Prop checked den Wert false hat
                expect(wrapper.find('Checkbox').at(1).childAt(0).
                prop('checked')).
```

```
                toBe(false);
            }),
            it('should be checked after first click', () => {
                const wrapper = mount(<CorporateView {...globalState}
                />);
                // Simuliere Klick auf Checkbox
                wrapper.find('Checkbox').at(1).childAt(0).
                simulate('click');
                // checked sollte nun true sein
                expect(wrapper.find('Checkbox').at(1).childAt(0).
                prop('checked')).
                toBe(true);
            });
        });
      }),
      describe('ViewSelector', () => {
        it('should not add customer data nor move to next view', () => {
                const store = mockStore(globalState);
                const wrapper = mount(<Provider store={store}>
                    <VisibleView />
                </Provider>);
                // Simuliere Klick auf Button
                wrapper.find('button').at(1).simulate('click');
                // store.getActions() liefert alle dispatchten Actions
                const actions = store.getActions();
                expect(actions).toMatchSnapshot();
            }),
            it('should add customer data and move to next dialog
            on button click if checkbox is checked', () => {
                const store = mockStore(globalState);
                const wrapper = mount(<Provider store={store}>
                    <VisibleView />
                </Provider>);
                // Simuliere Klick auf Checkbox
                wrapper.find('Checkbox').at(1).childAt(0).
                simulate('click');
                // Simuliere anschließend Klick auf Button
                wrapper.find('button').at(1).simulate('click');
                // store.getActions() liefert alle dispatchten Actions
                const actions = store.getActions();
                expect(actions).toMatchSnapshot();
            });
        });
});
```

Quellcode 8: Komplette Implementierung von /TS03/ mit Jest

```
import React from 'react';
import Paper from '../src/stp/components/elements/Paper';
import { expect } from 'chai';
import ButtonCmp from '../src/stp/components/elements/ButtonCmp';
import renderer from 'react-test-renderer';
import {shallow, mount} from 'enzyme';

describe('Test Suite 3', () => {
    describe('Paper', () => {
        it('should have empty style attribute', () => {
            const wrapper = shallow(<Paper />);
```

```
            expect(wrapper.find('div').prop('style')).to.eql({});
        }),
        it('should have the style given as a prop', () => {
            const wrapper = shallow(<Paper style={{height:'900px'}} />);
            expect(wrapper.find('div').prop('style')).to.eql({
            height:'900px' });
        }),
        it('should render button component inside of div element', () =>
        {
            const wrapper = mount(<Paper><ButtonCmp type='button'
                label='Button label' /></ Paper>);
            // Überprüfe, ob in wrapper ein Knoten vom Typ 'button' exis
            tiert
            // nur dann
            expect(wrapper.find('button').exists()).to.eql(true);
        });
    });
});
```

Quellcode 9: Komplette Implementierung von /TS04/ mit Mocha

```
import React from 'react';
import Paper from '../src/stp/components/elements/Paper';
import ButtonCmp from '../src/stp/components/elements/ButtonCmp';
import renderer from 'react-test-renderer';
import {shallow} from 'enzyme';

describe('Test Suite 3', () => {
    describe('Paper', () => {
        it('should have empty style attribute if style prop is
        not defined', () => {
            const wrapper = shallow(<Paper />);
            expect(wrapper.find('div').prop('style')).toEqual({});
        }),
        it('should have the style given as a prop', () => {
            const wrapper = shallow(<Paper style={{height:'900px'}} />);
            // keine bessere Lösung zum Testen des Styles
            expect(wrapper.find('div').prop('style')).
            toEqual({height:'900px'});
        }),
        it('should render another component inside of div element', () =>
        {
             const tree = renderer.create(
            <Paper ><ButtonCmp type='button' label='Button label' />
            </Paper>
            ).toJSON();
            expect(tree).toMatchSnapshot();
        });
    });
});
```

Quellcode 10: Komplette Implementierung von /TS04/ mit Jest

```
import * as dialogFilters from '../../src/stp/constants/DialogFilters';

export const globalState = {
        customer: {
        },
        config: {
        primaryColor: 'rgb(104,178,42)',
        grayColor: 'rgb(189,189,189)',
        warningColor: '#e53020',
        timeout: '180000',
        customFont: false,
        strings: {
            contactView: {
              title:'Dürfen wir...',
              subheader:'... Dich kontaktieren?',
              buttonLabelYes:'Na klar. Meldet Euch!',
              buttonLabelNo:'Weiter ohne Daten',
              lblName:'Name',
              lblPreName: 'Vorname',
              lblSurName: 'Nachname',
              lblTel:'Telefon',
              lblMail:'Email'
            },
            corporateView: {
              title:'Super!',
              subheader:'... Nur noch zwei Fragen.',
              question:'Dürfen wir das Bild für unsere <br/>
              <b>Social-Media-Aktivitäten</b><br/>verwenden? Du kannst
              dem jederzeit widersprechen!',
              agbBefore: 'Erklärst du dich zudem mit unseren',
              agb: '</br>Allgemeinen Geschäftsbedingungen</br>',
              agbAfter: 'einverstanden? Du musst diesen zustimmen, um den
              Vorgang abzuschließen.',
              agbCheckbox: 'Ja, ich stimme den AGB zu.',
              agbTitle:'Eligibility Requirements and Terms & Condi
              tions:',
              agbText:'AGB-Text',
              corporateCheckbox: 'Ja, ihr dürft mein Foto verwenden!',
              buttonLabelYes:'Endlich Mein Phantastic Photo drucken!',
              buttonLabelNo:'Nein, Danke.'
            },
            goodbyeView: {
              title:'Super!',
              subheader:'... Es geht los!',
              printing:'Alles klar, wir drucken dein Foto.<br/>Viel Spaß
              damit!'
            },
            sharingView: {
              title:'Sollen wir...',
              subheader:'... Dir das Bild schicken?',
              buttonLabelYes:'JA! Schickt mir das Bild',
              buttonLabelNo:'Nein, Danke.',
              lblMail:'Email'
            },
            startView: {
              title:'Phantastic Photobox',
              subheader:'Wähle Dein Bild',
              anotherPic:'Oder mache ein neues <b>Phantastisches
              Photo!</b>'
            },
            errorMessages: {
```

```
                name:'Bitte gib deinen Namen ein',
                preName:'Bitte gib deinen Vornamen ein',
                surName:'Bitte gib deinen Nachnamen ein',
                phone:'Bitte gib deine Telefonnummer ein',
                email:'Bitte gib deine E-Mail-Adresse ein'
            }
        },
        followups: {
            START_DIALOG : dialogFilters.CONTACT_DIALOG,
            CONTACT_DIALOG : dialogFilters.SHARING_DIALOG,
            SHARING_DIALOG : dialogFilters.CORPORATE_DIALOG,
            CORPORATE_DIALOG : dialogFilters.GOODBYE_DIALOG,
            GOODBYE_DIALOG : dialogFilters.START_DIALOG
        }
    },
    dialog: {
        recent: dialogFilters.GOODBYE_DIALOG,
        next: dialogFilters.START_DIALOG,
    },
    photos: [{
        id: 1,
        src: '/image.jpg',
        date: new Date()}]
};
```

Quellcode 11: Mock-Objekt für /TS05/ (für beide Test-Frameworks)

```
import React from 'react';
import {expect} from 'chai';
import {mount} from 'enzyme';
import StartView from '../src/stp/components/views/StartView';
import VisibleView from '../src/stp/containers/VisibleView';
import GridListPhotos from '../src/stp/components/elements/GridListPho-
tos';
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import {Provider} from 'react-redux';
import * as types from '../src/stp/constants/ActionTypes';
import {globalState} from './constants/StartView';
import {addCustomerData} from '../src/stp/actions/customer';
import customer from '../src/stp/reducers/customer';


const middlewares = [thunk];
const mockStore = configureStore(middlewares);
const payload = { src: "/image.jpg" };

const expectedAction = { type : 'ADD_CUSTOMER_DATA',
    payload: payload };

describe('Test Suite 4', () => {
    describe('StartView', () => {
        it('should display 1 image if 1 entry is in the global state', ()
        => {
            const wipeCustomerData = function(){};
            const wrapper = mount(<StartView {...globalState}
                wipeCustomerData={wipeCustomerData} />);
            // Jede Instanz von GridTile beinhaltet ein Bild
            // 1 Instanz entspricht 1 Bild, daher:
            expect(wrapper.find('GridTile')).to.be.an('object').
            that.has.length(1);
```

```
        }),
        it('should display 4 images if 4 entries are in the global
        state', () => {
            const wipeCustomerData = function(){};
            // Erstelle Kopie von globalState, damit das Original-Objekt
            // unverändert bleibt
            const modifiedState = JSON.parse(
                  JSON.stringify(globalState));
            // Erweitere das Array photos um 3 weitere Einträge
            modifiedState.photos.push({id: 2,src: '/image2.jpg', date:
            new Date()},
            {id: 3,src: '/image3.jpg', date: new Date()},
            {id: 4,src: '/image4.jpg', date: new Date()});
            const wrapper = mount(<StartView {...modifiedState}
            wipeCustomerData={wipeCustomerData} />);
            // 4 Einträge in photos Array --> 4 Instanzen von GridTile
            expect(wrapper.find('GridTile')).to.be.an('object').
            that.has.length(4);
        });
    }),
    describe('addCustomerData action', () => {
        it('should return the expected action', () => {
            expect(addCustomerData(payload)).
            to.be.an('object').and.to.eql(expectedAction);
        });
    }),
    describe('customer reducer', () => {
        it('should return the correct initial state if no
        action is given', () => {
            expect(customer(undefined, {})).to.eql({});
        });
        it('should update the state correctly if customer data is added',
        () => {
            expect(customer({}, addCustomerData(payload))).
            to.eql(payload);
        });
    }),
    describe('ViewSelector', () => {
        it('should add the selected photo to global state', () => {
            const store = mockStore(globalState);
            const wrapper = mount(<Provider store={store}>
                <VisibleView />
            </Provider>);
            // Simuliere Klick auf Bild (GridTile)
           wrapper.find('GridTile').at(0).
                simulate('click');
            const arr = store.getActions();
            let customerData = [];
            arr.forEach(function(item){
                if (item.type === 'ADD_CUSTOMER_DATA'){
                    customerData.push(item.payload.src);
                }
            }); // Ende forEach
            expect(customerData).to.contain(payload.src);
        });
    });
});
```

Quellcode 12: Komplette Implementierung von /TS05/ mit Mocha

```
import React from 'react';
import StartView from '../src/stp/components/views/StartView';
import VisibleView from '../src/stp/containers/VisibleView';
import GridListPhotos from '../src/stp/components/elements/
GridListPhotos';
import {mount} from 'enzyme';
import renderer from 'react-test-renderer';
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import { Provider } from 'react-redux';
import * as types from '../src/stp/constants/ActionTypes';
import {globalState} from './constants/StartView';
import {addCustomerData} from '../src/stp/actions/customer';
import customer from '../src/stp/reducers/customer';


const middlewares = [thunk];
const mockStore = configureStore(middlewares);
const payload = { src: "/image.jpg" };

const expectedAction = { type : 'ADD_CUSTOMER_DATA',
    payload: payload };

describe('Test Suite 5', () => {
    describe('StartView', () => {
        it('should display 1 image if 1 entry is in the global state', ()
        => {
            const wipeCustomerData = function(){};
            const wrapper = mount(<StartView {...globalState}
                wipeCustomerData={wipeCustomerData} />);
            // Jede Instanz von GridTile beinhaltet ein Bild
            // 1 Instanz entspricht 1 Bild, daher:
            expect(wrapper.find('GridTile')).toHaveLength(1);
        }),
        it('should display 4 images if 4 entries are in the global
        state', () => {
            const wipeCustomerData = function(){};
            const modifiedState = JSON.parse(
                  JSON.stringify(globalState));
            // Erweitere das Array photos um 3 weitere Einträge
            modifiedState.photos.push({id: 2,src: '/image2.jpg', date:
            new Date()},
            {id: 3,src: '/image3.jpg', date: new Date()},
            {id: 4,src: '/image4.jpg', date: new Date()});
            const wrapper = mount(<StartView {...modifiedState}
            wipeCustomerData={wipeCustomerData} />);
            // 4 Einträge in photos Array --> 4 Instanzen von GridTile
            expect(wrapper.find('GridTile')).toHaveLength(4);
        });
    }),
    describe('addCustomerData action', () => {
        it('should return the expected action', () => {
            expect(addCustomerData(payload)).toEqual(expectedAction);
        });
    }),
    describe('customer reducer', () => {
        it('should return the correct initial state if no
        action is given', () => {
            expect(customer(undefined, {})).toEqual({});
        }),
```

```
        it('should update the state correctly if customer data is added',
        () => {
            expect(customer({}, addCustomerData(payload))).
            toEqual(payload);
        });
    }),
    describe('ViewSelector', () => {
        it('should add the selected photo to global state', () => {
            const store = mockStore(globalState);
            const wrapper = mount(<Provider store={store}>
                <VisibleView />
            </Provider>);
            // Simuliere Klick auf Bild (GridTile)
            wrapper.find('.gridTile').at(0).
                simulate('click');
            const customerData = store.getActions();
            expect(customerData).toMatchSnapshot();
        });
    });
});
```

Quellcode 13: Komplette Implementierung von /TS05/ mit Jest

Snapshots

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Test Suite 2 Button should have the label given as a prop 1`] =
`
<button
  className=""
  data-react-toolbox="button"
  disabled={false}
  href={undefined}
  onClick={undefined}
  onMouseDown={[Function]}
  onMouseLeave={[Function]}
  onMouseUp={[Function]}
  onTouchStart={[Function]}
  type="button"
>
  Button label
</button>
`;

exports[`Test Suite 2 Button should not render anything if no label is
given 1`] = `null`;

exports[`Test Suite 2 config reducer should update the state correctly if
config is changed 1`] = `
Object {
  "customFont": false,
  "followups": Object {
    "CONTACT_DIALOG": "SHARING_DIALOG",
    "CORPORATE_DIALOG": "GOODBYE_DIALOG",
    "GOODBYE_DIALOG": "START_DIALOG",
    "SHARING_DIALOG": "CORPORATE_DIALOG",
    "START_DIALOG": "CONTACT_DIALOG",
  },
  "grayColor": "rgb(189,189,189)",
  "primaryColor": "rgb(104,178,42)",
  "strings": Object {
    "contactView": Object {
      "buttonLabelNo": "Nein, danke.",
      "buttonLabelYes": "Ja, bitte.",
      "lblMail": "Email",
      "lblName": "Name",
      "lblPreName": "Vorname",
      "lblSurName": "Nachname",
      "lblTel": "Telefon",
      "subheader": "... Dich kontaktieren?",
      "title": "Dürfen wir...",
    },
    "corporateView": Object {
      "agb": "</br>Allgemeinen Geschäftsbedingungen</br>",
      "agbAfter": "einverstanden? Du musst diesen zustimmen,
      um den Vorgang abzuschließen.",
      "agbBefore": "Erklärst du dich zudem mit unseren",
```

```
      "agbCheckbox": "Ja, ich stimme den AGB zu.",
      "agbText": "<h3>1. Images (Files and Usage)</h3>
      <p>Paragraph 1</p><p>Paragraph 2</p><p>Paragraph 3</p>
      <h3>2. Cancellation of Images</h3><p>Paragraph 1</p>
      <p>Paragraph 2</p>",
      "agbTitle": "Eligibility Requirements and Terms & Conditions:",
      "buttonLabelNo": "Nein, Danke.",
      "buttonLabelYes": "Endlich Mein Phantastic Photo drucken!",
      "corporateCheckbox": "Ja, ihr dürft mein Foto verwenden!",
      "question": "Dürfen wir das Bild für unsere <br/>
      <b>Social-Media-Aktivitäten</b><br/>verwenden? Du kannst dem
      jederzeit widersprechen!",
      "subheader": "... Nur noch zwei Fragen.",
      "title": "Super!",
    },
    "errorMessages": Object {
      "email": "Bitte gib deine E-Mail-Adresse ein",
      "name": "Bitte gib deinen Namen ein",
      "phone": "Bitte gib deine Telefonnummer ein",
      "preName": "Bitte gib deinen Vornamen ein",
      "surName": "Bitte gib deinen Nachnamen ein",
    },
    "goodbyeView": Object {
      "printing": "Alles klar, wir drucken dein Foto.<br/>
      Viel Spaß damit!",
      "subheader": "... Es geht los!",
      "title": "Super!",
    },
    "sharingView": Object {
      "buttonLabelNo": "Nein, Danke.",
      "buttonLabelYes": "JA! Schickt mir das Bild",
      "lblMail": "Email",
      "subheader": "... Dir das Bild schicken?",
      "title": "Sollen wir...",
    },
    "startView": Object {
      "anotherPic": "Oder mache ein neues <b>Phantastisches Photo!</b>",
      "subheader": "Wähle Dein Bild",
      "title": "Phantastic Photobox",
    },
  },
  "timeout": "180000",
  "warningColor": "#e53020",
}
`;

exports[`Test Suite 2 integration tests should not render anything if no
button label is given after changing global state 1`] = `null`;

exports[`Test Suite 2 integration tests should render button with the de-
fault label if global state is not modified 1`] = `
<button
  className=""
  data-react-toolbox="button"
  disabled={false}
  href={undefined}
  onClick={undefined}
  onMouseDown={[Function]}
  onMouseLeave={[Function]}
  onMouseUp={[Function]}
  onTouchStart={[Function]}
```

```
  type="button"
>
  Weiter ohne Daten
</button>
`;

exports[`Testsuite 2 integration tests should render the button with new
label if global state is modified 1`] = `
<button
  className=""
  data-react-toolbox="button"
  disabled={false}
  href={undefined}
  onClick={undefined}
  onMouseDown={[Function]}
  onMouseLeave={[Function]}
  onMouseUp={[Function]}
  onTouchStart={[Function]}
  type="button"
>
  Nein, danke.
</button>
`;
```

Quellcode 14: Kompletter Snapshot für /TS02/

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Testsuite 3 ViewSelector should add customer data and move to
next dialog on button click if checkbox is checked 1`] = `
Array [
  Object {
    "type": "SERVER/GET_CONFIG",
  },
  Object {
    "payload": Object {
      "acceptTerms": true,
      "allowCorporateSharing": undefined,
    },
    "type": "ADD_CUSTOMER_DATA",
  },
  Object {
    "next": "GOODBYE DIALOG",
    "recent": "CORPORATE_DIALOG",
    "type": "SHOW_NEXT_DIALOG",
  },
]
`;

exports[`Testsuite 3 ViewSelector should not add customer data nor move
to next view 1`] = `
Array [
  Object {
    "type": "SERVER/GET_CONFIG",
  },
]
`;
```

Quellcode 15: Kompletter Snapshot für /TS03/

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Testsuite 4 Paper should render another component inside of div
element 1`] = `
<div
  className="paper"
  style={Object {}}
>
  <button
    className=""
    data-react-toolbox="button"
    disabled={false}
    href={undefined}
    onClick={undefined}
    onMouseDown={[Function]}
    onMouseLeave={[Function]}
    onMouseUp={[Function]}
    onTouchStart={[Function]}
    type="button"
  >
    Button label
  </button>
</div>
`;
```

Quellcode 16: Kompletter Snapshot für /TS04/

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Testsuite 5 ViewSelector should add the selected photo to global
state 1`] = `
Array [
  Object {
    "type": "WIPE_CUSTOMER_DATA",
  },
  Object {
    "type": "SERVER/GET_CONFIG",
  },
  Object {
    "payload": Object {
      "process": false,
      "src": "/image.jpg",
    },
    "type": "ADD_CUSTOMER_DATA",
  },
  Object {
    "next": "CONTACT_DIALOG",
    "recent": "START_DIALOG",
    "type": "SHOW_NEXT_DIALOG",
  },
]
`;
```

Quellcode 17: Kompletter Snapshot für /TS05/

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Chemnitz, den 05. September 2017

Martin Gralka