App poupée

Wintersemester 19/20
kp037
31812

Inhalt

1. Projektziel

Das Ziel bestand darin, eine voll funktionsfähige Mobile-Shop-Applikation zu entwickeln. Im Rahmen dessen, sollten die für den Nutzer, wichtigsten Funktionen implementiert werden. Darunter fallen die Anmeldung, Registrierung, Navigation, Internationalisierung, Produkte in den Warenkorb legen und kaufen.

2. Motivation

Durch mein vorangegangenes Praxissemester, wurde mir die Möglichkeit gegeben, erste Erfahrung im Bereich der App-Entwicklung zu sammeln. Da ich aber selbst nicht aktiv an der Unternehmensinternen App arbeiten durfte, hatte ich nicht die Möglichkeit mich intensiv damit auseinander zu setzen. Durch das Projekt hatte ich nun die Chance, kurz gesehenes in Wissen umzuwandeln.

Darüber hinaus wollte ich ein Produkt entwickeln, das mir für die Zukunft einen Nutzen bringen könnte. Meine Mutter erstellt in ihrer Freizeit selbstgenähte Kuscheltiere. Sollte sie sich dazu entscheiden, ihre Handarbeiten in naher Zukunft gewerblich zu vertreiben, sah ich das Projekt als Chance ihr einen kleinen Einstieg in die mobile Welt zu geben.

3. Vorbereitung

Bevor ich mit der eigentlichen Programmierung starten konnte, gab es einige Fragen zu klären und die dazugehörigen Lösungen zu finden, die ich im Folgenden erörtern werde.

Welche Zielgruppe möchte ich ansprechen?

Kuscheltiere werden in erster Hinsicht mit Kindern in Verbindung gebracht. Doch mit dieser Feststellung lässt sich erst einmal nicht viel anfangen. Denn die eigentlichen Nutzer der App, sind nicht die Kinder, die auch rechtlich gar nicht befugt wären, eigenständig App-Käufe zu betätigen. Höchstens beim durchstöbern der Produktauswahl um eine Kaufentscheidung der entsprechenden Erwachsenen zu genieren, würden sie aktiv mit der App in Kontakt treten. Sondern die eigentlichen Nutzer der App sind die Eltern, Großeltern, Bekannte und Freunde, die sich dazu entscheiden ihren jungen Liebsten ein Geschenk zu machen. Darüber hinaus, ist die Produktpalette aktuell noch sehr beschränkt, was in einigen Nutzern einen Sammelinstinkt wecken könnte.

Nach dieser Schlussfolgerung und der zusätzlichen Information, dass das durchschnittliche Alter bei der ersten Geburt 30 Jahre beträgt, entschied ich mich dazu, meine Zielgruppe, nach dem Alter, auf ü25 einzuschränken. Das Geschlecht

sollte hierbei keine weitere relevante Rolle einnehmen.

Mit dieser Überlegung konnte ich mich dann der zweiten Frage widmen.

Wie soll der User meine App wahrnehmen?

Um diese Frage zu beantworten, verglich ich erst einmal verschiedene Anwendungen. Dabei gab es viele Ähnlichkeiten. Man könnte fast sagen, dass die verglichenen Applikationen einem Muster folgen. Folgende Punkte sind mir dabei aufgefallen:

Die Farben werden schlicht gehalten. Es gibt keine großen farblichen Experimente. Meist wird eine einheitliche Farbe für die Hauptelemente benutzt (wie dem Header) und ansonsten wird mit viel weiss und grau gearbeitet. Natürlich werden auch wichtige Signalfarben eingesetzt, wie z.B. rot bei Rabatten. Aber das Grundgerüst ist sehr einfach. Umgesetzt habe ich dieses Prinzip auch in meiner App. Wenig Farben machen meines Erachtens nach die App übersichtlicher. Da ich bei meiner Zielgruppe auch Personen höheren Alters anspreche, ist es dementsprechend auch von Nutzen, nicht zu viele visuelle Anreize zu liefern. Das erhöht die Benutzerfreundlichkeit, Bedienbarkeit und die Zufriedenheit.

Als Navigationselement hat sich das Burger Menü durchgesetzt. Ein Menü hat viele positive Eigenschaften. So lassen sich alle notwendigen und möglichen Interaktionen auf einen Blick anzeigen. Man verliert sich nicht in irgendwelchen Pages und findet sehr einfach zum gewünschten Ziel. Deshalb ist auch ein solches Menü unabdingbar für meine App.

Produkte werden strukturiert dargestellt. Suchleisten, Filter, Kategorien und Produktlisten führen dem Nutzer schnell zum gesuchten Produkt. In meiner App, habe ich aufgrund der begrenzten Produktpalette auf das meiste hier erwähnte verzichtet.

Als persönliches Fazit meiner kleinen Marktanalyse, habe ich mich an zwei Begriffen orientiert: schlicht und einfach. Der Nutzer soll einfach durch die App navigieren können, und in jedem Moment verstehen können, wie er die Applikation zu bedienen hat. Wenig ist in diesem Fall mehr.

Welche Funktionen sollen dem User zur Verfügung gestellt werden?

In erster Linie sollte die App eine Verkaufsplattform werden. Dazu gehören die Basisfunktionen: Anmeldung, Registrierung, Produkte in den Warenkorb legen, Produkte kaufen und Profil bearbeiten.

Es gab viele weitere denkbare Funktionen, wie einen Blog, ein Choice-Board, eine Tauschbörse neben dem Shop um Kunden einen Anreiz für die App zu bieten, obwohl sie kein Interesse an den Produkten haben.

Umgesetzt wurde von allen Ideen leider nur ein Bruchteil. Der Grund dafür, bestand darin, dass mein Teampartner, dem einige Aufgaben zugesprochen wurden, im letzten Moment von dem Projekt zurücktrat. Das veranlasste mich dann dazu angefangene Aufgaben fallen zu lassen und mich auf ein zuerst einmal lauffähiges Programm zu begrenzen.

Welche Produkte möchte ich verwenden?

Wie anfangs schon erwähnt, besteht die Produktpalette aus den selbst hergestellten Kuscheltieren meiner Mutter. Für anfängliche Tests verwendete ich Bilder aus dem Internet. Für das finale Ziel wollte ich aber tatsächliche Produktbilder verwenden. Deshalb veranstaltete ich ein kleines Fotoshooting und setzte die Produkte in Szene. Hierbei achtete ich auch darauf, dass der Hintergrund farblich mit meinem Design übereinstimmt.

Des Weiteren wollte ich individuelle interessante Texte für jedes einzelne Produkt. Die Idee kam mir durch das Smoothie Produkt der Marke True Fruits, die für jede einzelne Sorte einen produktbezogenen, witzigen Text verfasst. Das veranlasst den Käufer nämlich die Produktbeschreibung tatsächlich zu lesen und sich im besten Fall damit zu identifizieren. Außerdem gibt es dem Kunden die Illusion, dass es sich um ein qualitativ abgrenzendes und individuell wertvolles Produkt handelt.

Welche Tools brauche ich für die Umsetzung?

Für die Umsetzung wollte ich für das Front-End unbedingt **react-native** benutzen. Das lag daran, dass in meinem Praxissemester für die Unternehmensinterne App dieses Framework benutzt wurde. Und ich wollte mich speziell in dieses Framework einarbeiten. Darüber hinaus gibt es viele weitere bekannte Apps die mit **react-native** geschrieben wurden. Wie z.B. Facebook

Für die UI-Elemente habe ich mich für **NativeBase** entschieden. Für meine Zwecke ein absolut ausreichendes Tool und sehr einfach zu benutzen.

Um das Back-End zu realisieren fiel meine Wahl auf **Node.js**

Die Datenbank sollte mit **MongoDB Atlas** implementiert werden.

Darüber hinaus arbeitete ich für das Farbdesign mit dem **Color Tool** von Material-UI.

Auf die einzelnen technischen Umsetzungen werde ich in den nächsten Kapiteln

genauer eingehen.

Wie soll das Projekt aufgeteilt werden?

Das Projekt sollte grundsätzlich in zwei Aufgabenbereiche gesplittet werden – Front- und Back-End. An diese Aufteilung sollte man sich aber nicht strikt halten müssen. Programmieren geht zusammen besser. Pair-Programming hat nämlich mehrere Vorteile. Zum einen kann man gemeinsam an einem Problem arbeiten. Das spart Zeit und nicht immer kommt man allein auf eine Lösung. Zwei Köpfe sind nun mal produktiver als einer. Zum anderen lernt man selbstverständlich mehr. Schneller auf eine Lösung kommen, bedeutet, dass man sich auch schneller auf ein anderes Problem konzentrieren kann.

4. Programmierung

Im Gegensatz zu den vorherigen Kapiteln wird bei den nachfolgenden nur der Ist-Zustand und nicht der Soll-Zustand betrachtet.

4.1 Front-End

4.1.1 React-Native

React-Native ist ein von Facebook entwickeltes Framework. Mithilfe dieses Frameworks lassen sich plattformübergreifend native und performante Applikationen für Android **und** iOS entwickeln (ich habe nur für Android entwickelt – für iOS braucht man entsprechende zusätzliche Hardware).

Die Installation und das Anlegen eines Projektes sind kein Hexenwerk und mit der nötigen Konzentration auch schnell zu bewerkstelligen. Um schlussendlich an der App programmieren zu können, kann man neben einem Emulator auch sein eigenes Mobiltelefon verwenden (was ich gemacht habe).

Die grundlegende Struktur lebt von Komponenten. Diese Komponenten können mithilfe verschiedener Parameter angepasst werden. Diese Parameter werden in **React-Native** `props` genannt. Bspw. könnte das so aussehen:

```
let pic = { uri: […] };
<Image source={pic} />
```

In diesem Beispiel wäre die Property source. Die Variable pic wird mit {} in JSX eingebettet. Innerhalb dieser Klammern können beliebige **JSX expressions** stehen.

Neben den Propertys, die von ihren Eltern gesetzt werden und während dem Lebenszyklus der Komponente fix bleiben, gibt es noch den `state`. Auf `state` wird

zurückgegriffen, wenn Daten verändert werden sollen.

Mit diesen einfachen Basics, steht dem Einstieg in **React-Native** nichts mehr im Weg.

4.1.2 NativeBase

NativeBase ist eine kostenlose und OpenSource UI-Komponenten Bibliothek für React-Native. Die Dokumentation und die gestellten Komponenten werden in der Dokumentation übersichtlich dokumentiert und erleichtern die Gestaltung.

In meinem Projekt, habe ich oft auf Komponenten von **NativeBase** zurückgegriffen.

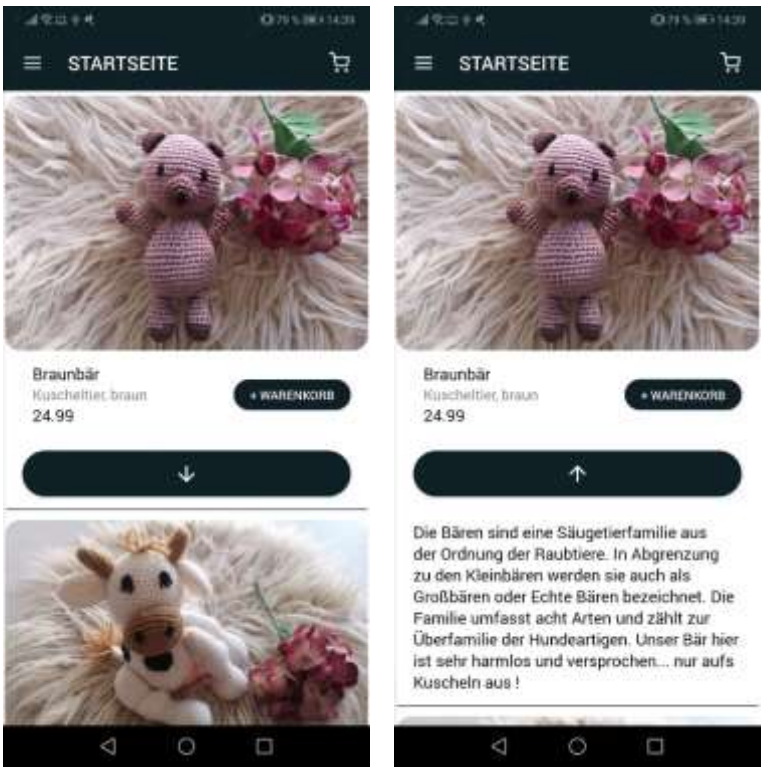
4.1.3 Screens

Um zu starten, brauchte ich erst einmal einen groben Plan, welche Screens ich in meine App implementieren möchte. Um dem Nutzer alle nötigen Funktionen zur Verfügung zu stellen, entschied ich mich folgende Screens zu erstellen:

1. Login
2. Signup
3. Home
4. Profile
5. Cart
6. Orders

Weitere intensivere Überlegungen gab es hier nur bei der Darstellung der Produkte. Auf dem Homescreen werden die jeweiligen Produkte angezeigt. Nun stellte ich mir folgende Frage: sollten die ProductDetails einen eigenen Screen bekommen, oder könnten sie auch übersichtlich auf dem Homescreen angezeigt werden. Das bedeutet, entweder ich klicke auf ein Produkt und werde an einen weiteren Screen weitergeleitet, der mir die nötigen Details, wie Produktbeschreibung etc. anzeigt, oder ich lasse mir die zusätzlichen Informationen direkt auf dem Homescreen anzeigen.

Nach reifer Überlegung, entschied ich mich dazu, die Produktinformationen auf dem Homescreen anzeigen zu lassen. Für größere Anwendungen und großen Produktpaletten, ist es sicherlich sinnvoll ein ProductDetail Screen zu erstellen. Auch weil es meist viel mehr relevante Informationen gibt, die eine Darstellung auf einem Screen unübersichtlich erscheinen lassen. Wie z.B. Nutzerbewertungen, mehrere Bilder etc. In meiner App gibt es diese zusätzlichen Funktionen nicht. Deshalb fand ich es optisch schöner, die Produktbeschreibungen per Toggle direkt in der Produktkarte anzeigen und verstecken lassen zu können.



Homescreen

4.1.4 Navigation

Für die User-Usability ist es eine einwandfrei funktionierende und übersichtliche Navigation Voraussetzung. Für die Umsetzung nahm ich zwei verschiedene Pakete unter die Lupe, **react-native-router-flux** und **react-navigation**.

React-navigation ist die Native Lösung zur Navigation und wohl das bekannteste und meist angewendete Package für diese Aufgabe. Wohingegen **react-native-router-flux** nur der Wrapper um **react-navigation** ist.

Nachdem ich beides angewendet hatte, habe ich mich schlussendlich für **react-native-router-flux** entschieden. Es ist einfacher zu implementieren und hat für meine Zwecke völlig ausgereicht. Ein großer Nachteil ist aber die ziemlich schlechte Dokumentation. Mit **react-native-router-flux** lassen sich alle Routen an einem zentralen Ort definieren:

```
//Routes.js
import Login from "../screens/Login"
export default class Routes extends Component{
   render(){
      return(
         <Router>
            <Scene
               key="root" hideNavBar={true}
               initial={!this.props.isLoggedIn}      //1
            >
               <Scene
```

```
                        key="login"                          //2
                        component={Login}
                        initial={true} />
                    </Scene>
                    <Scene
                        key="signup"                         //2
                        component={Signup}
                    />
                    </Scene>
                <Router />
            );
        }
    }
```

Interessant ist hier vielleicht der Schritt [1]. Hier wird die Property `!this.props.isLoggedIn` definiert. Ist der Nutzer nicht eingeloggt, hat er keine Möglichkeit, den Kern der App zu betreten, sondern befindet sich in dieser Scene. Sollte sich der Nutzer erfolgreich eingelogged haben, kann er „App-Scene“ betreten.

Die weitere Navigation zwischen den Screens ist jetzt nur noch ein Kinderspiel. In [2] wird für den key der Wert `"login"` definiert. Diesen Key können wir dann in den anderen Komponenten benutzen, um auf den gewünschten Screen zu navigieren:

```
//Login.js
import { Actions } from "react-native-router-flux"
signup() { Actions.signup() }
<TouchableOpacity onPress={this.signup}>
```

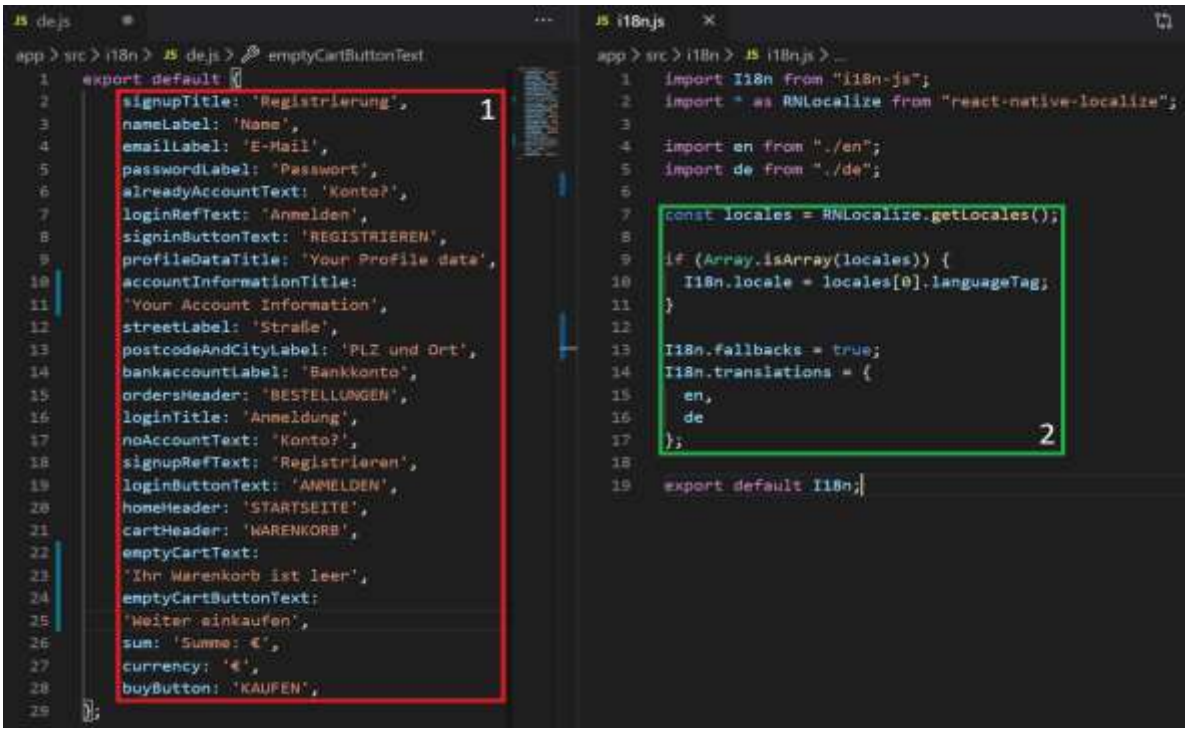
Für die Hauptnavigation habe ich einen **Drawer** aus **NativeBase** verwendet. Dieser lässt sich mit Klick auf ein Menü-Icon öffnen. Darin befinden sich alle Links für die relevanten Screens. Das Menü selber habe ich in der Datei **Menu.js** erstellt und nach meinen Wünschen gestaltet. Um das Menü in jedem Screen (außer Login und Signup) verwenden zu können, muss der **Drawer**
quasi als Container um den Content gelegt werden.

4.1.5 Internationalisierung

Soll der Verkauf der einzelnen Produkte global vonstattengehen, ist eine Internationalisierung der Anwendung Pflicht. Aber auch wenn die Produkte nur im Inland vertrieben werden sollen, sind durch die Vielfalt an Sprachen, Übersetzungen ein weiterer wichtiger Erfolgsfaktor, um den unterschiedlichen Nutzern, die Bedienung der App, so leicht und verständlich wie möglich zu machen.

Um die App International nutzen zu können, entschied ich mich für zwei Sprachen: Deutsch und Englisch. Der Hauptgrund dafür, ist wohlgemerkt, dass ich selber nur drei Sprachen beherrsche. Um semantisch und syntaktisch korrekt in eine andere Sprache zu übersetzen, bräuchte es einen professionellen Dolmetscher.

Um eine Applikation Mehrsprachen-tauglich zu machen, bedarf es nur sehr weniger Schritte.



Internationalisierung

Zur Realisierung der Internationalisierung, installierte ich zwei Pakete: 1. **i18n-js** und 2. **react-native-localize.**

Nach der Installation, habe ich eine JS-Datei **i18n.js** erstellt, in der ich neben, den eben erwähnten Modulen, auch die jeweiligen Übersetzungen importiere (Im Abschnitt 1 sind die Deutschen Übersetzungen zu sehen). Anschließend passieren im 2.ten Bereich der Abbildung zwei Dinge. Als erstes werden die aktuellen Locales ermittelt. Das dient dazu, die in den Handyeinstellungen definierte Sprache zu identifizieren und die dafür entsprechende Übersetzungsdatei zu laden. Im zweiten Schritt werden die importierten Übersetzungsdateien für die Nutzung hinzugefügt.

Um die Übersetzungen jetzt in anderen Dateien nutzen zu können, muss nur noch die Datei **i18n.js** importiert werden und kann mit folgendem Code `{I18n.t('')}` benutzt werden. Dabei steht das t für translations und der jeweilige key (name der Übersetzung) wird in die Hockommatas geschrieben.

4.1.6 Redux

Eine einheitliche Architektur, kann im Anwendungsbereich der Übersichtlichkeit und Wartbarkeit dienen. **Flux** ist eine solche Architekturidee. Und um diese Architekturidee zu implementieren, wurde **Redux** ins Leben gerufen. Im Folgenden möchte ich mithilfe meines Anwendungsbeispiels **loginUser**, auf die Funktionsweise

von Redux eingehen.

4.1.6.1 Actions

Redux ist ein vorhersehbarer Statuscontainer für JavaScript Apps. Die Idee ist es, den Zustand nicht den in den einzelnen Komponenten zu halten, sondern für diese Aufgabe eine zentrale Stelle zu definieren. Um das zu realisieren, habe ich das Modul **auth.actions.js** erstellt. In diesem Modul befinden sich alle notwendigen Actions um einen User erfolgreich einzuloggen. **Actions** werden in **Redux** als **Objekte** definiert, die zumindest aus einem Property mit dem Namen type bestehen um den Typ der **Action** festzulegen.

```
//auth.actions.js
export const loginUser = (payload) => {
   return async (dispatch) => {
      try{
         dispatch({type: "LOGIN_USER_LOADING"});
         const response = await fetchApi(...);
      catch(error){
         […]
      }
   }
}
```

4.1.6.2 Reducer

Im zweiten Schritt mussten die dazugehörigen **Reducer**, die dafür verantwortlich sind, einen Zustand in einen neuen zu überführen, erstellt werden. **Reducer** erhalten als Eingabeparameter den **aktuellen Zustand** und die **Action** die gerade ausgelöst wurde und erzeugen daraus eine Ausgabe.

```
//auth.reducer.js
const loginUser = (state={}, action) =>              //1
   switch(action.type){
      case "LOGIN_USER_LOADING"
         return{
            isLoading: true,
            isError: false,
            isSuccess: false,
            errors: null
         }
[…]
      default:
         return state;
   }
}
```

In der ersten Zeile wird der **Reducer** als Funktion definiert und bekommt hier den bisherigen Zustand (state) und eine Action (action) als Parameter übergeben. Aus diesen Parametern wird ein neuer Zustand erstellt und zurückgegeben.

4.1.6.3 Store

Der neue Zustand wird bei **Redux** in einem zentralen Store verwaltet. Der zentrale Store selber wird mit der Funktion **`createStore`** erzeugt, welche als Parameter eine **Reducer-Funktion** erwartet. Um mehrere Reducer zusammenzufassen und der Funktion `createStore` zu übergeben, werden Reducer mit der Funktion **`combineReducer`** zusammengefasst.

```
//store.js
import reducers from "../reducers"
const persistConfig = {
   key: "root",
   storage: AsyncStorage,
   whitelist: ["authReducer", …]
}
const persistReducer = persistReducer(
   persistConfig, reducer
);
export default () => {
   let store =
      createStore(persistReducer, {}, applyMiddleware(thunk)
   );
   return { store, … }
}

//auth.reducer.js
[…]
export default combineReducer({
   createUser, loginUser, authDate
});
```

Da der Zustand sich nun nicht mehr in der Komponente selber befindet, sondern in den **zentralen Store** ausgelagert wurde, müssen wir dafür sorgen, dass unsere Komponente darauf zugreifen kann. Im ersten Schritt [1] wird der Zustand aus dem zentralen Zustand der Komponente als **Property** übergeben.

```
//Login.js
const {loginUser} = this.props;                    //1
[…]
mapStateToProps = (state) => ({                    //2
   loginUser: state.authReducer.loginUser
});
mapDispatchToProps = (dispatch) => ({dispatch});   //3
```

Als nächstes wird festgelegt, welcher Teilzustand für die Komponente relevant ist [2]. Im letzten Schritt [3] müssen wir noch die Verteilung einer **Action** an alle Reducer gewährleisten. Das ist notwendig, da eine Action selbst, an sich, noch nichts bewirkt. Erst nachdem sie durch die **dispatch-Methode** geleitet werden, kann sie an alle Reducer verteilt werden und eine Zustandsänderung bewirken.

4.1.6 Redux-Form

Die Implementation und Benutzung von **redux-form** ist sehr einfach. Ich möchte an dieser Stelle kurz die Funktionsweise und Nutzung erläutern.

```
//Login.js
import { Field, reduxForm } from "redux-form"        //1
[…]
renderTextInput = (field) => {                       //2
   […]
}
[…]
<Field                                               //3
  name="password"
  placeholder={I18n.t('passwordLabel')}
  secureTextEntry={true}
  component={this.renderTextInput}
/>
[…]
const validate = (values) => {                       //4
   const error = {};
   if(!values.password){
      error.password = "Password is required"
   }
   return error;
}
export default compose(                              //5
   reduxForm({ form: "login", validate })
)(Login)
```

In der ersten Zeile importieren wir **Field** und **reduxForm**. In [2] definieren wir eine `renderTextInput` Methode. Das ist notwendig, da ich in einer anderen Datei **InputText.js** meinen InputText schon definiert habe, um sie nach meinen Wünschen anpassen und stylen zu können. Die Komponente übergebe ich dann in [3] einer **Field** Komponente, damit meine Custom TextInput Komponente auch gerendert wird. Die **Field** Komponente wird von **redux-form** zur Verfügung gestellt und hat die Aufgabe jeden Input mit dem Store zu verbinden. Schritt [4] hat die Aufgabe meine Eingaben zu validieren. In diesem Beispiel, wird bei einem leeren **TextInputField** ein Warntext `"Password is required"` angezeigt. Abschließend wird im letzten Schritt [5] wird die Komponente mit `reduxForm` gewrapped, damit die Komponente mit dem Store kommunizieren kann. Außerdem stellt es die Propertys über den Formzustand und die Funktionen um einen Submit zu handeln, zur Verfügung.

4.2 Back-End

4.2.1 Node.js

Node.js ist eine JavaSript-Laufzeitumgebung und beinhaltet alles, was man zur Umsetzung für eine Applikation benötigt. Mit dem dazugehörigen Paketmanager lassen sich verschiedenste benötigte Pakete mit `npm install` schnell und einfach installieren. Des Weiteren erweitert das Open Source-Framework **Express.js**, den mithilfe des http-Moduls erzeugten Webserver um zusätzliche Features wie Routing und Plugin-Infrastruktur.

Um einen Webserver aufzusetzen, bedarf es nur weniger Codezeilen:

```
const express = require("express");
const app = express();

app.listen(3000, () => {
   console.log("Server is running on port 3000");
});
```

Ich denke der Codeschnipsel ist hier selbsterklärend.

4.2.2 MongoDB Atlas

Im nächsten Schritt habe ich einen `userController` erstellt. Hier befinden sich alle für das Projekt relevanten Routen. Z.B. wird beim Submit der Login Credentials die Route `/login` aufgerufen:

```
router.post("/login", (req, res) => {[…]});
```

Anschließend müssen meine Daten irgendwo gespeichert und verwalten werden. Hier kommt **MongoDB Atlas** zum Einsatz. **MongoDB Atlas** ist ein **Database as a Service** Angebot. Nutzer können damit ihre Datanbankanwendungen in Amazon Web Services (AWS) umsetzen – kostenlos.

Um Daten in die MongoDB Atlas Datenbank hinzufügen und abrufen zu können, habe ich das Framework **mongoose** verwendet. Damit wir hier mit MongoDB arbeiten können, müssen einige Schritte beachtet werden. Zuerst müssen wir die Form unserer Daten definieren (das **Schema**). Danach muss eine Verbindung zur Datenbank hergestellt, unsere Abfragen ausgeführt und die Verbindung geschlossen werden. Die jeweiligen Codezeilen lassen sich gut aus dem Projektcode entnehmen, deswegen werde ich hier nicht weiter darauf eingehen.

Um dem User die entsprechenden Funktionalitäten zu gewährleisten, habe ich einige Methoden geschrieben. Bspw. wird beim Login erst einmal der zu den

eingegebenen Credentials passende User gesucht. Stimmen die E-Mail-Adresse und das Passwort überein, wird für den User ein Token generiert, der wieder weggenommen wird, sollte sich der User ausloggen. Darüber hinaus wird das Passwort gehashed um eine gewisse Sicherheit zu gewährleisten. Weitere Funktionen können im Code eingesehen werden.

5. Fazit

Das Projekt selber hat mir sehr viel Spaß gemacht. Durch die intensive Einarbeitung in ein neues Thema, speziell in das Framework **React-Native** hatte ich die Möglichkeit mir umfangreiche Basics in diesem Gebiet anzueignen. Neben vielen Erfolgen, gab es aber auch viele Komplikationen (was bei der Software-Entwicklung üblich ist).

Einer der Hauptkomplikationen bestand darin, dass mein Teampartner im letzten Moment von dem Projekt zurückgetreten ist. Ein voll funktionsfähiger Mobile-Shop erfordert einen enormen Zeitaufwand – da waren zwei Personen schon sehr gering angesetzt. Alleine ist es kaum zu bewältigen. Die verschiedenen Aufgaben waren verteilt, umgesetzt wurde aber nur etwas meinerseits. Ich ging eigentlich davon aus, dass mein Team-Partner seine Aufgaben umsetzen würde und half ihm wo ich konnte, bei Software-Problemen und auch mit Hardware half ich ihm aus. Am Ende hieß es dann nur, dass er es nicht geschafft hat, auch nur eine Zeile Code umzusetzen. Das hat mich sehr unter Zeitdruck gesetzt und ich musste angefangene Aufgaben (wie eigene Ads hinzuzufügen) hintenanstellen und schauen, dass ich überhaupt ein lauffähiges Programm entwickle und musste mich alleine dann an das Back-End setzen. Das entstandene End-Produkt ist nun ein sehr einfacher Mobile-Shop in dem nicht annähernd alle Produktideen umgesetzt wurden. Hauptsächlich bin ich aber nicht enttäuscht, dass das Endprodukt nicht meinen Wünschen entspricht, sondern dass ich nicht die Möglichkeit hatte, mehr zu lernen. Durch Pair-Programming lässt sich nun mal ganz anders an ein Problem herantreten. Lösungsvorschläge sind von mehreren Personen immer sinnvoller, als nur von einer. Ich bin kein sonderlich guter Back-End Programmierer, hätte aber persönlich Spaß dran gehabt, mich intensiver mit dieser Materie auseinander zu setzen.

Nichts desto trotz, kann ich behaupten, dass ich für mich selbst sehr viel aus dem Projekt mitnehme. Das Grundgerüst einer soliden App-Entwicklung ist geschaffen und ich kann mich auch privat in diesem großen Gebiet weiterentwickeln.

Probleme hatte ich auch mit der Installation der Packages, hauptsächlich mit **react-navigation**. Erstens ist es nicht einfach zu installieren. Und ist es einmal installiert, blockiert es viele andere Packages, was ich auf die harte Tour lernen musste. So musste ich nicht nur einmal das Projekt neu aufsetzen. Dabei ist es extrem Zeitraubend alle Packages wieder zu installieren (meine Hardware ist nicht die

beste).

Außerdem müssen bei jedem App-Start der Metro-Server und die Anwendung gestartet werden. Das kann schon mal bis zu fünf Minuten dauern. Muss man die App mehrere Male Neustarten, kann das schon sehr frustrierend sein, denn man muss teilweise mehr warten, als dass man programmieren kann.

Unterschätzt habe ich auch den Zeitaufwand für das Design. Auf den ersten Blick lässt sich der Schein erwecken, dass mit Bibliotheken wie **NativeBase** die nötigen Anforderungen in Rekordzeit umgesetzt werden können. Praktisch ist man grob gesagt mehr damit beschäftigt nach den richtigen Icons zu suchen und Farben auszuwählen, als die Funktionalitäten selbst zu implementieren.

Ich würde so ein Projekt jederzeit wieder durchführen, vorausgesetzt dass mir ein solides Team als Unterstützung zur Seite steht. Nicht vielleicht um den Zeitaufwand zu minimieren, sondern um mir persönlich auch das Wissen aneignen zu können, dass ich mir als Ziel gesetzt habe. Natürlich machen manche Aufgaben mehr Spaß als andere – keine Frage. Und das rumwühlen in Dokumentationen kann sehr nervenaufreibend sein. Aber zumindest sollte man die Möglichkeit haben, sich mit den Themen, die einem Spaß machen, richtig und lückenlos auseinander setzen zu können.