React

Ali Yalama

February 2023

1 Einrichten eines React Projekts

Ich setze Kenntnis mit anderen JS Frameworks (oder wenigstens JS) voraus.

1.1 Erste Schritte

Du startest deine Anwendung (standardweise) mit

```
npm run start
```

Wir gehen davon aus, dass du bereits eine react-Anwendung erstellt hast. Du hast mindestens 4 Dateien: App.js und css als auch index.js und css.

Was machen die Dinger?

App.js ist im Grunde nichts anderes als eine Komponente. Wir wollen uns Zuerst auf index.js konzentrieren. Die App.js kannst du theoretisch auch umändern.

1.2 Die index.js

Der Inhalt sieht so aus:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Wir nutzen dabei die React-API. Mittels ReactDOM und der Methode render wird die Methode gerendert. Was uns hier eigentlich juckt ist, dass die Wurzel (root) der Seite geholt wird und da die React-Anwendung reingepackt wird. Du hast hier StrictMode von React als Komponente. Damit werden Probleme mit der Anwendung (in der Konsole?) angezeigt. Lass den scheiss einfach so und benutz App.js als einstiegspunkt.

1.3 Die App.js

Das ist deine Erste richtige Komponente. Hier würdest du eigentlich dann anfangen deine App zu schreiben. Gucken wir einfach mla rein:

```
import './App.css';

function App() {
  return (
    <div className="App">
      <p>Vergil status</p>
    </div>
  );
}

export default App;
```

Was passiert hier? Wir definieren unsere Komponente als Funktion und Exportieren diese. Die Funktion hat als Rückgabewert JSX-Code. Alles was wir an berechnungen durchführen, wird vor dem return befehl gemacht.

Aber fass die App.js auch nicht weiter an, ausser du musst. Denk dir einfach eine Hierachie von Komponenten aus. Die Top-level Komponenten fliegen als Kinder von App.js in die App-Komponenten rein. Oder du schreibst für die nen extra parent und die fliegt alleine in die App.js rein.

2 Schreiben unserer Ersten Komponente + Databinding

Wir schreiben hier unsere erste Komponente. Wir wollen da auch Databinding betreiben, d.h. Variablen einbinden. Im Folgenden installieren wir auch bootstrap.

2.1 Das Grundgerüst

Wir erstellen eine .js Datei. Du kannst css dateien Importieren als import. Wir haben noch keine. Das erste was wir schreiben ist die Function. Die benennung ist Wichtig: Bennene sie so wie du sie exportieren willst.

```
function MyFirstComponent() {
    return (
        <div>
        <p>Hier steht gleich was anderes</p>
        </div>
    );
}

//du referenzierst die Bennenung deiner Methode
export default MyFirstComponent;
```

2.2 Daten und Methoden

Wir können im Grunde Variablen definieren auf die normale JS art und weise. Aber ich machs anders. Siehe nächstes Kapitel Ich lass mal Redux hier erstmal aussen vor. Methoden definieren ist Kacke auf diese ARt und weise, da sie wie variablen aussehen.

```
function MyFirstComponent() {

    const [myNum, setMyNum] = useState(0);

    /*Methodendefinition in react auf Funktionale art und weise ist
         richtig behindert
    In die Lambda parameter gehen deine Parameter rein*/
    const increaseNum = () => {
        setMyNum(myNum+1);
    }

    const decreaseNum = () => {
        setMyNum(myNum-1);
    }

    return (
    <div>
        <div className="container">
            <div className="text-center p-4 p-lg-5">
                <h1 className="fw-bold mb-4">{myNum}</h1>
                <button className="btn btn-primary fs-5 me-2 py-2
                    px-4"
                        type="button"
                        onClick={() => increaseNum()}
                >Increase</button>
                <button className="btn btn-light fs-5 py-2 px-4"
                        type="button"
                        onClick={() => decreaseNum()}
                >Decrease</button>
            </div>
        </div>
    </div>
);
}
```

In die Klammern kommen Parameter rein. Die definition von methoden in einer Funktion ist bäh.

2.3 useState-Hook und warum ich die Variablen so geschrieben habe

Was passiert hier? Du definierst hier eine Variable und einen Setter gleich mit dazu. Mit useState hast du eine React-Hook. Alle Komponenten, die die Variable nutzen, werden refreshed, wenn diese verändert werden. Ich werde später auf Hooks eingehen. Ich nutze das damit Änderungen am Wert in der UI sichtbar werden

2.4 Warum sehen onclicks so scheisse aus?

Damit sie nicht bei laden gefuert werden und so funktionieren wie sie sollen. Sieht in anderen JS-Frameworks nicht so aus. Egal. Ist halt so.

3 Props

Damit übergibst du Daten in eine Komponente. Schreiben wir also eine Komponente, die Props annehmen soll. Das entscheidende hier ist, dass du in die Definition der Funktion, die die Komponente abbildet, du einen Parameter angibst. Nennen wir diesen props.

Wir definieren erstmal in der Komponente vom vorigen Kapitel noch ne Variable, die wir später übergeben.

```
1     const [message, setMessage] = useState("Nothing changed");
2
3     /*Methodendefinition in react auf Funktionale art und weise ist
          richtig behindert
4     In die Lambda parameter gehen deine Parameter rein*/
5     const increaseNum = () => {
6         setMyNum(myNum+1);
7         setMessage("Increased");
8     }
9
10    const decreaseNum = () => {
11        setMyNum(myNum-1);
12        setMessage("Decreased");
13    }
```

Dann definieren wir eine neue Komponente. Denk dran, dass du props in die Methodensignatur der Komponente gibst.

```
function MyFirstComponent(props) {
    return (
        <div>
            <p>{props.msg}</p>
        </div>
    );
}
export default MyFirstComponent;
```

Listing 1: Hast du jetzt was spannendes erwartet

Das mitzugebende Prop soll also msg heissen. Das gute: Keine weitere vorbereitung nötig. Wir können als Attribut dann einfach angeben, welcher wert mitgegeben werden soll und wie der heißen soll. Wir gehen zurück zur Mutterkomponente und binden das ein.

```
<div className="container">
    <div className="text-center p-4 p-lg-5">
        <h1 className="fw-bold mb-4">{myNum}</h1>
        <button className="btn btn-primary fs-5 me-2 py-2 px-4"
                type="button"
                onClick={() => increaseNum()}
        >Increase</button>
        <button className="btn btn-light fs-5 py-2 px-4"
                type="button"
                onClick={() => decreaseNum()}
        >Decrease</button>
    </div>
    {/* Du brauchst keine Variablen fuer Props vordefinieren
     Gib einfach ein, wie der Kenner fuer den Wert ist und
     Gib ihm die Nachricht
     */}
    <KomponenteMitProp msg={message}></KomponenteMitProp>
</div>
);
```

4 Databinding

Es gibt ein Haufen wege das zu machen. Ich will hier zwei dinge haben:

1. Bei ändern des wertes via UI, soll sich der Wert der gebundenen Variablen ändern
2. Wird der Wert ohne Eingabe (über UI) gesetzt, soll sich der Wert, der im Input-Feld steht, ändern

Eine Implementierung kann wie folgt aussehen:

```
function InputKomponente() {

```

```
    const[myTextInput, setMyTextInput] = useState('Default Val');
    const[myBooleanInput, setMyBooleanInput] = useState(false);

    const handleTextChange = (val) => {
        setMyTextInput(val);
    }

    const flipBoolean = () => {
        setMyBooleanInput(!myBooleanInput);
    }

    return (
        <div>
            <div className="input-group mb-3">
                <input type="text" className="form-control" value={
                    myTextInput} aria-label="Username"
                    onChange={(event) => handleTextChange(event.
                        target.value)}
                    aria-describedby="basic-addon1"/>
            </div>
            <div className="form-check">
                <input className="form-check-input" type="checkbox"
                    onChange={() => flipBoolean()}
                    checked={myBooleanInput} id="
                        flexCheckChecked" />
            </div>
        </div>
    );
}
```

Was passiert hier? Wir haben hier wieder zwei Variablen. Wir definieren ebenfalls methoden, die gefuert werden, wenn sich der wert ändert - siehe onChange - das funktioniert nach demselben prinzip wie onclick - als Event. Den Wert holen wir uns aus dem Event. Wenn da mal der Wert nicht extrahiert wird, weil z.B. die falschen Felder angesteuert werden, debug das. Um die Werte auch zu binden, nehmen wir bei Texten value und bei Checkboxen checked. Die Methode flipBoolean ist eher eine Stütze, da das rumgebuggt hat.

5 Konditionales Rendering

Was in Angular ngIf und in Vue v-if ist, ist in JSX mit React ein Bisschen komischer. Wir gehen einfach mal davon aus, dass du irgendwo schon variablen definiert hast. Nehmen wir das aus dem vorigen Kapitel.

```
return (
    <div>
        <div className="input-group mb-3">
            <input type="text" className="form-control" value={
                myTextInput} aria-label="Username"
                onChange={(event) => handleTextChange(event.
                    target.value)}
                aria-describedby="basic-addon1"/>
        </div>
        <div className="form-check">
```

```
            <input className="form-check-input" type="checkbox"
                   onChange={() => flipBoolean()}
                   checked={myBooleanInput} id="flexCheckChecked"
                       />
        </div>
        {myBooleanInput && <p> Box value true </p>}
    </div>
);
```

Du verknüpfst also UI mit Wahrheitswerten und AND-Operator.

6 Iteration in der UI

```
const myMap = [{key0: "val0"}, {key1: "val1"}, {key2: "val2"}]

const myArr = ["elem0", "elem1", "elem2"];

return (
    <div>
        <div className="input-group mb-3">
            <input type="text" className="form-control" value={
                myTextInput} aria-label="Username"
                   onChange={(event) => handleTextChange(event.
                       target.value)}
                   aria-describedby="basic-addon1"/>
        </div>
        <div className="form-check">
            <input className="form-check-input" type="checkbox"
                   onChange={() => flipBoolean()}
                   checked={myBooleanInput} id="flexCheckChecked"
                       />
        </div>
        {myBooleanInput && <p> Box value true </p>}
        {
            myArr.map((elem) => <p>{elem}</p>)
        }
        {
            myMap.map((key, value) => <p>{value}</p>)
        }
    </div>
);
```

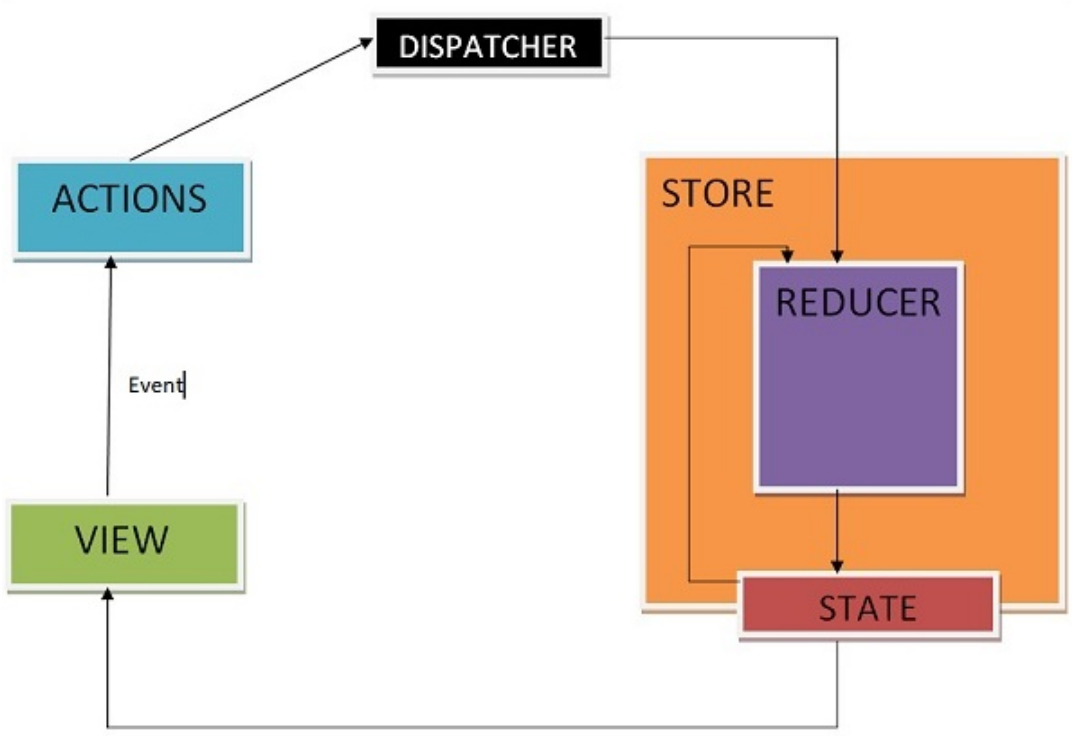
7 Sessionspeicher - Redux

Du willst Daten im Sessionspeicher haben, z.B. um zwischen verschiedenen Ansichten daten zu übertragen. Ferner gilt: Single Source of Truth. Du hast einen allgemeingültigen Datenbestand und das ist der Sessionspeicher. Dafür installieren wir Redux.

Mit Redux zu arbeiten ist gar nicht so straightforward. Du hast 3 Bestandteile:

1. Den State: Bestehend aus den Sessionvariablen
2. Eine Action: Actions sind strengenommen Objekte, die eine Aktion referenzieren. Sie referenzieren eine Methode und haben meist eine Payload. Richtig komisch, die Dinge sind aber nunmal wie sie sind. Actions sind bei uns Referenzen auf Reducer. Die Action ist im Grunde also ein Objekt, welcher einen Reducer referenziert. Redux Ding halt (ich hasse es)
3. Ein Reducer: Der Reducer ist das, was die Änderung der Action ausführt. Reducer sind ein Redux ding, weil du nur mit Reducern zustände Ändern sollst. Soll Zentralisieren

Das ganze Funktioniert dann ungefährt so



Du musst halt erstmal alle Sachen anlegen, ehe du dann weitergehst und den Store enablest. Wir wollen einfach mal eine Storevariable für die Text-Eingabe aus dem Letzen kapitel anlegen und einen für den Counter.

7.1 Setup

Du willst Redux installieren. Dann erstelle am Besten direkt ein Verzeichnis redux, mit 2 unterordnern actions und reducers.

7.2 Reducer anlegen

Leg einfach ne file namens actions.js an. Ok du willst jetzt folgendes machen: Du musst definieren wie deine Actions aussehen. Wir sagen einfach, dass das Action dann ein Objekt hat mit einem Feld Type und wir gleichen den String ab - einfahc mittels Switch-Anweisung.

```
import {combineReducers} from "redux";

const myTextReducer = (state = null, action) => {
    switch(action.type) { //action ist das mitgegebene Objekt -
        type ist einfach nur ein Feld
        case "SET_MY_TEXT" : return action.payload;
        default: return state;
    }
}

const myCounterReducer = (state = null, action) => {
    switch(action.type) { //action ist das mitgegebene Objekt -
        type ist einfach nur ein Feld
        case "SET_COUNTER" : return action.payload;
        default: return state;
    }
}
```

Was passiert hier? Da wenn in action.type der Wert stimmt, dann wird der Setter gesetzt. Du willst auch die Reducer exportieren. Dafür nutzt du dann allReducers.

```
const allReducers = combineReducers({
    text: myTextReducer,
    counter: myCounterReducer
})

export default allReducers;
```

Damit sind die Reducer geschrieben.

7.3 Actions

Die Actions zu definieren ist weniger kompliziert.

```
export const setMyText = (text) => {
    const valueObj = {type: "SET_MY_TEXT" /* MUSS UEBEREINSTIMMEN
        MIT REDUCER  */, payload: text};
    return valueObj;
};

export const setCounter = (counter) => {
    const valueObj = {type: "SET_COUNTER" /* MUSS UEBEREINSTIMMEN
        MIT REDUCER  */, payload: counter};
    return valueObj;
};
```

7.4 Enablen von Redux

Das ist alles ein bisschen zyklisch. Du musst den Store anhand von Reducern laden, mittels createStore.

Jetzt gehst du in den index.js und machst das hier:

```
const store = createStore(allReducers)

root.render(
  <React.StrictMode>
      <Provider store={store}>
          <App />
      </Provider>
  </React.StrictMode>
);
```

Spätestens jetzt musst du React Dev Tools und Redux Dev Tools installieren.

7.5 Die Setter nutzen

Das machst du mittels useDispatch.

```
    const disptach = useDispatch();

    /*Methodendefinition in react auf Funktionale art und weise ist
         richtig behindert
    In die Lambda parameter gehen deine Parameter rein*/
    const increaseNum = () => {
        setMyNum(myNum+1);
        setMessage("Increased");
        disptach(setCounter(myNum));
    }

    const decreaseNum = () => {
        setMyNum(myNum-1);
        setMessage("Decreased");
        disptach(setCounter(myNum));
    }
```

Das rufst du so auf:

```
let counter = useSelector((state) => state.counter);
```

Auch jetzt noch mit dem Text

```
    const disptach = useDispatch();

    let text = useSelector((state) => state.text);
    text = (text == null) ? '' : text;

    const[myTextInput, setMyTextInput] = useState(text);
    const[myBooleanInput, setMyBooleanInput] = useState(false);

    const handleTextChange = (val) => {
        setMyTextInput(val);
        disptach(setMyText(val));
    }
```

7.6 Bei Seitenwechsel Daten behalten: redux-persist

Wenn du Seite neulädst gehen die Daten weg. Das ist scheisse. Du kannst redux-persist dafür. Was dabei passiert ist mir scheissegal.

Hier ist was du machst:

Was du an store importierst in index.js machst du nun in einer neuen Datei in configureStore

```
import { createStore } from 'redux'
import { persistStore, persistReducer } from 'redux-persist'
import storage from 'redux-persist/lib/storage' // defaults to
    localStorage for web

import allReducers from "../reducers/reducers";

const persistConfig = {
    key: 'root',
    storage,
}

const persistedReducer = persistReducer(persistConfig, allReducers)

export default () => {
    let store = createStore(persistedReducer)
    let persistor = persistStore(store)
    return { store, persistor}
}
```

Änderungen in der index.js

```
import configureStore from "./redux/persist/configureStore";
let { store, persistor } = configureStore();


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(
  <React.StrictMode>
      <Provider store={store}>
          <PersistGate loading={null} persistor={persistor}>
              <App />
          </PersistGate>
      </Provider>
  </React.StrictMode>
);
```

8 React Hooks + Lifecycles

Es gibt nen Haufen React Hooks. Die Motivation dahinter ist, dass die UI nicht statisch ist, sondern aufÄnderungen, Lifecycle Events etc. reagieren muss. Ohne sich auf ES6-Klassen zu verlassen können wir nun Hooks nutzen. Es gibt viele Hooks, die grundlegenden sind:

1. useState

2. useEffect

3. useContext

Stateful-Logic, d.h. Daten die sich in deiner React-Anwendung ändenr, waren idR an Klassen gebunden. In React hat man für den Umgang damit einzelne Komponenten geschrieben. Renderprops und so ein scheiss. Alle React Hooks haben den Praefix use.

Vorischt: Du musst sie wirklich am Top-Level einer Functional Component (also die Art Komponente die wir haben) aufrufen. Snicht innerhalb von Code oder innerhalb von methoden. Siehe dafür den Trick mit useDispatch und dispatch,

8.1 useState

Hatten wir bereits. Der Sinn von useState ist, mit sich ändernde Daten zu hantieren. Wenn sich diese Daten ändern, bitte UI-Rerendern.

```
1  const[myVar, setMyVar] = useState('foo');
```

Die besonderheit ist hier auch, dass ein useState ein Array ist und Wert an Index 1 immer ein Setter ist. Wert an Index 0 ist die Reactive Value. Der Wert ist nur mit dem Setter zu ändern.

8.2 useEffect + React Komponenten Lebenszyklus

Komponenten haben einen Lebenszyklus. Diese bestehen aus 3 Schritten

1. componentDidMount(): Wenn die Komponente an die UI gemounted worden ist

2. componentDidUpdate(): Wenn sich Reactive Data (z.B. State) updated

3. componentWillUnmount(): Wenn die Komponente aus der UI geschmissen wird

useEffect fängt nun alle diese Schritte ab.

```
1     useEffect(() => {
2         console.log("This use Effect runs when Component is Loaded
              first, Has Reactive Data Update (useState) and when
              this component is destroyed")
3     })
4
5     //when dependencies is empty array it runs on first load
6     useEffect(() => {
7         console.log("This use Effect runs when Component is Loaded
              first because dependencies, the array as second
              argument is empty Array")
8     }, [])
9
10    //listens to any dependency - use reactive data for this
```

```
    useEffect(() => {
        console.log("This use Effect runs when myNum is updated")
    }, [myNum])

    useEffect(() => {
        console.log("This use Effect runs when message is updated")
    }, [message])

    useEffect(() => {
        console.log("This use Effect runs when message or myNum is
            updated")
    }, [myNum, message])

    //With return the code runs before Component is destroyed
    useEffect(() => {
        return () => console.log("This use Effect runs when
            Component is destroyed")
    })
```

Das erste Beispiel läuft bei allen Lebenszyklus schrittne. Das zweite nimmt auf Lambda-Ebene des Aufrufs ein zweites ARgument ein Leeres Array. Da kommen alle Variablen rein, die abgehört werden sollen (da kannst du mental ne Brücke zu Event Listener schlagen, auch wenn der vergleich hinkt). Ist das leer, dann nur bei startup. Hat dein useEffect nen return, dann wird das abgefeuert wenn UI destroyed.

8.3 useContext

Wir arbeiten mit der Context-API. Hier gehts ums Teilen von Daten zwischen Komponenten. Das ist ein Thema für sich,

useRef wird genutzt um Elemente aus der Dom in Code zu integrieren. Du Kannst damit z.B. Dom-Aktionen ausführen

```
function App() {
  const inputElement = useRef();

  const focusInput = () => {
    inputElement.current.focus();
  };

  return (
    <>
      <input type="text" ref={inputElement} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}
```

Es gibt noch weitere Hooks, die müssen aber für einen Überblick reichen.

9 Configs einrichten: dev, local, staging, prod

Du willst für verschiedene Umgebungen verschiedene Werte für Variablen haben. Beispielsweise hatte ich das oft mit URLs, wo ich URLs fürs Backend für Dev/Local/Staging/Prod abgelegt habe. Das heisst also, ich habe für jede Umgebung je eine Config-Datei, wo für denselben Bezeichner unterschiedliche Werte abgelegt werden. Im code habe ich eine Referenz auf eine Variable aus der Config. Welche Config-Datei angesprochen wird ist davon abhängig, ob ich z.B. grade auf Dev arbeite oder für local/staging/prod gebaut hab.
Ich will hier ein ganz einfaches beispiel machen. Wir legen eine Variable für 4 verschiedene Umgebungen an: Development (wenn du das auf localhost via IDE am laufen hast), Lokal, Staging Prod. Die 3 letzteren sollen BUILDS sein. Development ist theoretisch auch ein Build.
Da du 3rd Party dependencies für den Umgang mit Configs in JS-Frameworks installierst, variiert der umgang je nach npm-package. Beachte, hier nutzen wir env-cmd.

9.1 Anlegen von Configs

Wir laden das npmpackage env-cmd runter. Wir legen im Root-Verzeichnis 4 Dateien an:

1. .env.development für Dev
2. .env.local für Lokales Bauen
3. .env.staging für Staging
4. .env.production für Produktion

Da legen wir jetzt eine Variable zur Demonstration der Funktionsweise an: MY_CONFIG_VAR wo wir einen Wert reinschreiben, der sich zwischen den Configs unterscheiden. REACT_APP_ muss für jede Env Variable in React dabei sein.

```
1 REACT_APP_MY_CONFIG_VAR = "DEVELOPMENT CONFIG LOADED"
```

Listing 2: Die .env.development Datei

```
1 REACT_APP_MY_CONFIG_VAR = "LOCAL CONFIG LOADED"
```

Listing 3: Die .env.local Datei

```
1 REACT_APP_MY_CONFIG_VAR = "STAGING CONFIG LOADED"
```

Listing 4: Die .env.staging Datei

```
1 REACT_APP_MY_CONFIG_VAR = "PRODUCTION CONFIG LOADED"
```

Listing 5: Die .env.production Datei

9.2 Umgebungen einrichten

Du hast die Dateien angelegt. Jetzt musst du die package json verändern. Denk dran, das ist eine Anpassung nach env-cmd.

```
  "scripts": {
    "start:development": "env-cmd -f .env.development react-scripts
        start",
    "start:local": "env-cmd -f .env.local react-scripts start",
    "start:staging": "env-cmd -f .env.staging react-scripts start",
    "start:production": "env-cmd -f .env.production react-scripts
        start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "build:local": "env-cmd -f .env.local react-scripts build",
    "build:staging": "env-cmd -f .env.staging react-scripts build",
    "build:production": "env-cmd -f .env.production react-scripts
        build"
  }
```

Du siehst hier, dass wir eine Config file ansteuern. Wenn du z.B. für production bauen willst, machst du:

```
npm run build:production
```

Statt npm run start musst du nun:

```
npm run start:development
```

eingeben. Mit den anderen Startbefehlen kannst du verschiedene Umgebungen testen.

9.3 Abrufen der Werte

Jetzt willst du die Dateien Abrufen. Du hast immer so:

```
process.env.nameDerVariable

//z.B. hier
const variableAusConfig = process.env.REACT_APP_MY_CONFIG_VAR;
```

9.4 Wo sind meine Builds

Findest du unter build. Deine js Datei findest du unter build/static/js. Mittels index.html kannst du die app nutzen. Ein Brot-Butter deployment sieht so aus, dass du einfach den Ordner ablegst und auf die index.html zugreifst.

Du siehst auch hier, dass mit dem ¡script¿-Tag die Anwendung eingebettet wird.

```
    <script defer="defer" src="/static/js/main.4195305f.js"></
        script>
```

Listing 6: Auszug aus der index.html