React-Beispielanwendung

Web-Engineering II

Prof. Dr. Sebastian von Klinski

Ziel

- Umsetzen des HTML-Prototyps als „echte“ Webanwendung mit React und REST-Backend
- Möglichst viele Aspekte sollen vom HTML-Prototyp übernommen werden
- Die React-Anwendung soll auf den REST-Server aus der zweiten Übung zugreifen
- Für die Umsetzung der Webanwendung soll verwendet werden
  - React
  - React-Redux
  - Redux-Thunk

Ausgangspunkt: HTML-Prototyp

React – First Steps

- Vom HTML-Prototyp kann viel, aber nicht alles übernommen werden
- React verwendet in render()-Methode JSX, nicht alle HTML-Konstrukte werden von JSX/ React unterstützt

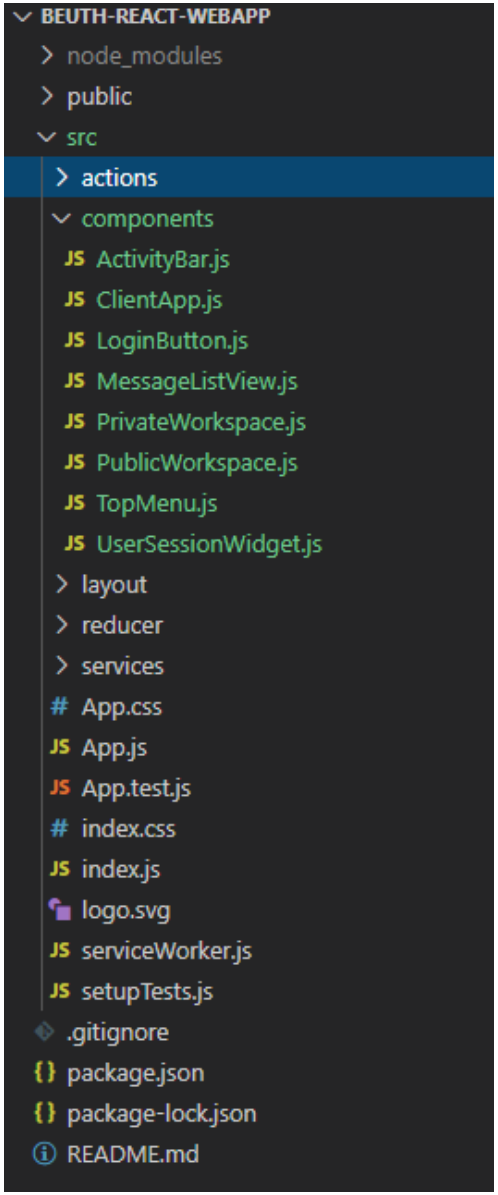
- Mögliche Schritte zum Umsetzen der Anwendung
  1. Anlegen einer Container-Komponente, die alle anderen Komponenten beinhaltet
  2. Anlegen der Reducer und Action-Komponenten
  3. Anlegen der Komponenten, die die verschiedenen Bereiche der Anwendung abbilden (z.B. Menu, Login-Komponente, Activity-Bar, etc.)
  4. Kopieren des HTML-Codes in die Render-Methode und Anpassen des HTML-Codes an die Anforderungen von JSX
  5. Umsetzen der funktionalen Aspekte mit React/ Redux

React – First Steps

- Zunächst sollte Verzeichnisstruktur festgelegt werden
- Die folgenden Artefakte gibt es
  - React-Komponenten
  - Redux-Actions und Action-Types
  - Funktionen, die die dispatch()-Methode erhalten und durch Redux-Thunk wie Actions ausgeführt werden (z.B. zum Abrufen von Posts, Thread, etc.)
  - Reducer zum Ändern des States
  - Layout-Dateien (z.B. CSS, Bilder, Icons, etc.)

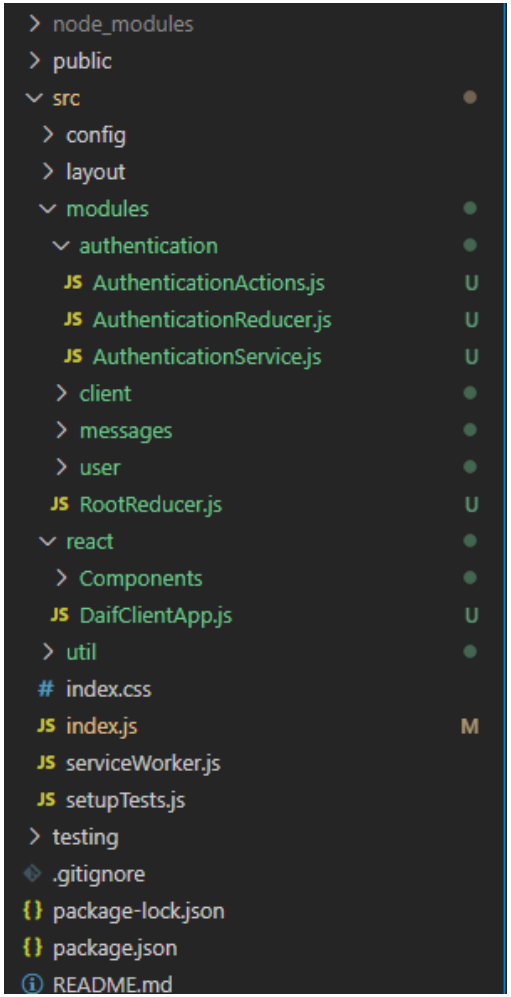
React – First Steps

- Für Übungsaufgabe kann Verzeichnisstruktur auf Basis der Art der Datei angewandt werden (components, actions, reducer, etc.)
- Für kleine bis mittlere Projekte ist das ok
- Für große Projekte ist Unterteilung nach funktionalen Modulen besser
- Verzeichnisse in „src“
  - „actions“ für Actions und Action-Types
  - „components“ für React-Komponenten
  - „layout“ für CSS-Dateien, Bilder, Icons, etc
  - „reducer“ für die Redux-Reducer
  - „services“ für die Redux-Thunk-Funktionen, die thematisch in Services zusammengefasst werden können
- Hinweis: Damit React-Komponenten auf css-Dateien zugreifen können, müssen sie unter „src“ liegen



React – First Steps

- Bei größeren Projekten ist eine Strukturierung nach funktionalen Modulen besser
- Auf der obersten Ebene
  - config: Konfigurationsangaben für die Anwendung
  - layout: CSS-Bibliotheken, Bilder, etc.
  - modules: funktionale Module mit Actions, Reducer, Services
  - react: React-Komponenten
  - util: übergreifende Hilfsfunktionen und –module
- Anwendungslogik und React-Client sollten getrennt werden
  - modules beinhaltet vor allem Anwendungslogik und Anbindung von Backend
  - React-Komponenten greifen häufig auf mehrere Module zu



1. Anlegen der Container-Komponente

Anlegen der Container-Komponente

- Container-Komponente enthält lediglich die wesentlichen Sub-Komponenten

```
import React, { Component } from "react";
import TopMenu from './Components/TopMenu'
import WorkspacePublic from './Components/WorkspacePublic'

class App extends Component {
  render() {
    return (
      <div>
        <TopMenu />
        <WorkspacePublic />
      </div>
    )
  }
}

export default App
```

Am Anfang kann zunächst nur die Startseite umgesetzt werden. Die bedingte Formatierung für den eingeloggten Zustand kann nachträglich ergänzt werden

Einbinden der Container-Komponente

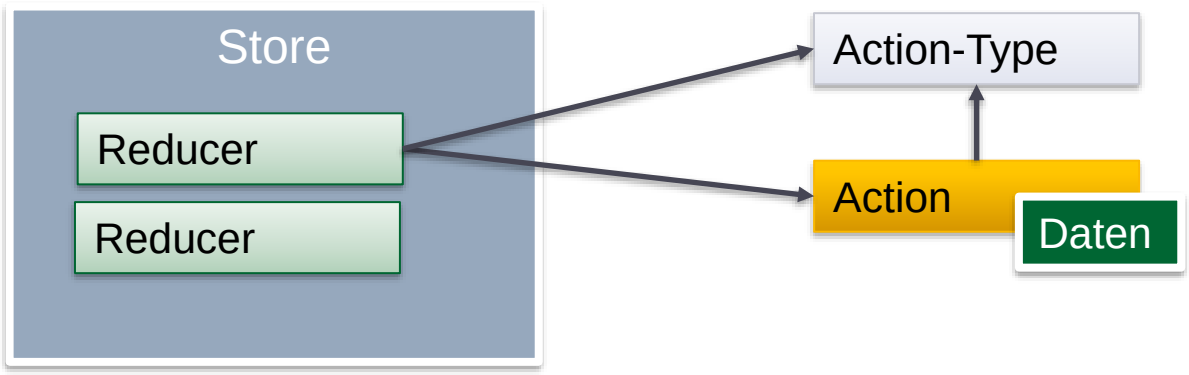
- Container-Komponente muss noch mit Redux und Thunk verbunden werden

```
…
import { applyMiddleware, createStore } from 'redux'
import thunk from 'redux-thunk';
import { Provider } from 'react-redux'
import App from './daifClient/DaifClientApp'
import rootReducer from './daifClient/Reducer/RootReducer'

const middlewares = [thunk];
const initialState = {}
const store = createStore(rootReducer,  initialState,
applyMiddleware(...middlewares));

ReactDOM.render(
  <Provider store={store}><App /></Provider>,
  document.getElementById('root')
);
```

2. Anlegen der Reducer- und Actions

Zusammenspiel (Wiederholung)

- Der Store wird mit den Reducern angelegt
- Die Reducer entscheiden anhand des Action-Types, ob sie die Action bearbeiten und welche Änderungen sie vornehmen
- Eine Action wird mit einem Action-Type angelegt
- Eine Action hat bei Bedarf noch Daten (payload), die zum Bearbeiten der Aktion notwendig sind



Root-Reducer

- In der Regel ist thematisch Aufteilung der Reducer sinnvoll
- Root-Reducer lädt die verschiedenen Reducer und kombiniert sie
- Im Beispiel: ein Reducer für die Authentifizierung und einer für das Verwalten von Nachrichten (Messages)
- **Hinweis**: Jeder Reducer erhält seinen eigenen Teil des State (z.B. state.authenticationReducer und state.messageReducer)

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

import authenticationReducer  from './AuthenticationReducer';
import messageReducer from './MessageReducer';

const rootReducer = combineReducers({
    authenticationReducer,
    messageReducer
});

export default rootReducer;
```

Authentication Reducer

- Als erstes bietet sich zur Umsetzung ein Reducer für Authentifizierung an
- Als Imports benötigt der Reducer die Action-Types

```
import * as authenticationActions from '../Actions/AuthenticationActions';
const initialState = {
    showLoginDialog: false,
    loginPeding: false,
    user: null
};

function authenticationReducer(state = initialState, action) {
    switch (action.type) {
        case authenticationActions.SHOW_LOGIN_DIALOG:
            return {
                ...state,
                showLoginDialog: true
            }
        …
```

Actions und Action-Types

- Action-Types und Action-Factory-Methods können in einer Datei definiert werden

```
export const SHOW_LOGIN_DIALOG = 'SHOW_LOGIN_DIALOG';
export const HIDE_LOGIN_DIALOG = 'HIDE_LOGIN_DIALOG‘;
…
export function getShowLoginDialogAction() {
    return {
        type: SHOW_LOGIN_DIALOG
    }
}

export function getHideLoginDialogAction() {
    return {
        type: HIDE_LOGIN_DIALOG
    }
}
…
```

3. Anlegen der Sub-Komponente

Anlegen der Sub-Komponenten

- Im ersten Schritt können einfache Klassen- oder Funktionskomponenten für die einzelnen Bereiche angelegt werden
- Beispielsweise
  - Menu-Komponente
  - Login-Komponente
  - Öffentliche Seite
  - Etc.
- In der Render-Methode kann zunächst einfach nur ein Text zurückgegeben werden
- In der Container-Komponente können dann alle Sub-Komponenten eingebunden werden
- Wenn das klappt, kann schrittweise der HTML-Code übernommen werden

Sub-Komponenten anlegen

- Implementierungen können erst mal sehr einfach gehalten sein

```
import React, { Component } from "react";
import Button from 'react-bootstrap/Button'

class LoginDialog extends Component {
    render() {
        return (
            <div>
                <Button variant="light" >Login</Button>
            </div>
        )
    }
}

export default LoginDialog;
```

5. Kopieren und Anpassen des HTML-Codes

Übernehmen des HTML-Codes

- Der HTML-Code aus dem Prototypen kann in die render()-Methode kopiert werden
- Anschließend muss der HTML-Code an die Anforderungen von React/ JSX angepasst werden
- JSX unterstützt nicht alle Aspekte von HTML in der gleichen Form
  - Code muss XML-konform sein
  - Style-Attribut muss Objekt sein
  - Class-Attribute müssen umbenannt werden
  - An HTML-Events müssen Callback()-Objekte übergeben werden
  - CSS-Attribute dürfen kein Bindestrich haben

HTML-Code muss XML-konform sein

- Es müssen zu allen geöffneten Tags auch die schließenden Tag geben (auch bei
, <img /> etc.)
- Fehlende schließende Tags müssen ergänzt werden
- Bei <img> muss das Alternative-Attribute gesetzt sein, ansonsten wird eine Warnung ausgegeben

```
// Fehler
 <br>
 <img src="../daifFiles/icons/person.svg" >

// Richtig
 <p>Hallo</p>
 <br />
<img src="../daifFiles/icons/person.svg" alt=“Bild von Person" />
```

class-Attribute umbenennen

- JSX wird nach JavaScript transkompiliert
- In JavaScript ist „class“ ein Schlüsselbegriff, der nicht anderweitig verwendet werden darf
- In React wird anstelle von „class“ nun „className“ verwendet
- → class-Attribute müssen in „className“ umbenannt werden

```
// Fehler
<div class=„InfoText">
…
</div>

// Richtig
<div className=„InfoText">
…
</div>
```

Style-Attribut

- Das Style-Attribute erwartet ein Objekt
- Bindestriche sind in den Attributnamen nicht zulässig

→ Attributnamen werden zusammengeschrieben mit Großbuchstaben

→ Werte müssen in JavaScript-Objekten angegeben werden

```
// Fehler
<img src="../daifFiles/icons/shield-lock.svg" alt="" style="padding-right: 10px;" />
<div class="row mt_50" style="margin-top: 30px; margin-bottom: 30px;">

// Richtig
…
const paddingRight = { paddingRight: '10px' };
…
 <img src="../daifFiles/icons/shield-lock.svg" alt="" style={paddingRight} />
 <img src="../daifFiles/icons/shield-lock.svg" alt="" style={{ paddingRight: '10px' }} />
 <div className="row mt_50" style={{marginTop: '30px', marginBottom: '30px'}}>
```

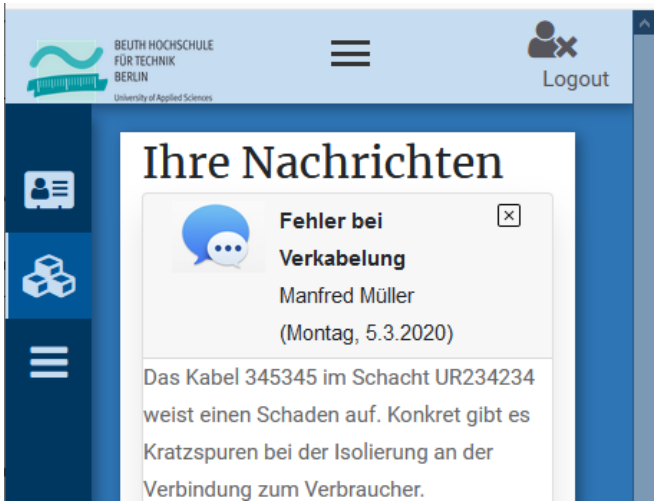
Funktionen

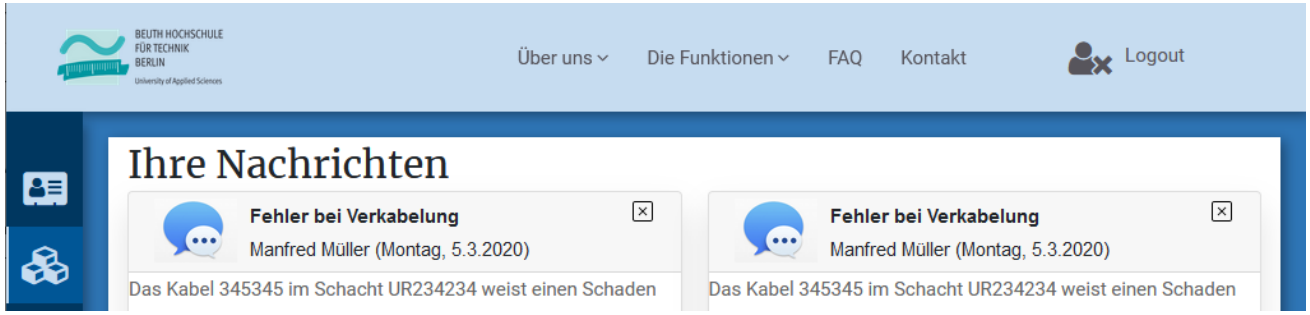
- Regulär deklarierte Funktionen müssen im Konstruktor an „this“ gebunden werden
- Beim Anbinden an Event müssen Callback-Funktionen übergeben werden (in {} ohne ())
- Die HTML-Funktionsnamen sind leicht anders: onclick() wird zu onClick()

```
…
   this.endLogin = this.endLogin.bind(this);
…
// Fehler
<button type="button" className="close" data-dismiss="modal" aria-label="Close"
                onclick=„endLogin()“>Cancel</button>
// Richtig
<button type="button" className="close" data-dismiss="modal" aria-label="Close"
                onClick={endLogin}>Cancel</button>
```

Übernehmen des HTML-Codes

- CSS-Bibliotheken wie Bootstrap verwenden für responsive Umsetzung von Komponenten JavaScript
- Beispiel: Nav für das Top-Menu
- In Abhängigkeit von Screen Size wird Rendering von Menu mit JavaScript angepasst
- Da React kein Ändern des DOM durch andere Komponenten zulässt, funktionieren solche Funktionen nicht mehr
- JavaScript-Bibliotheken z.B. von Bootstrap als HTML-Code interpretiert und nicht ausgeführt





Übernehmen des HTML-Codes

- Optionen für Umsetzen von responsive Komponenten
  - Verwendung von React-Varianten der CSS-Bibliotheken
  - Eigene Anpassung von Komponenten
- React-Varianten der CSS-Bibliotheken
  - Für verschiedene CSS-Bibliotheken gibt es React-Varianten
  - Bootstrap: react-bootstrap
  - Material: material-ui/core
- Eigene Anpassung
  - Einfache Umsetzung durch Module, die Größenbestimmung erleichtern
  - Beispiel: react-responsive
  - Einfach Abfrage der Screen Size: const isTabletOrMobile = useMediaQuery({ query: '(max-width: 1224px)' })

React-Bootstrap

- Installation: npm install react-bootstrap bootstrap
- React-Implementierungen für alle Komponenten von Bootstrap (z.B. Button, Cards, Forms, Navs etc.)
- Erscheinungsbild fast identisch zu normalen HTML-Komponenten
- Zentrale CSS-Datei sollte in index.js importiert werden

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import '../node_modules/bootstrap/dist/css/bootstrap.min.css';
…
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

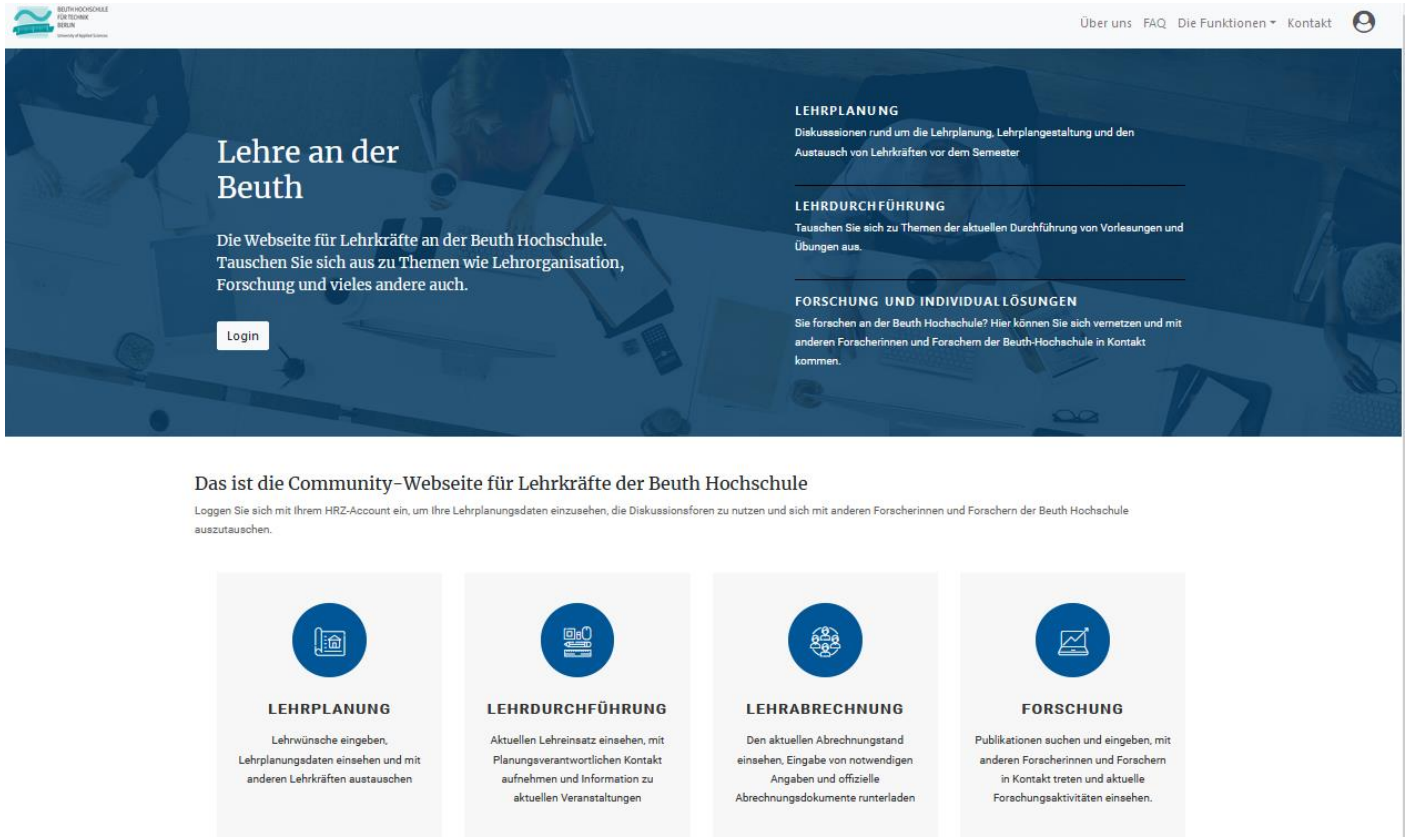
React-Bootstrap

- Für Bootstrap-Komponenten gibt es in der Regel entsprechende React-Komponenten
- Zum Nutzen müssen diese importiert werden

```
…
import Navbar from 'react-bootstrap/Navbar'
import Nav from 'react-bootstrap/Nav'
import NavDropdown from 'react-bootstrap/NavDropdown’
…
 render() {
    return (
      <Navbar bg="light" expand="lg">
        <Navbar.Brand href="#home">
          <img src={logo} alt="Logo" width="130px" />
        </Navbar.Brand>
…
```

Ergebnis von direkten Übernehmen des HTML-Codes

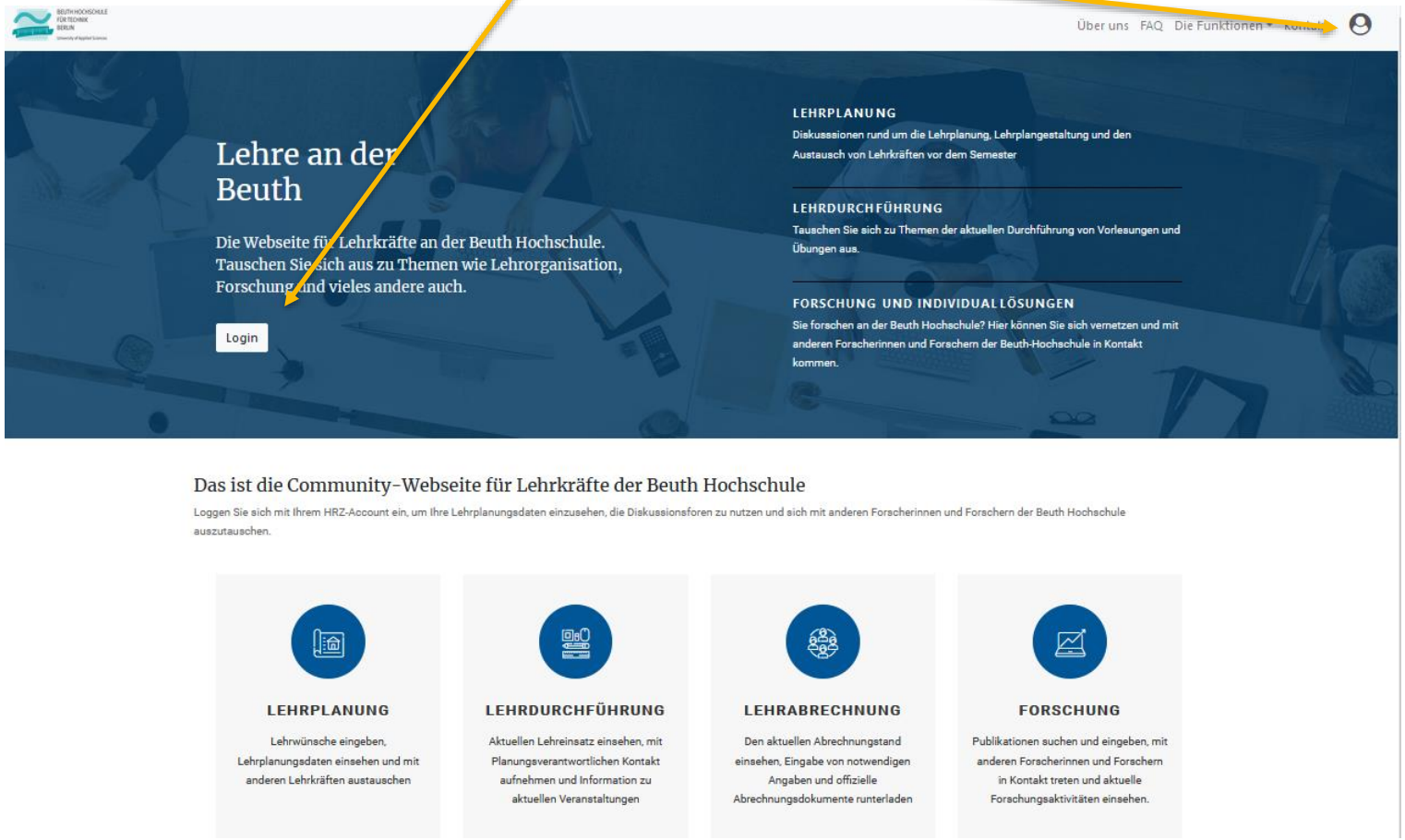
- Abgesehen von leichten Design-Unterschieden weitgehend identisch
- Einige Anpassungen bei Fonts und Farben leicht anders



5. Funktionale Aspekte umsetzen
Beispiel: Login

Ausgangspunkt: HTML-Prototyp

- Login-Dialog soll über Button und Session-Widget geöffnet werden können



Login

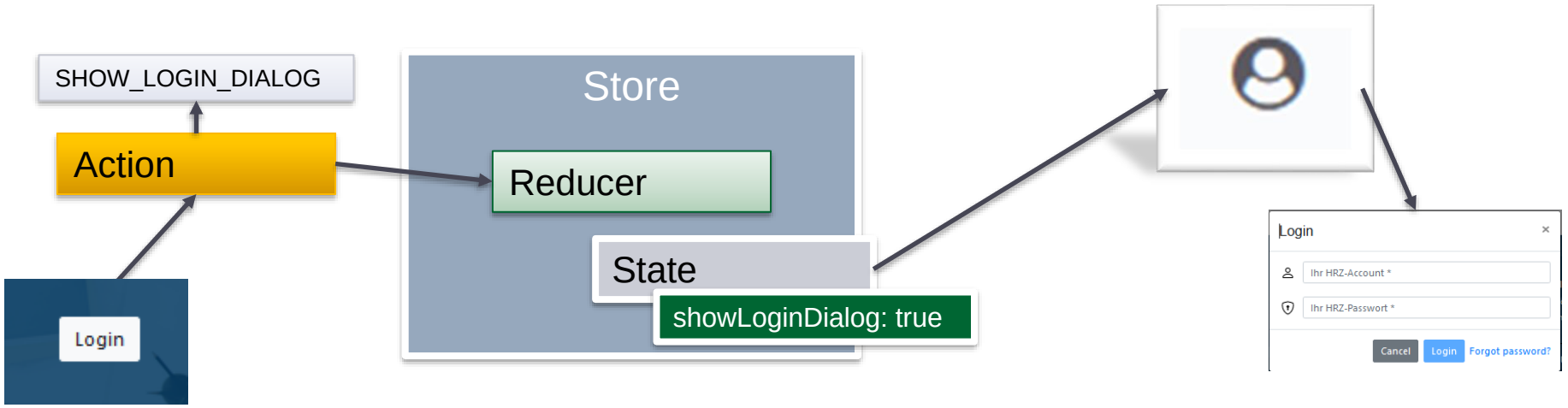
- Login soll über modalen Dialog umgesetzt werden
- Damit Login-Dialog nicht mehrfach implementiert wird, müssen…
  - Button und Session-Widget miteinander kommunizieren können
  - Bei einem von beiden muss der Dialog angehangen werden
- Kommunikation zwischen Komponenten soll über Redux umgesetzt werden
- Gewählter Ansatz: Umsetzen des Login-Dialogs in Session-Widget, da das Top-Menu immer angezeigt werden soll (egal ob eingeloggt oder nicht)

Login

- Da sich Login-Button und Session-Widget gegenseitig nicht kennen, wird Redux für die Kommunikation verwendet
- Login-Button soll Session-Widget über eine Action Bescheid sagen, dass Login-Dialog angezeigt werden soll

Login-Button-Kommunikation

- Damit Button dem Session-Widget Bescheid sagen kann, wird benötigt …
  - Action-Type „SHOW_LOGIN_DIALOG“
  - Action mit den Action-Type „SHOW_LOGIN_DIALOG“
  - Action-Factory-Method zum Anlegen der Action
  - Reducer, der den State-Wert „showLoginDialog“ auf true setzt
  - Login-Button muss dem Store Bescheid sagen können
  - Session-Widget muss den State vom Store erhalten und Dialog öffnen



Login-Button

- Zum Öffnen des Dialogs wird entsprechende Action verschickt

```
…
class LoginButton extends Component {
    constructor(props) {
        super(props);
        this.showLoginDialog = this.showLoginDialog.bind(this);
    }
    showLoginDialog() {
        const dispatch = this.props.dispatch
        dispatch(getShowLoginDialogAction())
    }
    render() {
        return (
            <div>
                <Button variant="light" onClick={this.showLoginDialog}>Login</Button>
            </div>
 )}}
…
```

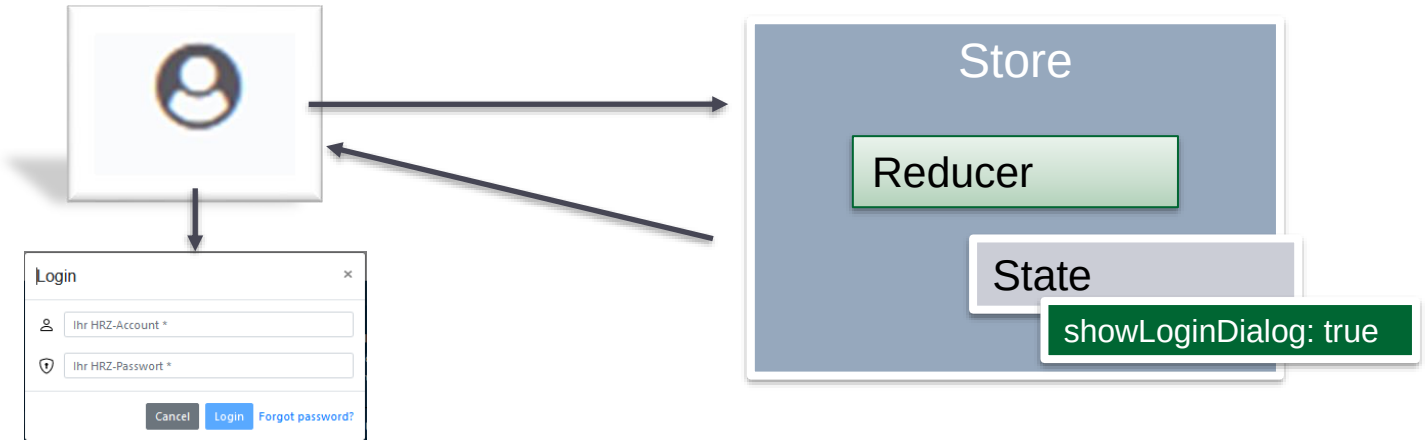
Login-Button

- Damit Komponenten die dispatch()-Methode vom Store bekommt, muss die Komponente per connect() mit dem Store verbunden werden
- Daraufhin erhält die Komponente in den Props die dispatch()-Methode
- Über die dispatch()-Methode kann die Action verschickt werden

```
…
  showLoginDialog() {
        const dispatch = this.props.dispatch
        dispatch(getShowLoginDialogAction())
    }
    render() {
        return (
            <div>
                <Button variant="light" onClick={this.showLoginDialog}>Login</Button>
            </div>
 )}}
…
export default connect()(LoginButton)
```

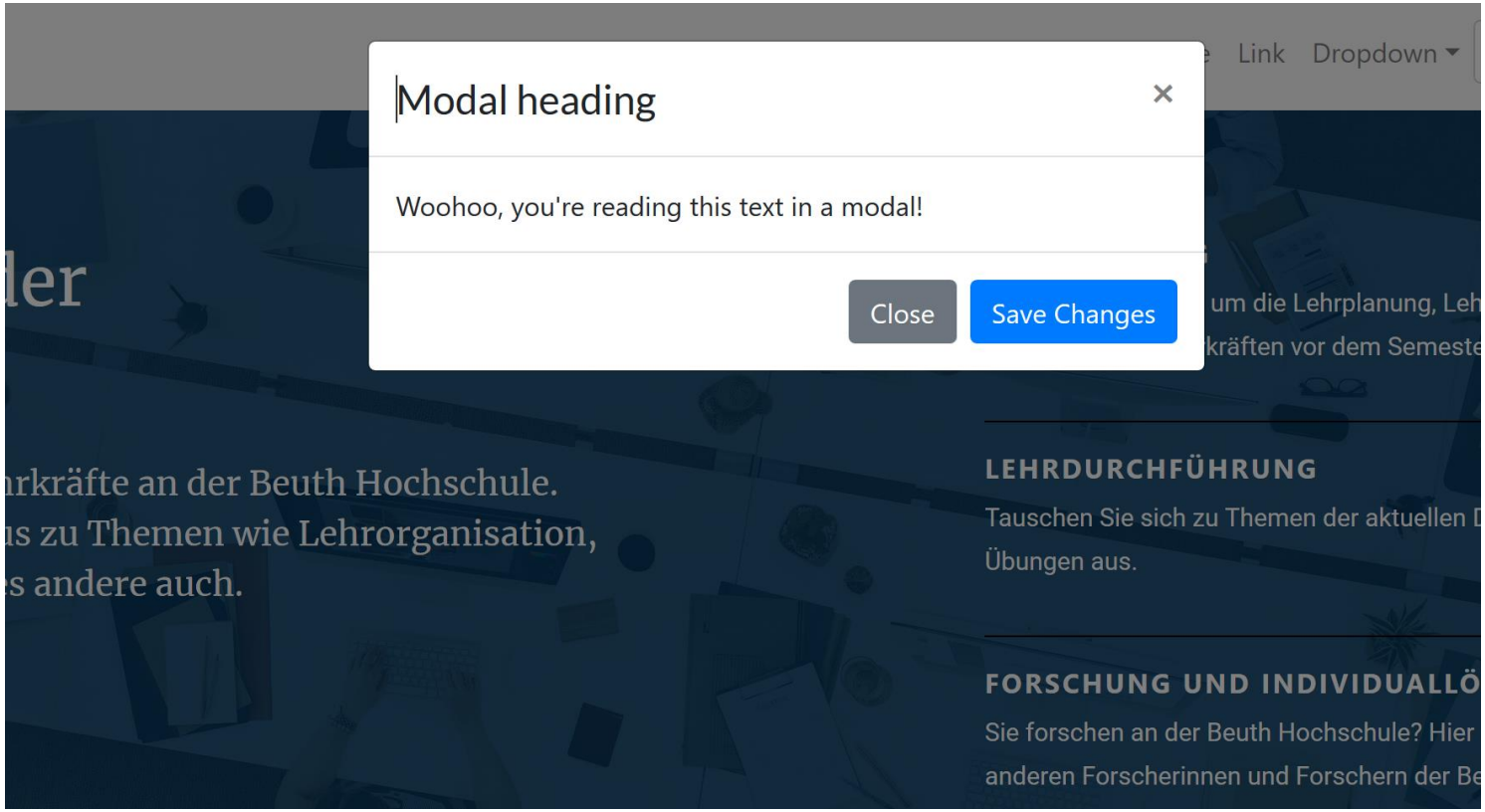
Session-Widget

- Session-Widget setzt in der render()-Methode um, ob modaler Dialog angezeigt wird oder nicht
- Wenn im State „showLoginDialog“ true ist, soll Dialog angezeigt werden
- Der State kann durch Login-Button oder Session-Widget gesetzt werden
- Damit der State konsistent bleibt, sollte auch das Session-Widget den Dialog über den State im Store öffnen



Modaler Dialog mit React/ Bootstrap

- Modaler Dialog kann beispielsweise mit Bootstrap-Komponenten umgesetzt werden



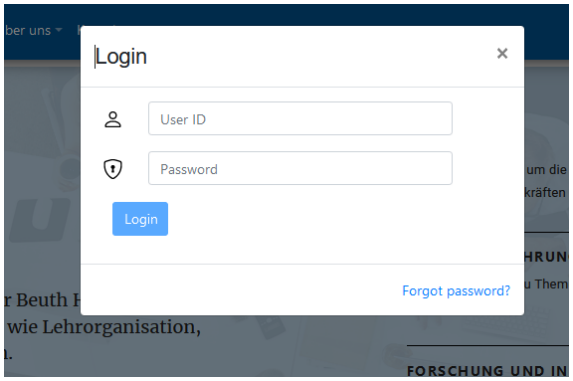
Session-Widget

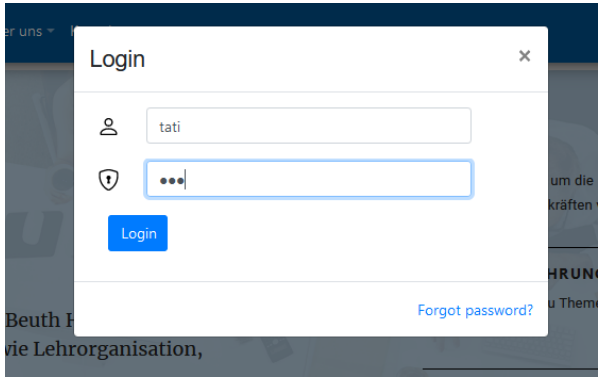
- Für modalen Dialog gibt es Standardkomponente in React-Bootstrap
- Für Eingabe von User-ID und Password muss in den modalen Dialog ein Form integriert werden

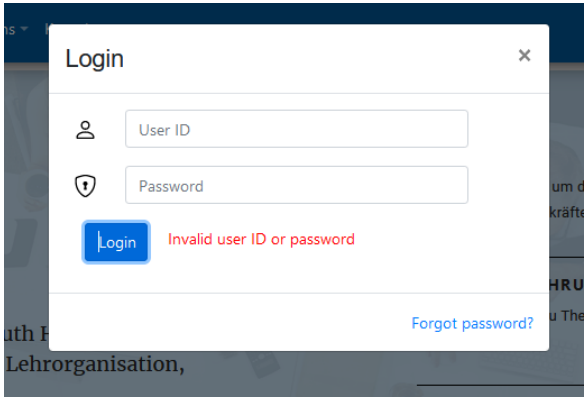
```
…
<Modal show={showDialog} onHide={this.handleClose}>
    <Modal.Header closeButton>
        <Modal.Title>Modal heading</Modal.Title>
    </Modal.Header>
    <Modal.Body>Woohoo, you're reading this text in a modal!</Modal.Body>
    <Modal.Footer>
        <Button variant="secondary" onClick={this.handleClose}>Close</Button>
        <Button variant="primary" onClick={this.handleClose}>Save Changes</Button>
    </Modal.Footer>
</Modal>
…
```

Session-Widget: Login-Dialog öffnen

- Der Login-Dialog ist ein modaler Dialog aus React-Bootstrap
- Die Login-Felder sind in einem Form
- Wenn der Login-Button gedrückt wird, wird handleSubmit() aufgerufen
- Der Login-Button wird nur aktiviert, wenn User-Name und Passwort eingegeben sind
- Wenn die Authentifizierung scheitet, wird eine entsprechende Fehlernachricht ausgegeben







Anzeigen von modalen Dialog

- Anzeigen des modalen Dialogs wird durch boolschen Wert festgelegt
- Vor dem zurückgeben von JSX-Content muss festgestellt werden, ob Dialog angezeigt werden soll

```
…
var showDialog = this.props.showLoginDialog
return (
 <>
  <Modal show={showDialog} onHide={this.handleClose}>
    <Modal.Header closeButton>
        <Modal.Title>Modal heading</Modal.Title>
    </Modal.Header>
    <Modal.Body>Woohoo, you're reading this text in a modal!</Modal.Body>
    <Modal.Footer>
        <Button variant="secondary" onClick={this.handleClose}>Close</Button>
        <Button variant="primary" onClick={this.handleClose}>Save Changes</Button>
    </Modal.Footer>
</Modal>
…
```

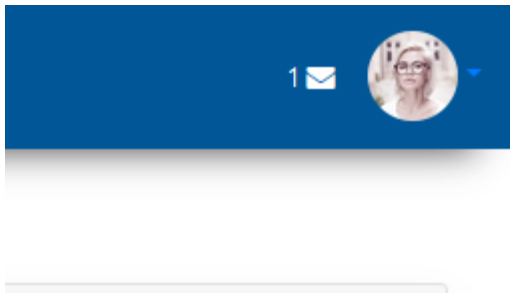
Login-Dialog (Ausschnitte vom Code)

```
<Modal show={showDialog} onHide={this.handleClose}>
   <Modal.Body>
              <Form ref={form => this.messageForm = form}>
               <Form.Group as={Row} controlId="formHorizontalEmail">
                  <img src={personIcon} alt="" title="User ID" />
                   <Col sm={9}>
                  <Form.Control type="text" placeholder="User ID" name="username" onChange={this.handleChange} />
                    </Col>
               </Form.Group>
               <Form.Group as={Row} controlId="formHorizontalEmail">
                  <img src={shieldLock} alt="“ title="Password" />
                  <Col sm={9}>
                     <Form.Control type="password" placeholder="Password" name="password"
                                 onChange={this.handleChange} /></Col>
                </Form.Group>
                <Form.Group controlId="Message">
                  <div className="d-flex align-items-center">
                    {loginButton}
                    {isError && <Form.Label style={{ color: "red"}}>Invalid user ID or password</Form.Label>}
                    {pending && <Spinner animation="border" variant="primary" />}
                   </div>
                 </Form.Group>
                </Form>
    </Modal.Body>
</Modal>
```

Session-Widget

- Session-Widget soll je nach Login-Status unterschiedliche Funktionen haben
- Es gibt keinen eingeloggten Nutzer
  - Drücken auf das Session-Widget öffnet die Login-Dialog
- Es ist ein User eingeloggt
  - Es wird das Profilbild der eingeloggten Person dargestellt
  - Links vom Profilbild werden die neuen Nachrichten angezeigt
  - Beim Drücken auf das Session-Widget öffnet sich ein Drop-Down-Menu
  - Einer der Menüeinträge ist das Ausloggen

Nicht eingeloggt



eingeloggt

Session-Widget: Bedingte Ausgabe

Wenn es „user“ gibt, ist das Widget anders als ohne „user“

```
…
let widgetButton;
    if (user) {
      const navIcon = <img src={userProfileImagePath} width="50" className="rounded-circle z-depth-0“ alt="avatar"></img>
        const userName = user.userName
        widgetButton =
          <>
            <Nav.Link href="#home" id="my-top-menu-drop-down" className="color_ff">1
                        <i className="color_ff fa fa-envelope mr-3 fa-fw"></i></Nav.Link>
            <NavDropdown alignRight title={navIcon} id="my-top-menu-drop-down">
              <NavDropdown.Item><i className="fa fa-user fa-fw"></i>: {userName}</NavDropdown.Item>
              <NavDropdown.Divider />
              <NavDropdown.Item href="*" onClick={this.handleLogout}>
                        <i className="fa fa-sign-out fa-fw"></i> Logout</NavDropdown.Item>
            </NavDropdown>
          </>
      }
      else {
        widgetButton = <a href="*"><i style={{ color: 'white', fontSize: '30px' }} onClick={this.handleShowLoginDialog}
                        className="fa fa-user-circle loginMenuText"></i>
          </a>
      }
…
```

Session-Widget: Login-Dialog öffnen

- Validierungen und Anpassungen in der Darstellung werden durch lokale Funktionen umgesetzt

```
…
 canLogin() {
        const { username, password } = this.state;
        if (username && password) {
            return true;
        }
        return false;
    }

// Das Anzeigen des Dialog muss über Redux gesteuert werden, damit es zu einen Problemen mit dem
// LoginButton kommt

    handleShowLoginDialog(e) {
        e.preventDefault();
        const { showLoginDialogAction } = this.props;
        showLoginDialogAction();
    }
…
```

Der Login-Prozess

Login-Prozess

- Für Login-Prozess muss React-Anwendung auf REST-Backend zugreifen
- Umsetzung des Logins im Dialog
  - Wenn Login-Button gedrückt wird, wird Spinner eingeblendet und damit angezeigt, dass Login gestartet wurde
  - Dann wird die Login-Anfrage an das Backend geschickt
  - Wenn Antwort vom Backend zurück ist, wird der Spinner wieder ausgeblendet und entweder…
    a. Der Login-Dialog geschlossen und der private Bereich angezeigt
    b. Die Felder im Login-Dialog gelöscht und die Nachricht zum gescheiterten Login angezeigt
- Für Anzeigen des Spinners wird im State das Attribute „pending“ abgelegt, solange der Login-Prozess läuft
- Wenn das Login erfolgreich war, wird im State der eingeloggte User abgelegt

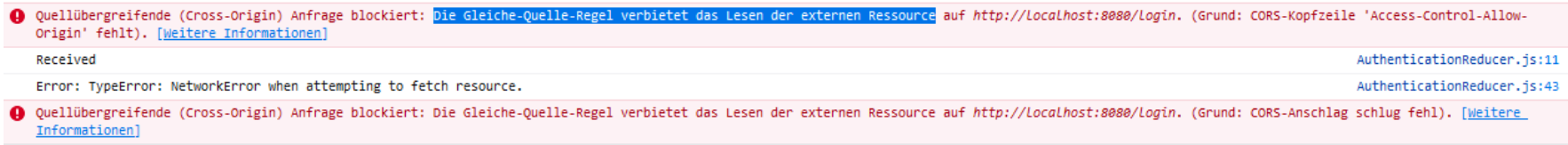
Login-Prozess: Implementierungsschritte

1. Button in Login-Form ruft lokale submit()-Funktion auf
2. Lokale submit()-Funktion holt sich die mit connect() verbundene Login-Servicefunktion für das Authentifizieren aus den Props und ruft Funktion mit User-Name und Passwort aus dem Form auf
3. Login-Servicefunktion wird ausgeführt:
   1. Es wird eine Action an den Store geschickt, dass Login-Prozess gestartet ist und dann
   2. Wird der REST-Call an das Backend geschickt
   3. Wenn die Antwort zurück ist, wird entweder
      a) Das User-Objekt und der Token in eine Action gepackt und an den Store schickt oder …
      b) eine Login-Failed-Action an den Store geschickt
4. Der Reducer ändert den State entsprechend

→ Neuzeichnen der verbundenen Komponenten wird durch Änderung des States ausgelöst

Cross-Origin Resource Sharing (CORS)

- Wenn React-Anwendung auf den REST-Server zugreift, wird zunächst eine Fehlermeldung angezeigt werden
- React-Anwendung und REST-Backend laufen üblicherweise auf unterschiedlichen Servern
- Wenn React-Anwendung auf Express-Anwendung zugreifen will, gibt es eine Cross-Origin-Exception → Zugriff wird verhindert



- **Ursache**: Browser stellt fest, dass React-Anwendung auf einen anderen Server zugreifen will als der, von dem die React-Anwendung kommt
- Browser verhindern aus Sicherheitsgründen, dass Client-Skripte auf andere Server zugreifen
- Der Zugriff auf andere Server könnte durch Schad-Code ausgeführt werden

Cross-Origin Resource Sharing (CORS)

- In der Regel gilt bei Webanwendungen die Same-Origin-Policy
  - Webanwendungen dürfen nur auf den eigenen Server zugreifen
  - Wenn auf andere Server zugegriffen werden soll, prüft Browser anhand der HTTP-Header, ob das ok ist
  - Ein Wechsel ist immer dann gegeben, wenn ein von den folgenden Aspekten sich ändert: Server, Port oder Protokoll
- Damit Zugriff möglich ist, muss Cross-Origin Ressource Sharing für den Zielserver eingestellt werden
- Das heißt: Der REST-Server muss Zugriff vom React-Server zulassen

Cross-Origin Resource Sharing (CORS)

- Der REST-Server muss Zugriff vom React-Server zulassen
- In Express muss Middleware-Funktion hinzugefügt werden
  - In Header von allen Nachrichten werden Angaben zum Access-Control gesetzt
- Im Beispiel werden durch „*“ Anfragen von beliebigen Servern zugelassen
- Es können aber auch konkrete Server oder Filter (z.B. URL, IP, Port, etc.) gesetzt werden

```
…
app.use("*",cors())

app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  res.header(“Access-Control-Expose-Headers”,“Authorization”);
  next();
});
…
```

Cross-Origin Resource Sharing (CORS)

- Zuweilen treten trotz dieser Angaben noch Probleme auf
- Gängige Ursachen:
  - Browser akzeptieren selbst erstelltes Zertifikat nicht
  - Auf dem REST-Server tritt eine Exception auf, die nicht gefangen wird
- Auch bei diesen Fehlern zeigen die Browser zuweilen eine CORS-Fehler an, was nicht richtig ist!
- Nach dem Lösen des CORS-Problems tritt häufig auch das Problem auf, dass Header-Daten nicht mit ausgeliefert werden
  - Auf dem REST-Server muss explizit angegeben werden, dass oder welche Header in der Web-Anwendung gelesen werden können
  - Wenn der Name des Headers einen Schreibfehler hat, wird er ggfls. auch entfernt

Cross-Origin Resource Sharing (CORS): Zertifikat

- Browser akzeptieren in der Regel keine selbst erstellten Zertifikate
- Firefox gibt bei diesem Fehler auch einen CORS-Fehler auf der Entwicklerkonsole aus! Das ist nicht richtig und verwirrend.
- Chrome gibt korrekterweise aus, dass ein Zertifikat-Error auftritt
- Falls Ihr Browser der Zertifikat des REST-Servers noch nicht kennt, lehnt er die Kommunikation mit dem REST-Server per HTTPS ab
- Lösung
  - Öffnen Sie mit dem Browser eine GET-Route des REST-Service per HTTPS
  - Der Browser wird dann anzeigen, dass das Zertifikat nicht gültig ist
  - Legen Sie eine Ausnahmeregel an (akzeptieren Sie das Zertifikat)
    - Der Browser legt dann das Zertifikat in seinem Secure-Store ab
    - Bei weiteren Anfragen sollte das Zertifikat akzeptiert werden
  - Diese Lösung funktioniert nicht bei Chrome unter Mac, dort muss das Zertifikat explizit zum Key-Store hinzugefügt werden

Cross-Origin Resource Sharing (CORS)

- Durch Same-Origin-Policy wird auch Weitergabe von HTTP-Header-Daten eingeschränkt
- Die Folge: der Authorization-Header wird nicht an React-Frontend geschickt
- Der React-Client braucht aber den Authorization-Header, um den Token auszulesen
- Der Header ist jedoch ohne Anpassung an Express leer!

```
…
 return fetch('http://localhost:8080/login', requestOptions)
        .then(handleResponse)
        .then(userSession => {
            return userSession;
        });
}

function handleResponse(response) {
    const authorizationHeader = response.headers.get('Authorization');
…
```

Cross-Origin Resource Sharing (CORS)

- Auch Header-Daten müssen bei Express explizit freigegeben werden, damit sie an den React-Server gesandt werden
- Mit der Freigabe in Express, kann in React der Token gelesen werden
- Er sollte im State abgelegt und bei folgenden Anfragen verwendet werden

REST-Server

```
…
const cors = require('cors');
…
app.use(cors({
  exposedHeaders: ['Authorization'],
}));
…
```

React-Anwendung

```
…
const authorizationHeader = response.headers.get('Authorization’);
token = authorizationHeader.split(" ")[1];
…
```

Fehlende Header

- Wenn trotz der Angaben der Header nicht vorhanden ist, kann die Ursache woanders liegen
- Mit dem folgenden Code im REST-Server können alle Header freigegeben werden

```
const cors = require('cors')
…
app.use('*', cors({
    exposedHeaders: ['*'],
}));
…
```

- Anschließend kann man wie folgt in der React-Anwendung alle Header ausgeben

```
response.headers.forEach(function(val, key) { console.log(key + ' -> ' + val); });
```

Cross-Origin Resource Sharing (CORS)

- Der Browser zeigt häufig in der React-Anwendung einen CORS-Fehler an, obwohl das nicht korrekt ist
- Fast alle Kommunikationsfehler beim Zugriff auf den REST-Server werden als CORS-Fehler angezeigt
  - Falscher Port
  - Falsches Protokoll
  - Port und Protokoll passen nicht
  - Falsche URL
  - Fehler, die auf dem REST-Server passieren
  - …
- Schritte zur Analyse
  - REST-Service mit der gleichen URL mit REST-Client testen
  - Konsole-Ausgaben beim REST-Server prüfen (kommt die Anfragen an?)
  - Ist die CORS-Konfiguration richtig?

Login-Prozess: 1. lokales submit() aufrufen

- Durch bedingte Ausgabe kann der Login-Button aktiviert und deaktiviert werden
- Validierung wird durch die lokale Funktion canLogin() vorgenommen

```
…
 let loginButton;
        if (this.canLogin()) {
            loginButton = <Button … type="submit" onClick={this.handleSubmit}>Login</Button>
        }
        else {
            loginButton = <Button … type="submit" disabled>Login</Button>
        }
…
 <Form.Group controlId="Message">
    {loginButton}
    {isError && <Form.Label style={{ color: "red", marginLeft: '20px' }}>Invalid user ID or
                password</Form.Label>}
    {pending && <Spinner animation="border" style={{ color: "red",}} variant="primary" />}
</Form.Group>
…
```

Login-Prozess: 2. mit dem Store verbundene Funktion

- Zum Starten des Login-Verfahrens wird die Login-Servicefunktion verwendet, die per connect() mit dem Store verbunden wird

```
…
startAuthentication() {
        const { authenticateUserAction } = this.props;
        const { username, password } = this.state;
        authenticateUserAction(username, password);
    }
…
 const mapDispatchToProps = dispatch => bindActionCreators({
    …
    authenticateUserAction: authenticationService.authenticateUser
}, dispatch)

const ConnectedUserSessionWidget = connect(mapStateToProps,
mapDispatchToProps)(UserSessionWidget);

export default ConnectedUserSessionWidget…
```

Login-Prozess: 3+4. Login-Actions ausführen

- Zunächst wird der Status gesetzt, dass der Login-Prozess gestartet ist (um z.B. den Spinner anzuzeigen)
- Dann wird der REST-Call in login(userID,password) abgeschickt

```
…
function authenticateUser(userID, password) {
    return dispatch => {
        dispatch(getAuthenticateUserPendingAction());
        login(userID, password)
            .then(
                userSession => {
                    dispatch(getAuthenticationSuccessAction(userSession));
                },
                error => {
                    dispatch(getAuthenticationErrorAction(error));
                }
            )
            .catch(error => {dispatch(getAuthenticationErrorAction(error));
   })}}
…
```

Login-Prozess: 3+4. Login-Actions ausführen

- In der Login-Funktion wird der REST-Call ausgeführt
- Die Funktion handleResponse() liest aus der Antwort die User-Daten und den Token

```
…
function login(userID, password) {
    const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ userID, password })
    };

    return fetch('http://localhost:8080/login', requestOptions)
        .then(handleResponse)
        .then(userSession => {
            return userSession;
        });
}
…
```

Login-Prozess: 5. Reducer ändert den State

- Der Reducer übernimmt im State den User und den Token aus der Antwort
- Der Login-Dialog wird geschlossen und pending wird auf „false“ gesetzt, damit der Spinner wieder ausgeblendet wird

```
…
function authenticationReducer(state = initialState, action) {

    switch (action.type) {
    …
        case authenticationActions.AUTHENTICATION_SUCCESS:
            {
                return {
                    ...state,
                    showLoginDialog: false,
                    pending: false,
                    user: action.user,
                    accessToken: action.accessToken
                }
            }
…
```

Login-Prozess: Komponenten aktualisieren

- Wenn der Reducer den State verändert hat, werden die verbundenen Komponenten informiert
- Diese können sich aus dem State die entsprechenden Informationen aus dem State holen und neu rendern (siehe mapStateToProps())
- **Hinweis**: Jeder Reducer hat seinen eigenen State → die Props müssen ggfls. zusammengestellt werden

```
…
const mapStateToProps = state => {
   return state.authenticationReducer
};
…
 render() {

       var user = this.props.user
       var showDialog = this.props.showLoginDialog
 …
 }
…
```

Nachrichtenlisten, etc.

- Die Prozesse zum Abrufen von Messages, Posts, etc. ist fast identisch
- Es gibt nur wenigen Abweichungen:
  - In Nachrichtenlisten, etc. werden die Abfragen in der Regel nicht über einen Button gestartet
  - Statt dessen sollten die Daten geholt werden, sobald die Komponente dargestellt wird
  - Hierfür bietet sich die componentDidMount()-Methode an

```
…
class MessageListView extends Component {

  componentDidMount() {
    const { fetchMessages } = this.props;
    fetchMessages(this.props.accessToken);
  }
…
```

Nachrichtenlisten, etc.

- **Wichtig**: Der Access-Token muss mit jeder Anfrage verschickt werden
- Der Access-Token kann im State abgelegt werden und als Parameter bei den Abfragen übergeben werden

```
…
class MessageListView extends Component {

  componentDidMount() {
    const { fetchMessages } = this.props;
    fetchMessages(this.props.accessToken);
  }
…
```