# Preparing the project and creating first components

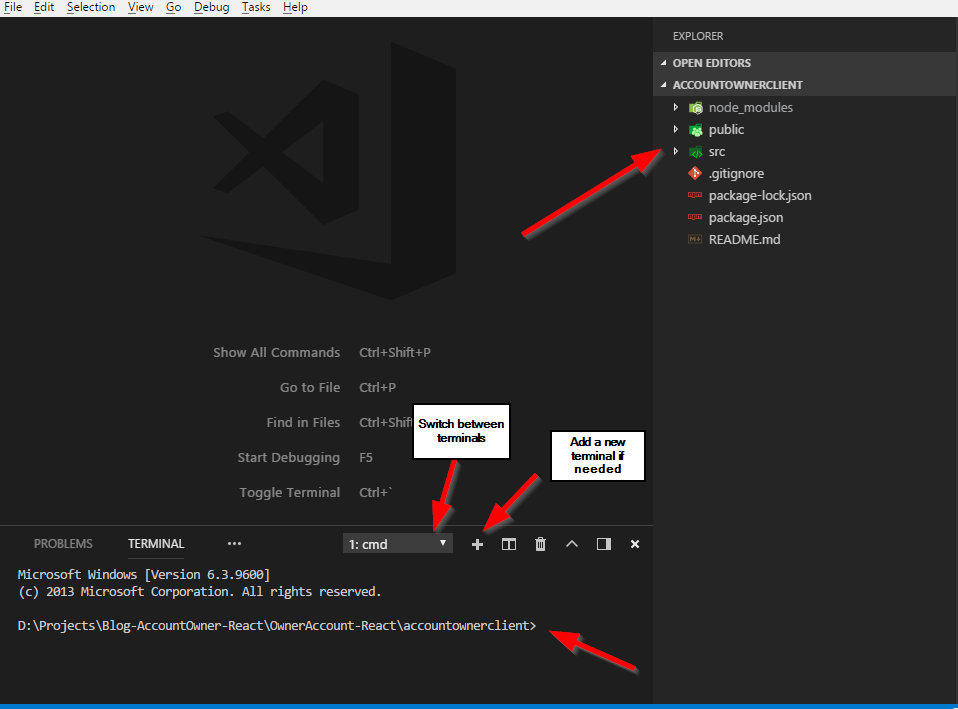
Creating the server part (.NET Core web API part) is just a half of the job we want to accomplish. From this point onwards, we are going to dive into the client side of the application to consume the web API part and show the results to the user by using React components and many other features.

## Creating a New Project

To create a new project, we need to execute Create React App command for creating a new application. Open Visual Studio Code, in a terminal window (CTRL+`), navigate to the folder you want your project in and type the command:

npx create-react-app accountownerclient

After some time a new project is going to be created:



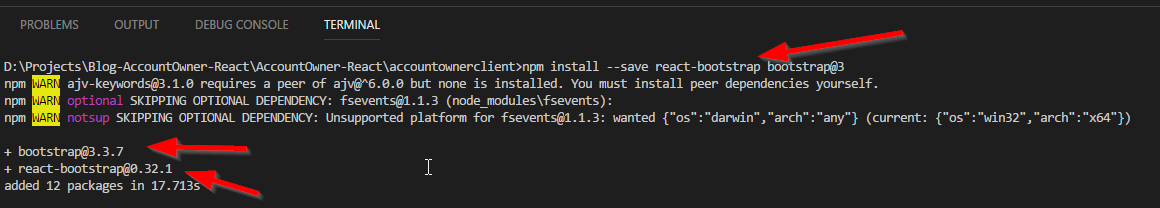
If you want to learn in more detail about the Create React App, visit this site: <https://github.com/facebook/create-react-app>

## Third-Party Libraries

We are going to use the React-Bootstrap library for styling our components so let’s install it and import it into the project.

Type this command to install the React-Bootstrap and the Bootstrap version 3 libraries:

npm install --save react-bootstrap bootstrap@3



After the installation, import the Bootstrap library inside the index.js file:

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

Now we can use bootstrap in our project. We are going to install more third-party libraries inside our project, but we are going to deal with those installations once we need additional libraries.

## React Components Overview

The React is a framework for creating the SPA’s (Single Page Application) applications. Therefore, we are going to create all of our pages on one page. That page is the index.html. If you look at that page, you are going to notice this line of code:

<div id="root"></div>

This is the place where all of our pages are going to be generated.

But how does React know to render all the pages inside that tag?

Well, if we look in the index.js file, we are going to notice this line of code:

ReactDOM.render(<App />, document.getElementById('root'));

What this means is that React will render all the components from the App component to the index.html page inside the div with id=root. This also means that App component is going to be the main component of our entire application.

So let’s talk a bit about components in React. There are two types of components:

* Stateful (class) components and
* Stateless (functional) components

The App component is a stateful component because it has access to the state. Stateful components also have the lifecycle events and access to the props with the this.props expression. If you look in the App.js file, you are going to notice the render() function which is one of the lifecycle functions. Stateful components must have at least render() function from all the lifecycle functions inside the React. They must extend from the Component class.

Stateless components are much simpler than the stateful components. They don’t have a lifecycle and should be used whenever you don’t need state inside your component. They also have access to the props, which React provides to them as a parameter.

Whether you using the stateful or the stateless components you must export that component to use it in any other components.

## Creating Our Components

Before we start with the creation, let’s modify the App.js and App.css files. Remove all the code from the App.css file and modify the App.js file:

import React, { Component } from 'react';

import logo from './logo.svg';

import './App.css';

class App extends Component {

render() {

return (

);

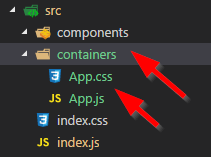
}

}

export default App;

You are going to have an error now because return() functions demand one root tag, but we are going to fix this soon.

Before we continue, let’s create the base folder structure for the containers (stateful components) and components (functional components). This action is not a must but it is a good practice to separate your class and functional components. We are going to put the App.js file inside containers folder and to modify index.js file because it imports the App.js.

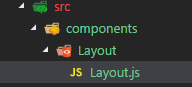


Now just modify import statement inside the index.js file and remove import for the logo.svg:

import App from './containers/App';

Excellent.

Let’s create our first component. In the components folder, create a new folder and name it Layout. Then inside this folder, create a Layout.js file and modify it:



import React from 'react';

import { Grid, Row } from 'react-bootstrap';

const layout = (props) => {

return (

<Grid>

<Row>

This is the place for the navigation component.

</Row>

<main>

{props.children}

</main>

</Grid>

)

}

export default layout;

Let’s take some time to review this code. First of all, this is a stateless component (functional) and you may notice that because we don’t have class in here but just function (arrow function to be more precise). Because this is not a class component we don’t need to extend Component, therefore we are not importing it at all. The Grid and a Row are React-Bootstrap components. Those are equivalents to the <div class=”row”> and <div class=”container”> elements if you are familiar with the Bootstrap library.

The functional component is fetching the props object through the props argument and all the properties from the props object are going to be available inside the functional component. One of those properties is the “children” property, which is going to show all the data between opening and closing Layout tag (<Layout> everything in here is children property of props object </Layout>). We are going to see this in action in next example. Finally, we are exporting this component. This type of export is called default export.

Let’s continue by modifying the App.js file:

import React, { Component } from 'react';

import './App.css';

import Layout from '../components/Layout/Layout';

class App extends Component {

render() {

return (

<Layout>

<strong>This content is going to be rendered as the props.children inside Layout component.</strong>

</Layout>

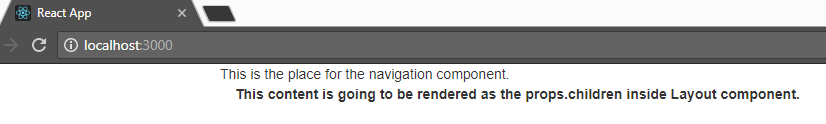
);

}

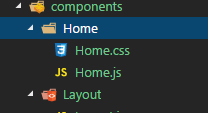
}

export default App;

Execute the npm start command in your terminal and you should be able to see your page on the localhost:3000 (not so pretty page but it is a start :D )



We are going to continue with the Home component. So, firstly create a folder structure for this component:



Modify the Home.js component:

import React from 'react';

import { Col, Row } from 'react-bootstrap';

import './Home.css';

const home = (props) => {

return (

<Row>

<Col md={12}>

<div className={'homeText'}>

"WELCOME TO ACCOUNT-OWNER APPLICATION"

</div>

</Col>

</Row>

)

}

export default home;

Then modify the Home.css file:

.homeText{

font-size: 35px;

color: red;

text-align: center;

position: relative;

top:30px;

text-shadow: 2px 2px 2px gray;

}

Finally, modify the App.js file:

import React, { Component } from 'react';

import './App.css';

import Layout from '../components/Layout/Layout';

import Home from '../components/Home/Home';

class App extends Component {

render() {

return (

<Layout>

<Home />

</Layout>

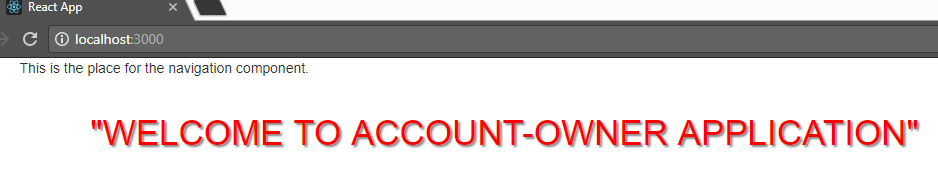
);

}

}

export default App;

When you save all of your files, the page on localhost:3000 should look like this:



# Navigation and Routing

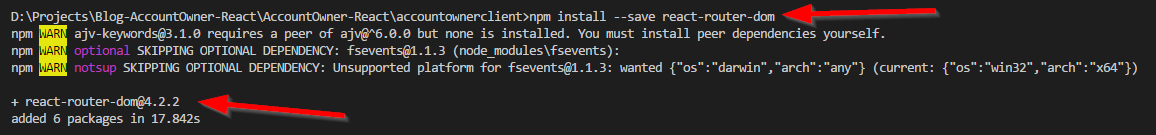
Even though we are generating all of our pages inside one page, we still need a navigation to load these different pages inside our index.html. React Router enables this navigation functionality for our project.

We are going to create three menu options: Home, Owner Actions, and Account Actions. This would be enough for us to see how navigation works in the React and how to set up lazy load functionality for asynchrony component loading inside the project.

## React Router Installation, BrowserRouter, Switch, and Route

To install React Router execute this command in the terminal window:

**npm install --save react-router-dom**



After the installation completes, let’s modify the App.js file:

import React, { Component } from 'react';

import './App.css';

import Layout from '../components/Layout/Layout';

import Home from '../components/Home/Home';

import { BrowserRouter, Switch, Route } from 'react-router-dom';

class App extends Component {

render() {

return (

<BrowserRouter>

<Switch>

<Layout>

<Route path="/" exact component={Home} />

</Layout>

</Switch>

</BrowserRouter>

);

}

}

export default App;

For the routing to work, the central component of our application must be inside BrowserRouter component. Our central component is the Layout component and we are wrapping it with the BroswerRouter. You could do this in another way inside the index.js file by wrapping the <App/> component inside the ReactDOM.render() function. Either way you choose it gives the same result.

Switch component renders the first child that matches the location. Once it finds the matching rout it will stop searching for another route.

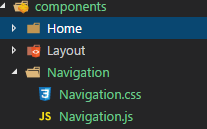
Finally, the Route component is the one that is going to redirect us from one component to another. For now, we only have the Home component and it is going to be served on the “/” path.

If you navigate to the localhost:3000 you are going to see the same result as before but now we are using routing to provide our components.

## Navigation menu

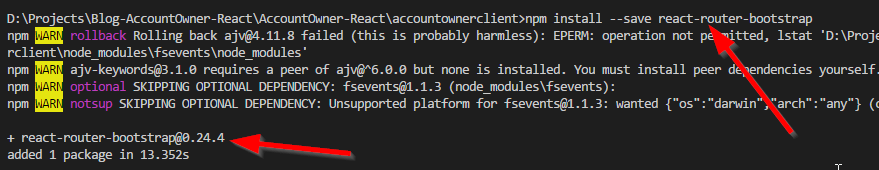
Now when we have the routing in place let’s create a navigation menu.

Inside the components folder, create a new folder and name it Navigation. Inside create two files Navigation.js and Navigation.css:



We are going to use Bootstrap navigation element for creating our own navigation. Prior to creation let’s install another library which React needs to merge Bootstrap navigation with the React-Router navigation:

npm install --save react-router-bootstrap



Now, modify the Navigation.js file:

import React from 'react';

import './Navigation.css';

import { Col, Navbar, Nav, NavItem } from 'react-bootstrap';

import { NavLink } from 'react-router-dom';

import { LinkContainer } from 'react-router-bootstrap';

const navigation = (props) => {

return (

<Col md={12} >

<Navbar inverse collapseOnSelect>

<Navbar.Header>

<Navbar.Brand>

<NavLink to={'/'} exact >Account-Owner</NavLink>

</Navbar.Brand>

<Navbar.Toggle />

</Navbar.Header>

<Navbar.Collapse>

<Nav>

<LinkContainer to={'/owner-list'} exact>

<NavItem eventKey={1}>

Owner Actions

</NavItem>

</LinkContainer>

<LinkContainer to={'/account-list'}>

<NavItem eventKey={2}>

Account Actions

</NavItem>

</LinkContainer>

</Nav>

</Navbar.Collapse>

</Navbar>

</Col>

)

}

export default navigation;

Let’s explain this code. As you might have noticed from the import statement Navbar, Nav and NavItem are the Bootstrap components and they serve us to create visual part of our navigation component. But for the routing to work we need to use NavLink, which is the react-router-doms component. With NavLink component we can navigate to exact path and also it implements styling the active link inside the navigation.

Now pay attention to the NavItem component. This is Bootstraps component to create a single navigation item. And as you already know we need to use the NavLink and not the NavItem for the routing to work. Replacing the NavItem with the NavLink won't do the trick because visual part of navigation would be scrambled. So the solution is using LinkContainer component from the react-router-bootstrap library, which is already installed. This component simulates the NavLink component completely, therefore enabling us to use this navigation without any problems.

All we have to do is to modify Navigation.css file and to include this Navigation component into the Layout component:

div a.active, ul li.active{

font-weight: bold!important;

font-style: italic!important;

color: #fff!important;

}

import React from 'react';

import { Grid, Row } from 'react-bootstrap';

import Navigation from '../Navigation/Navigation';

const layout = (props) => {

return (

<Grid>

<Row>

<Navigation/>

</Row>

<main>

{props.children}

</main>

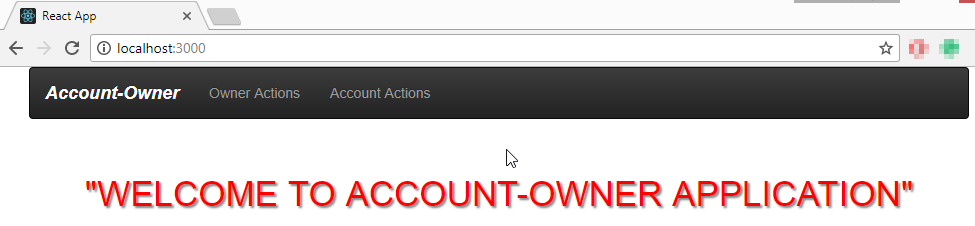
</Grid>

)

}

export default layout;

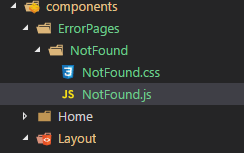
And you can check the result:



## Creating Not-Foud Component

We are going to add one more feature to complete this part of the post. Whenever a user types a nonexisting url address in the browser we are going to redirect him to the not-found (404) component.

Firstly let’s create a new folder inside the components folder and name it ErrorPages. Inside create a new folder with a name NotFound. There, create two new files NotFound.js and NotFound.css:



Modify the NotFound.js file:

import React from 'react';

import './NotFound.css'

const notFound = (props) => {

return (

<p className={'notFound'}>

"404 SORRY COULDN'T FIND IT!!!"

</p>

)

}

export default notFound;

We are just creating a functional component which returns some JSX code inside return block. JSX is syntax extension to JavaScript. It is a recommendation to use it with the React to describe what the UI should look like. Even though it could remind you of a template language, it comes with the full power of JavaScript.

Then modify the NotFound.css file:

.notFound{

font-weight: bold;

font-size: 50px;

text-align: center;

color: #f10b0b;

}

Finally, modify the App.js file:

import NotFound from '../components/ErrorPages/NotFound/NotFound';

<BrowserRouter>

<Layout>

<Switch>

<Route path="/" exact component={Home} />

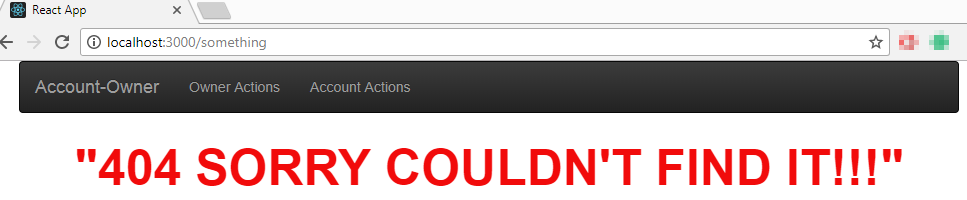
<Route pat="\*" component={NotFound} />

</Switch>

</Layout>

</BrowserRouter>

With this modification in place whenever the user types an unexisting url address, the NotFound component is going to be rendered (even if you type localhost:3000/404 :D )



# HTTP, Axios, Redux

In this post, I am going to explain how to send HTTP requests from the React. You can send those type of requests from every component but we are going to do that by centralizing the HTTP logic as some kind of a repository. For this kind and size of the project, this centralized solution fits perfectly fine. But if you have bigger projects and more complicated it would be a good solution to split repository files thus splitting the states. We are going to use Axios as a third party library to send HTTP requests and also Redux for centralizing the repository logic.

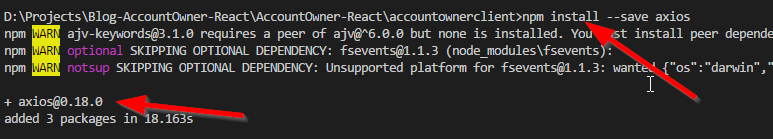
*One important notification. You don’t have to use redux for creating a central place for handling HTTP requests (this is just one of the ways), you could create an additional file and export functions (that are going to handle HTTP requests) from that file. But Redux is quite common in the React projects, therefore I am going to try to explain to you how the Redux work for this example.*

So let’s start.

## Creating the Axios instance

To install Axios execute this command:

npm install --save axios



Even though you could use native axios instance (the one you just installed) to send HTTP requests, the better way is to create your own instance. In this custom instance, you may define base url property and assign headers and all different and useful stuff. If you need more instances of axios you can create them as well. So let’s create our custom axios instance.

Inside src folder, create a new folder and name it axios. Inside that folder create a new file and name it axios.js. Modify axios.js file:

import axios from 'axios';

const instance = axios.create({

baseURL: 'http://localhost:5000',

headers: {

headerType: 'example header type'

}

});

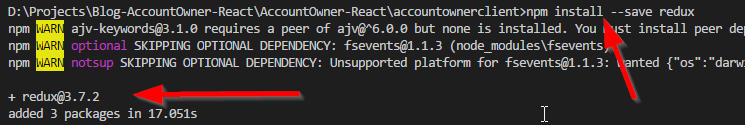
export default instance;

We are importing the axios library and then creating a new instance with the additional properties. This is very useful because you don’t have to write full endpoints anymore (axios.get(<http://locahost:5000/api/owner>)) for your HTTP requests. Now, it is just enough the shorter version (axios.get(/api/owner)) because we are using our custom axios instance with the predefined baseURL property.

## About the Redux

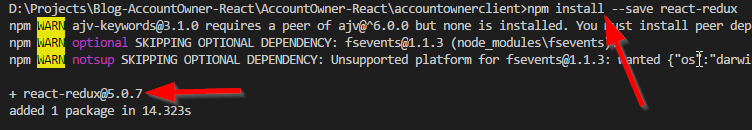
Redux is a state container for the JavaScript applications. Even though it could seem a little hard to learn at the beginning, after some practice you are going to see how easy it is and how much beneficial it is as well. Redux is not React’s library, it can fit with any other JavaScript framework, but it fits very well with the React. To install the Redux execute this command:

npm install --save redux



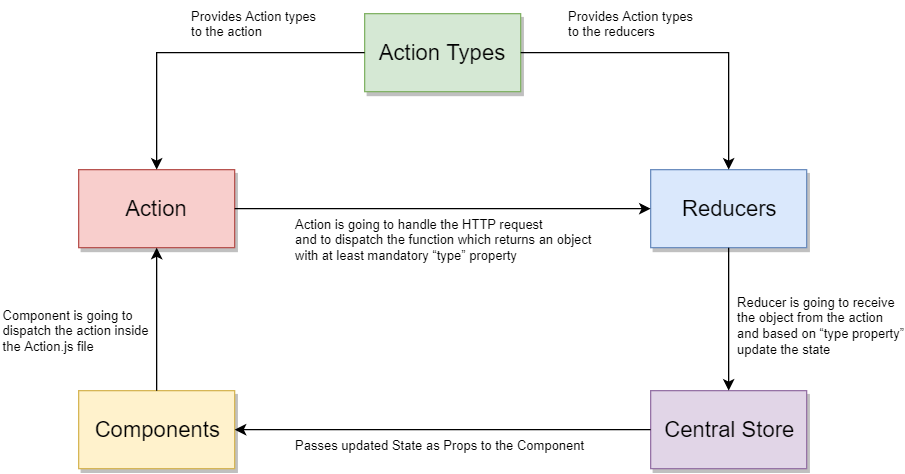
The Redux on its own is not going to be enough for the React project. We need to create a relation between the React and the Redux and to accomplish that let’s install the react-redux library:

npm install --save react-redux



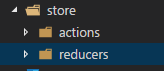
Excellent.

Now we have both libraries required for our application to work with the Redux. So, we can continue to the Redux implementation. But before we continue, take a look at this diagram which closely explains how the Redux works:



## Redux - Action Types

Let’s start by creating the following structure inside the src folder:



In the folder actions create a new file and name it actionTypes.js. Modify that file:

export const GET\_DATA\_SUCCESS = 'GET\_DATA\_SUCCESS';

export const POST\_DATA\_SUCCESS = 'POST\_DATA\_SUCCESS';

export const PUT\_DATA\_SUCCESS = 'PUT\_DATA\_SUCCESS';

export const DELETE\_DATA\_SUCCESS = 'DELETE\_DATA\_SUCCESS';

Our reducer files (which we are going to create later in this post) are going to use this action types to switch between different ways of updating the state.

## Redux - Repository actions (Action Container)

Create a new file in the actions folder and name it repositoryActions.js. We are going to handle http async requests inside this file and return an object which reducer file is going to use to update the state. Every returned object must have at least one property with a name “type”. The value for the type property would be one of the actionTypes from the actionType.js file.

Now add the import statements for the actionTypes and for our instance for axios:

import \* as actionTypes from './actionTypes';

import axios from '../../axios/aixos';

Then add this functions to handle GET request that returns an array from the server:

const getDataSuccess = (data) => {

return {

type: actionTypes.GET\_DATA\_SUCCESS,

data: data

}

}

export const getData = (url, props) => {

return (dispatch) => {

axios.get(url)

.then(response => {

dispatch(getDataSuccess(response.data));

})

.catch(error => {

//TODO: handle the error when implemented

})

}

}

What are we doing in here?

We are exporting the function getData which we are going to call from our component while fetching the data. Then with the axios, we are sending the GET request. If it is successful we are dispatching the getDataSuccess function which just returns the object to our reducer file. This object has the mandatory type property and also the data property fetched from the server.

Below the getData function, implement all the other functions:

const postDataSuccess = (response) => {

return {

type: actionTypes.POST\_DATA\_SUCCESS,

response: response

}

}

export const postData = (url, obj, props) => {

return (dispatch) => {

axios.post(url, obj)

.then(response => {

dispatch(postDataSuccess(response));

})

.catch(error => {

//TODO: handle the error when implemented

})

}

}

const putDataSuccess = (response) => {

return {

type: actionTypes.PUT\_DATA\_SUCCESS,

response: response

}

}

export const putData = (url, obj, props) => {

return (dispatch) => {

axios.put(url, obj)

.then(response => {

dispatch(putDataSuccess(response));

})

.catch(error => {

//TODO: handle the error when implemented

})

}

}

const deleteDataSuccess = (response) => {

return {

type: actionTypes.DELETE\_DATA\_SUCCESS,

response: response

}

}

export const deleteData = (url, props) => {

return (dispatch) => {

axios.delete(url)

.then(response => {

dispatch(deleteDataSuccess(response));

})

.catch(error => {

//TODO: handle the error when implemented

})

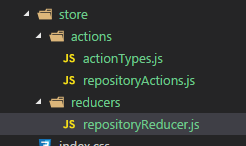
}

}

That’s it, we now have an implementation for the actions.js file and it is time to create and implement the reducers file.

## Redux – Reducer

Create a new file inside reducers folder and name it repositoryReducer.js:



In this file, we are going to check the type property which we return from the repositoryAction.js file. Then, based on that property value, we are going to update our state.

Modify repositoryAction.js file:

import \* as actionTypes from '../actions/actionTypes';

const initialState = {

data: null,

showSuccessModal: false

}

We are adding the actionTypes and creating the state with the name initialState. The data property is going to store the data from the server and the last property is here for showing and hiding the success modal when POST, PUT or DELETE action is successful.

Now let’s create reducer function below our state object:

const reducer = (state = initialState, action) => {

switch (action.type) {

case actionTypes.GET\_DATA\_SUCCESS:

return executeGetDataSuccess(state, action);

case actionTypes.POST\_DATA\_SUCCESS:

return executePostDataSuccess(state, action);

case actionTypes.PUT\_DATA\_SUCCESS:

return executePutDataSuccess(state, action);

case actionTypes.DELETE\_DATA\_SUCCESS:

return executeDeleteDataSuccess(state, action);

default:

return state;

}

}

export default reducer;

This reducer function is accepting two parameters, the state which we initialize with our initial state and the action. We are going to use this param state to update our initialState and parameter action to collect all the data sent from the repositoryAction.js file. So, whenever you dispatch any action (which returns an object with at least type property and all the other properties) from the repositoryAction.js file, this reducer function is going to trigger and to accept sent object inside the action parameter. Then reducer is going to switch through the action types and to execute the corresponding function. In the end, we must export our reducer function.

Finally, let’s add those corresponding functions right above our reducer function:

const executeGetDataSuccess = (state, action) => {

return {

...state,

data: action.data

}

}

const executePostDataSuccess = (state, action) => {

return {

...state,

showSuccessModal: true

}

}

const executePutDataSuccess = (state, action) => {

return {

...state,

showSuccessModal: true

}

}

const executeDeleteDataSuccess = (state, action) => {

return {

...state,

showSuccessModal: true

}

}

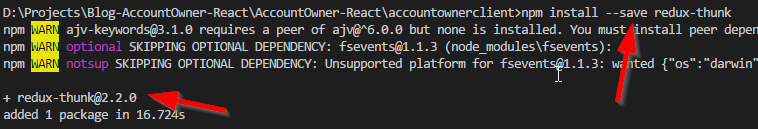
All of these functions are updating our state. Firstly we are deeply cloning our state object by using spread (…) operator and then just overriding the property we want to update in our state object. Because objects and arrays are referential types you must clone them deeply prior to any changes. That way you are updating the state immutably.

For this state to be available inside any owner component we must first register this reducer to index.js file.

## Registering reducer file

Before we register our reducer file we must install one more third-party library named thunk:

npm install --save redux-thunk



This library enables us to send async requests inside the Redux.

Now we can register our reducer.

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './containers/App';

import registerServiceWorker from './registerServiceWorker';

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

import repositoryReducer from './store/reducers/repositoryReducer';

import { Provider } from 'react-redux';

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

const store = createStore(repositoryReducer, applyMiddleware(thunk));

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

registerServiceWorker();

We are importing all the necessary files we need to register our reducer. Then we are creating the store and applying the middleware thunk and finally using the Provider to provide our reducer to the React app.

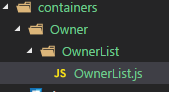
There it is, we have prepared our redux setup for our project. The best part about it is that you can use it for any other component inside your project and if you have components which require more complex or different kind of redux setup, all you have to do is to create another action and reducer file and to register it inside the index.js file. In one of the next posts, I am going to show you how to accomplish that by combining reducers inside the index.js file.

# Lazy loading and Owner List

In the previous post, we have set up the Redux environment completely and created the Axios instance for sending the HTTP requests. As a continuation, in this post, we are going to register our Redux repository inside the component and to use it to fetch the data from the server. Furthermore, we are going to load our component lazily to see the advantage of the lazy content loading.

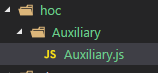
## OwnerList Component Creation, Hoc, and Routes

In the containers folder, create a new folder and name it Owner. Inside create a new folder and name it OwnerList. In there create a new file OwnerList.js.



Prior to modifying our OwnerList component, let’s create one higher order component (Hoc) which we are going to use as a helper to wrap our content inside the return block in a component. As you already know the return block requires one root tag and all the content inside that root tag. If you don’t want to use a div or a p tag that can mess up your stylings, you can create this auxiliary component. It is going to create a wrapper that returns all the children content inside it without messing up the stylings of your project.

So, inside the src folder create a new folder and name it hoc. In that folder create a new one and name it Auxiliary. In there create a new file Auxiliary.js.



Modify the Auxiliary.js file:

const auxiliary = (props) => props.children;

export default auxiliary;

All we are doing in here is just returning the children content which is going to be placed between open and close auxiliary tag.

Now, we can modify the OwnerList.js file:

import React, { Component } from 'react';

import { Table, Col, Row } from 'react-bootstrap';

import { Link } from 'react-router-dom';

import Aux from '../../../hoc/Auxiliary/Auxiliary';

class OwnerList extends Component {

render() {

let owners = [];

return (

<Aux>

<Row>

<Col mdOffset={10} md={2}>

<Link to='/createOwner' >Create Owner</Link>

</Col>

</Row>

<br />

<Row>

<Col md={12}>

<Table responsive striped>

<thead>

<tr>

<th>Name</th>

<th>Date of birth</th>

<th>Address</th>

<th>Details</th>

<th>Update</th>

<th>Delete</th>

</tr>

</thead>

<tbody>

{owners}

</tbody>

</Table>

</Col>

</Row>

</Aux>

)

}

}

export default OwnerList;

It is pretty straightforward logic in here. We are creating the empty array with named owners which is going to contain all the owners from the database. Above the table, you can notice a link to the Create Component. We are using the Link component of the react-router-dom. Then we are just creating a table which will hold all of the owners, even though it is an empty array for now.

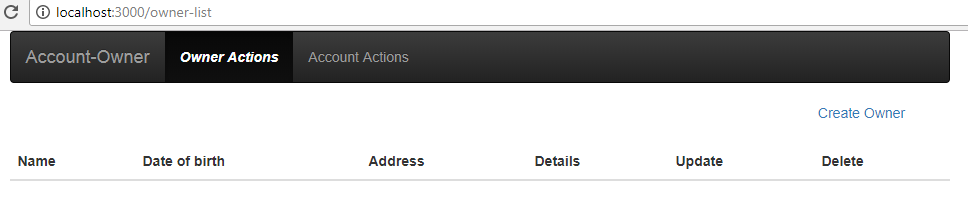
Let’s continue by modifying the App.js file to add the route to this component:

import OwnerList from './Owner/OwnerList/OwnerList';

<Route path="/" exact component={Home} />

<Route path="/owner-list" component={OwnerList} />

<Route pat="\*" component={NotFound} />

If you start your application with the npm start command, you are going to be able to navigate to the OwnerList component by clicking the Owner Actions menu.

## Implementing the Redux

We don’t have any data, for now, so let’s fix this.

Firstly we have to connect our component to our reducer which is going to set up the state and pass the data to this component as a property inside the props object.

To do that import connect from the react-redux and all the actions from the repositoryAction.js file:

import { connect } from 'react-redux';

import \* as repositoryActions from '../../../store/actions/repositoryActions';

Then below the closing bracket of our component and right above the export statement add these two functions:

}

const mapStateToProps = (state) => {

return {

data: state.repositoryReducer.data

}

}

const mapDispatchToProps = (dispatch) => {

return {

onGetData: (url, props) => dispatch(repositoryActions.getData(url, props))

}

}

export default connect(mapStateToProps, mapDispatchToProps)(OwnerList);

The mapStateToProps function is mapping the data property from the repositoryReducer’s initialState object to the data property inside the OwnerList component. To access this data property, we just need to call it like this: this.props.data.

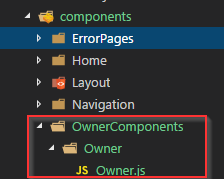
The mapDispatchToProps function is creating additional property onGetData which again we can call with the statement: this.props.onGetData. Then, it is going to dispatch the action inside the repositoryActions.js file which is going to fetch the data from the server.

*Right now if you look at the diagram from the previous post, all makes perfect sense. From this component, we are calling the action inside the repositoryActions.js file. This action is fetching the data from the server and triggers the reducer. Our reducer is going to update the state by updating the data property inside the initialState object. Finally, the Central Store is going to map that data property to this component with the mapStateToProps function.*

## Creating the Owner component

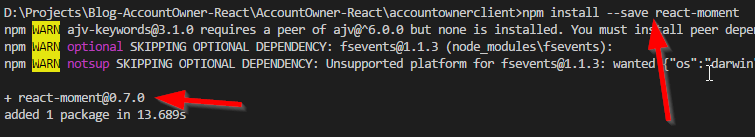
Because we are going to display multiple owners inside the OwnerList component, it is the best practice to create the Owner component and just to import it into the OwnerList component. So, let’s do exactly that.

In the components folder create the following structure:



For working with dates we need to install one more third-party library react-moment:

npm install --save react-moment



Now we can modify the Owner.js file:

import React from 'react';

import Aux from '../../../hoc/Auxiliary/Auxiliary';

import Moment from 'react-moment';

import { Button } from 'react-bootstrap';

const redirectToOwnerDetails = (id, history) => {

history.push('/ownerDetails/' + id);

}

const redirectToUpdateOwner = (id, history) => {

history.push('/updateOwner/' + id);

}

const rediterctToDeleteOwner = (id, history) => {

history.push('/deleteOwner/' + id);

}

const owner = (props) => {

return (

<Aux>

<tr>

<td>{props.owner.name}</td>

<td><Moment format="DD/MM/YYYY">{props.owner.dateOfBirth}</Moment></td>

<td>{props.owner.address}</td>

<td>

<Button onClick={() => redirectToOwnerDetails(props.owner.id, props.history)}>Details</Button>

</td>

<td>

<Button bsStyle="success" onClick={() => redirectToUpdateOwner(props.owner.id, props.history)}>Update</Button>

</td>

<td>

<Button bsStyle="danger" onClick={() => rediterctToDeleteOwner(props.owner.id, props.history)}>Delete</Button>

</td>

</tr>

</Aux>

)

}

export default owner;

In this child component, through the props, we receive the owner data and create a row with the data and couple of buttons for the navigation to different components. Every button has a function for redirection to the details, create or the update component.

Pay attention that the props object has the “history” property that allows us to navigate programmatically. Furthermore, notice that we are using date format: “DD/MM/YYYY” just to show how easy is to work with formats by using Moment library. For create and update we are going to use “MM/DD/YYYY”. All we have left to do is to import this component inside the OwnerList component and to display all the owners on the screen.

## Displaying all the owners

Import the Owner component inside the OwnerList component:

import Owner from '../../../components/OwnerComponents/Owner/Owner';

Then above the render() part, create a new function with the call to the OnGetData to fetch the data from the server:

componentDidMount = () => {

let url = '/api/owner';

this.props.onGetData(url, { ...this.props });

}

The componentDidMount is a creation lifecycle hook and it is going to trigger as soon as component mounts.

Finally, inside the render function add a code for populating our “owners” array:

let owners = [];

if (this.props.data && this.props.data.length > 0) {

owners = this.props.data.map((owner) => {

return (

<Owner key={owner.id} owner={owner} {...this.props} />

)

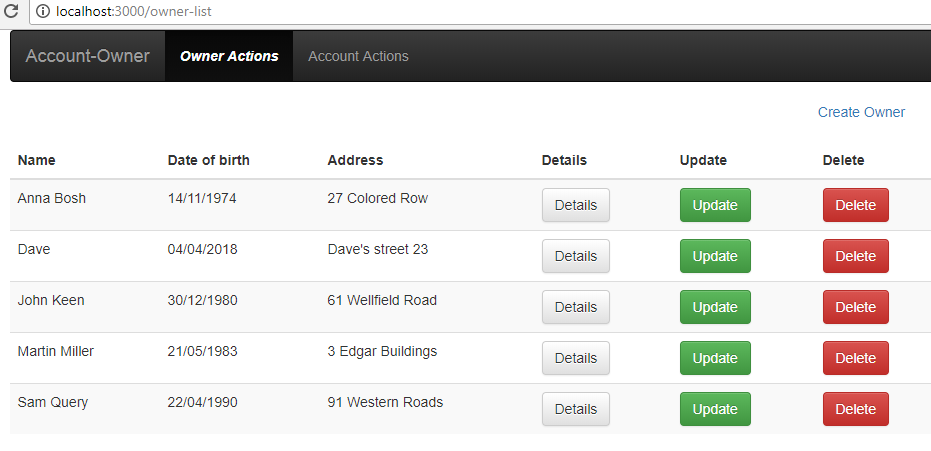
})

}

We are just checking if the data property is populated (because http requests are async requests) and if it is an array. Then for each owner of the array, we are populating the Owner component with the data. Pay attention to the key property which is mandatory when creating child components from the array of objects. Moreover, we are passing the props object immutably to have access to the “history” property of that object.

Excellent.

Now if you navigate to the Owner Actions menu you are going to see the result with all the owners:



## Lazy Content Loading

Until now, we were loading our OwnerList component eagerly and not lazily, which means that once the application starts all the resources are loaded as well. This is not the best practice because a user maybe never visits Owner Actions page, therefore resources for this page shouldn’t be loaded either.

Let’s modify our project to use lazy load for our component.

Inside the hoc folder create a new folder AsyncComponent and inside create a new file AsyncComponent.js and modify it:

import React, {Component} from 'react';

const asyncComponent = (importComponent) => {

return class extends Component{

state = {

component: null

}

componentDidMount(){

importComponent()

.then(cmp => {

this.setState({component: cmp.default});

});

}

render(){

const C = this.state.component;

return C ? <C {...this.props} /> : null;

}

}

}

export default asyncComponent;

With this component, we are going to load our component asynchronously.

To finish this action modify the App.js file to load our OwnerList component in async mode:

//import OwnerList from './Owner/OwnerList/OwnerList';

import asyncComponent from '../hoc/AsyncComponent/AsyncComponent';

const AsyncOwnerList = asyncComponent(() => {

return import('./Owner/OwnerList/OwnerList');

});

class App extends Component {

render() {

return (

<BrowserRouter>

<Layout>

<Switch>

<Route path="/" exact component={Home} />

<Route path="/owner-list" component={AsyncOwnerList} />

<Route pat="\*" component={NotFound} />

</Switch>

</Layout>

</BrowserRouter>

);

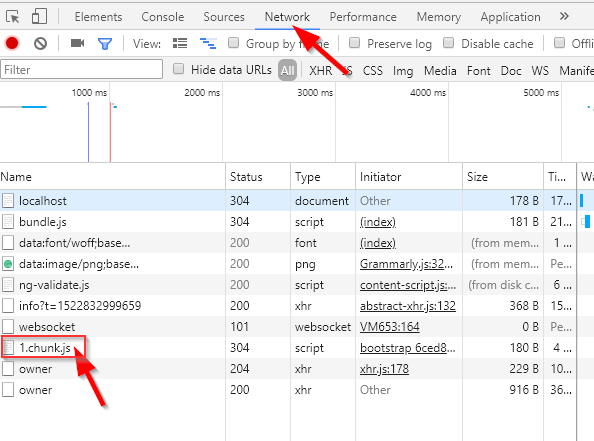
}

}

export default App;

I have commented import for the OwnerList just for you to see that we don’t need it anymore. You can delete that import statement. Now we are importing our Async component and inside we import our OwnerList component. Finally, we are not loading anymore OwnerList inside the Route component, but the AsyncOwnerList.

There it is. Now when you navigate, after restarting your app, to the Owner Actions page you are going to see additional chunk file loaded just for this page.

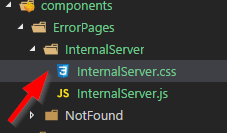


# Error Handling and the OwnerDetails Component

Handling errors while sending HTTP requests towards your server is always a must. This is exactly what we are going to do in this blog post. If the server returns the 404 or 500 error, we are going to redirect a user to a specific page. Handling other errors than the 404 or 500 is going to be executed with the modal form components. We already have the page for the 404 error so let’s continue with the creation of the 500(Internal Server) error component.

## Creating the 500(Internal Server) Error Component

Inside the ErrorPages folder create a new folder InternalServer and in there create two new files and name them InternalServer.js and InternalServer.css:



Modify the InternalServer.js file:

import React from 'react';

import './InternalServer.css';

const internalServer = (props) => {

return (

<p className={'internalServer'}>{"500 SERVER ERROR, CONTACT ADMINISTRATOR!!!!"}</p>

)

}

export default internalServer;

Then modify the InternalServer.css file:

.internalServer{

font-weight: bold;

font-size: 50px;

text-align: center;

color: #c72d2d;

}

Finally modify the App.js file:

import internalServer from '../components/ErrorPages/InternalServer/InternalServer';

<Route path="/" exact component={Home} />

<Route path="/owner-list" component={AsyncOwnerList} />

<Route path="/500" component={internalServer} />

<Route pat="\*" component={NotFound} />

## Implementing the Redux for Handling Errors

As we did with the repository part of this application, we are going to create another redux instance to handle errors in a single place in our app. We are familiar with the redux flow so this part is going to be quite easy to implement. Only additional thing is that after creating new reducer we are going to have two reducer files and therefore, we need to make different registration process inside the Index.js file.

Let’s continue by modifying the ActionTypes.js file by adding three additional action tpes:

export const HTTP\_404\_ERROR = 'HTTP\_404\_ERROR';

export const HTTP\_500\_ERROR = 'HTTP\_500\_ERROR';

export const HTTP\_OTHER\_ERROR = 'HTTP\_OTHER\_ERROR';

Inside actions folder, we have the repositoryActions.js file. Now add one more action file errorHandlerActions.js inside actions folder. Modify that file:

import \* as actionTypes from './actionTypes';

const execute404Handler = (props) => {

return {

type: actionTypes.HTTP\_404\_ERROR,

props: props

}

}

const execute500Handler = (props) => {

return {

type: actionTypes.HTTP\_500\_ERROR,

props: props

}

}

const executeOtherErrorHandler = (error) => {

return {

type: actionTypes.HTTP\_OTHER\_ERROR,

error: error

}

}

export const handleHTTPError = (error, props) => {

if (error.response.status === 404) {

return execute404Handler(props);

}

else if (error.response.status === 500) {

return execute500Handler(props);

}

else {

return executeOtherErrorHandler(error);

}

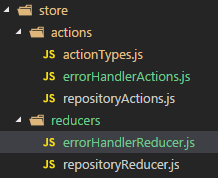
}

In here we are exporting the action handleHTTPError in which we are checking the status of the error and executing the corresponding function. You are noticing that this is the same thing we did with the repositoryActions file.

Inside the repositoryActions.js file, we need to import this errorHandlerActions file and to replace all the comments with this line of code:

dispatch(errorHandlerActions.handleHTTPError(error, props));

Continue by creating a new reducer file errorHandlerReducer.js inside a folder with the name reducers. Right now you should have this file and folder structure:



Modify errorHandlerReducer.js file:

import \* as actionTypes from '../actions/actionTypes';

const initialState = {

showErrorModal: false,

errorMessage: ''

}

const execute404 = (state, action) => {

action.props.history.push('/404');

return { ...state };

}

const execute500 = (state, action) => {

action.props.history.push('/500');

return { ...state };

}

const executeOtherError = (state, action) => {

return {

...state,

showErrorModal: true,

errorMessage: action.error.response.data

};

}

const reducer = (state = initialState, action) => {

switch (action.type) {

case actionTypes.HTTP\_404\_ERROR:

return execute404(state, action);

case actionTypes.HTTP\_500\_ERROR:

return execute500(state, action);

case actionTypes.HTTP\_OTHER\_ERROR:

return executeOtherError(state, action);

default:

return state;

}

}

export default reducer;

This logic is familiar as well. We are creating the state (initialState) and then the function reducer which accepts the state and the action parameters. The reducer function is going to update the state based on type sent from the errorHandlerActions file.

To finish the Redux setup let’s modify the Index.js file to register this reducer as well:

import repositoryReducer from './store/reducers/repositoryReducer';

import errorHandlerReducer from './store/reducers/errorHandlerReducer';

import { Provider } from 'react-redux';

import { createStore, applyMiddleware, combineReducers } from 'redux';

import thunk from 'redux-thunk';

const rootReducers = combineReducers({

repository: repositoryReducer,

errorHandler: errorHandlerReducer

})

const store = createStore(rootReducers, applyMiddleware(thunk));

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

registerServiceWorker();

That’s it. We are importing errorHandlerReducer and also combineReducers. Then with a help of the combineReducers, we create the rootReducers object which contains all of our reducers. Then we just pass this object to the store creation.

We have to do one more additional thing. In the OwnerList component, we need to change the way to set up data property from the repositoryReducer file. Modify the mapStateToProps function:

const mapStateToProps = (state) => {

return {

data: state.repository.data

}

}

We must do this because we don’t have just one reducer file anymore. Both of our reducer files are registered inside one root object and therefore, we must specify which reducer we want to use.

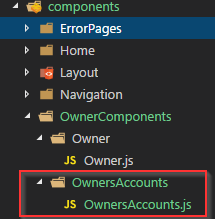
Excellent.

We can try this by modifying the code in the server’s method GetAllOwners. As the first line of a code add return NotFound() or return StatusCode(500, “Some message”), and you are going to be redirected to the right error page for sure.

## Implementation of the OwnerDetails Component

In this component, we are going to show individual owner with all of its accounts. If you are thinking: “Well, we could split this up into two components”, you are totally right. We are going to do exactly that. The parent component is going to be OwnerDetails component and the child component is going to be OwnersAccounts component. So, let’s start by creating child component first.

For the OwnersAccounts component create this structure:



Modify the OwnersAccounts.js file:

import React from 'react';

import { Row, Col, Table } from 'react-bootstrap';

import Moment from 'react-moment';

const ownersAccounts = (props) => {

let accounts = null;

if (props.accounts) {

accounts = props.accounts.map(account => {

return (

<tr key={account.id}>

<td>{account.accountType}</td>

<td><Moment format="DD/MM/YYYY">{account.dateCreated}</Moment></td>

</tr>

);

})

}

return (

<Row>

<Col md={12}>

<Table responsive striped>

<thead>

<tr>

<th>Account type</th>

<th>Date created</th>

</tr>

</thead>

<tbody>

{accounts}

</tbody>

</Table>

</Col>

</Row>

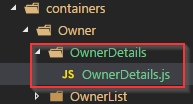
)

}

export default ownersAccounts;

We are doing great.

As a continuation let’s create a folder structure for the OwnerDetails component:



Import all the necessary files inside the OwnerDetails component:

import React, { Component } from 'react';

import { connect } from 'react-redux';

import { Well, Row, Col } from 'react-bootstrap';

import \* as repositoryActions from '../../../store/actions/repositoryActions';

import Moment from 'react-moment';

import OwnersAccounts from '../../../components/OwnerComponents/OwnersAccounts/OwnersAccounts';

import Aux from '../../../hoc/Auxiliary/Auxiliary';

Then add the component implementation:

class OwnerDetails extends Component {

render() {

const owner = this.props.data;

return (

<Aux>

<Well>

<Row>

<Col md={3}>

<strong>Owner name:</strong>

</Col>

<Col md={3}>

{owner.name}

</Col>

</Row>

<Row>

<Col md={3}>

<strong>Date of birth:</strong>

</Col>

<Col md={3}>

<Moment format="DD/MM/YYYY">{owner.dateOfBirth}</Moment>

</Col>

</Row>

{this.renderTypeOfUserConditionally(owner)}

</Well>

<OwnersAccounts accounts={owner.accounts} />

</Aux>

)

}

}

export default OwnerDetails;

As you are noticing we are using this.props.data but we haven’t implemented reducer yet. We are going to do that in a minute. Notice the renderTypeOfUserConditionally function call. In this function, we render the owner's data conditionally and return the jsx code for display. We are going to implement that function in a minute as well. Bellow the owner’s data we are displaying all the accounts related to the owner.

To implement renderTypeOfUserConditionally add this code above the render function but still inside the class:

class OwnerDetails extends Component {

renderTypeOfUserConditionally = (owner) => {

let typeOfUser = null;

if (owner.accounts && owner.accounts.length <= 2) {

typeOfUser = (

<Row>

<Col md={3}>

<strong>Type of user:</strong>

</Col>

<Col md={3}>

<span className={'text-success'}>Beginner user.</span>

</Col>

</Row>

);

}

else {

typeOfUser = (

<Row>

<Col md={3}>

<strong>Type of user:</strong>

</Col>

<Col md={3}>

<span className={'text-info'}>Advanced user.</span>

</Col>

</Row>

);

}

return typeOfUser;

}

render() {

Finally, let’s connect this component to the reducer file to fetch the owner’s data below the render function:

const mapStateToProps = (state) => {

return {

data: state.repository.data

}

}

const mapDispatchToProps = (dispatch) => {

return {

onGetData: (url, props) => dispatch(repositoryActions.getData(url, props))

}

}

export default connect(mapStateToProps, mapDispatchToProps)(OwnerDetails);

And above the renderTypeOfUserConditionally function add this lifecycle hook:

class OwnerDetails extends Component {

componentDidMount = () => {

let id = this.props.match.params.id;

let url = '/api/owner/' + id + '/account';

this.props.onGetData(url, { ...this.props })

}

renderTypeOfUserConditionaly = (owner) => {

We are fetching the id from the url by calling the match.params.id from the props object, then we are just creating our url and calling onGetData to fetch the data from the server.

The results:

