# Preparing the project and creating first components

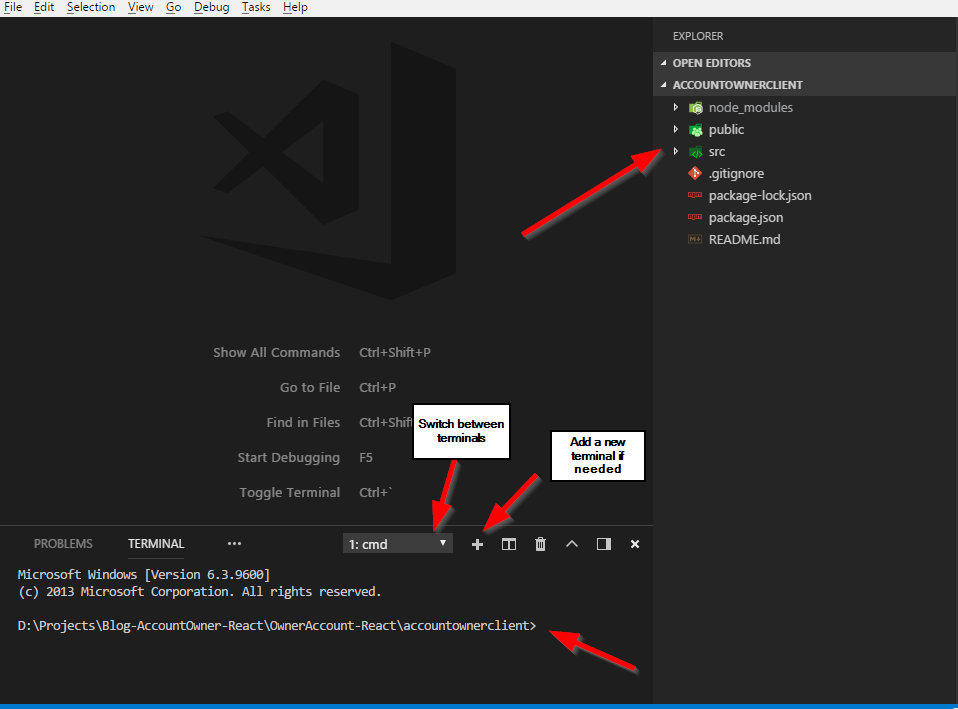
Creating the server part (.NET Core web API part) is just a half of the job we want to accomplish. From this point onwards, we are going to dive into the client side of the application to consume the web API part and show the results to the user by using React components and many other features.

## 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:



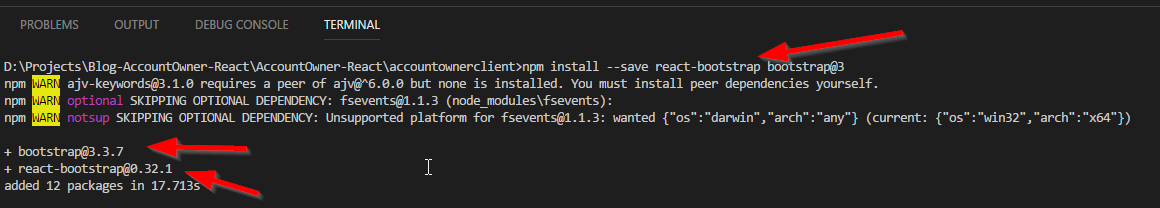
If you want to learn in more detail about the Create React App, visit this site: <https://github.com/facebook/create-react-app>

## 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 (

);

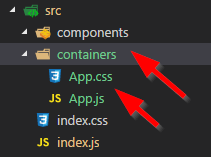
}

}

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.

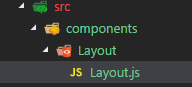


Now just modify import statement inside the index.js file and remove import for the logo.svg:

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>

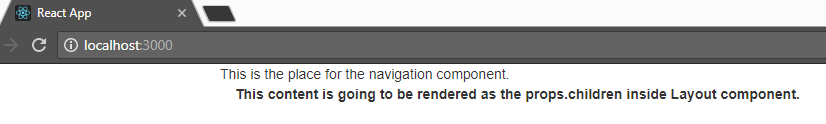
);

}

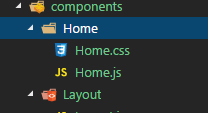
}

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>

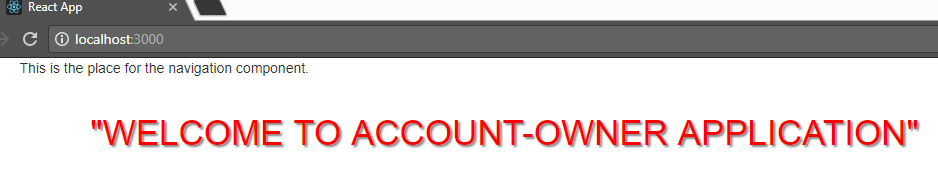
);

}

}

export default App;

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



# Navigation and Routing

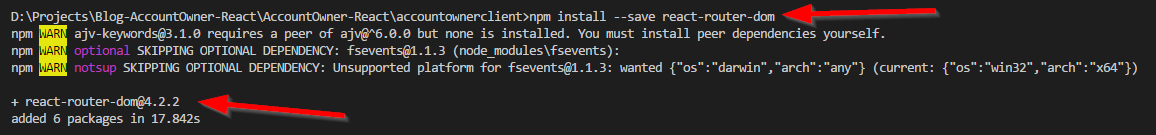
Even though we are generating all of our pages inside just one page, we still need a navigation to load these different pages inside our index.html. React Router enables this navigation functionality for our project.

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 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.

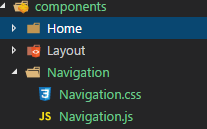
Finally, the Route component is the one that is going to redirect us from one component to another. For now, we only have the Home component and it is going to be served on the “/” path.

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

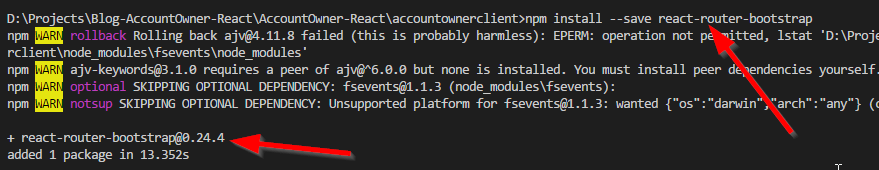
Now when we have the routing in place let’s create a navigation menu.

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 Bootstrap’s 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 need 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>

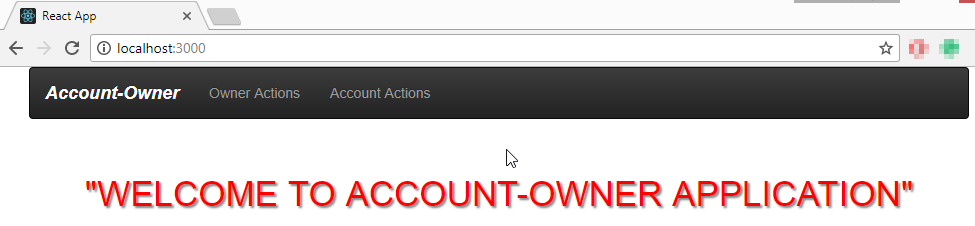
</Grid>

)

}

export default layout;

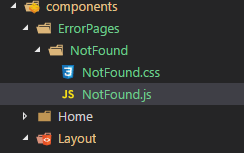
And you can check the result:



## 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 or her to the not-found (404) component.

First, 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 create 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} />

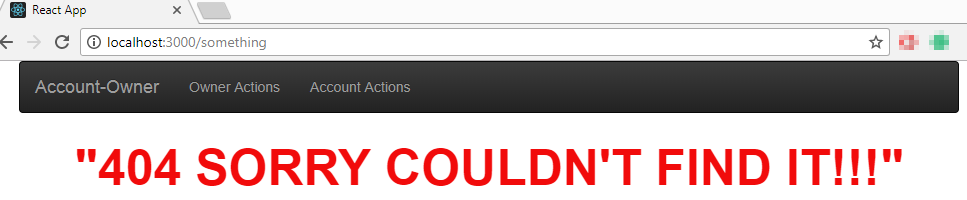
<Route pat="\*" component={NotFound} />

</Switch>

</Layout>

</BrowserRouter>

With this modification in place whenever a user types an URL address that doesn’t exist, the NotFound component is going to be rendered (even if you type localhost:3000/404 :D )



# HTTP, Axios, Redux

In this post, we are going to explain how to send HTTP requests from the React. We 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 type and size of the project, this centralized solution fits perfectly fine. But if you have bigger and more complicated projects 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.

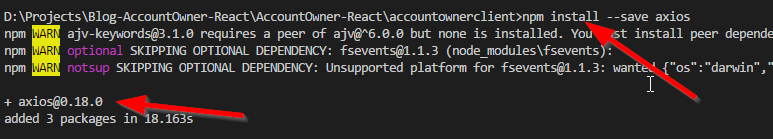
*One important notification. You don’t have to use redux for creating a central place for handling HTTP requests (this is just one way), you could create an additional file and export functions (that are going to handle HTTP requests) from that file. But Redux is quite common in the React projects, therefore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

So let’s start.

## Creating the Axios instance

To install Axios execute this command:

npm install --save axios



Even though we 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 may 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'

}

});

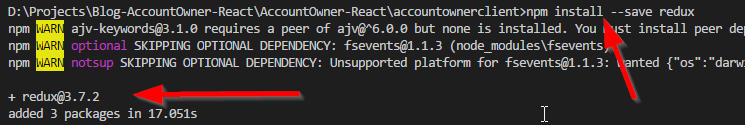
export default instance;

We import the axios library and then create 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 Redux

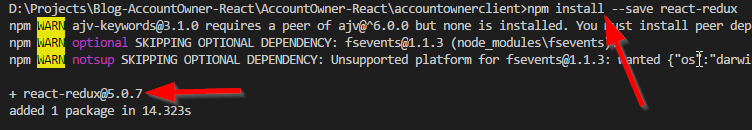
Redux is a state container for the JavaScript applications. Even though it could seem a little hard to learn at the beginning, after some practice you are going to see how easy it is and how much beneficial it is as well. Redux is not React’s library, it can fit with any other JavaScript framework, but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

npm install --save redux



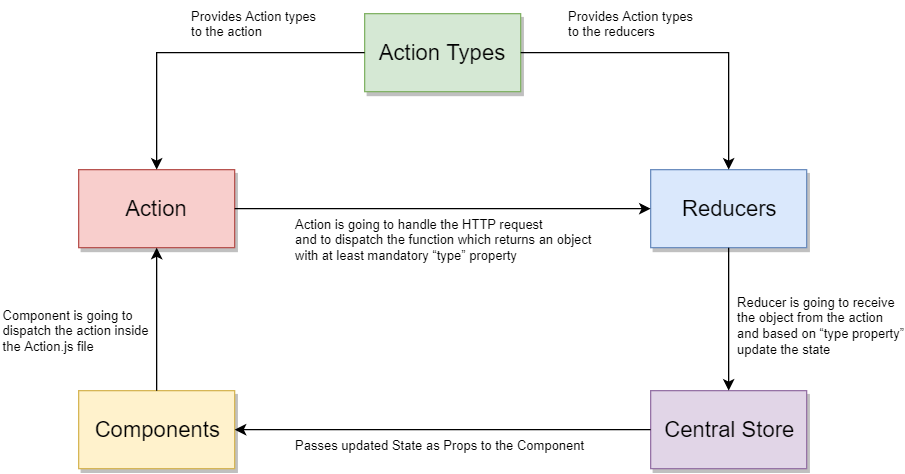
The Redux on its own is not going to be enough for the React project. We need to create a relation between the React and the Redux and to accomplish that let’s install the react-redux library:

npm install --save react-redux



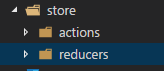
Excellent.

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 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

})

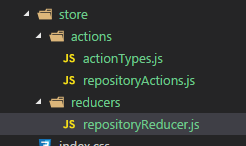
}

}

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 importing 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 the 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

}

}

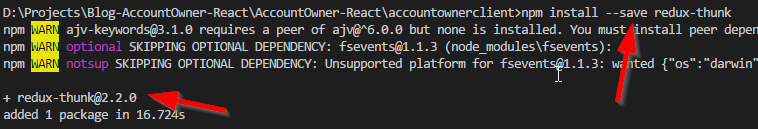
All of these functions are updating our state. Firstly we are deeply cloning our state object by using spread (…) operator and then just overriding the property we want to update in our state object. Because objects and arrays are referential types you must clone them deeply prior to any changes. That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

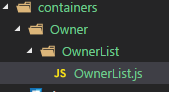
There it is, we have prepared our redux setup for our project. The best part about it is that you can use it for any other component inside your project and if you have components which require more complex or different kind of redux setup, all you have to do is to create another action and reducer file 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 HOC Component

In the previous post, we have set up the Redux environment completely and created the Axios instance to send 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.

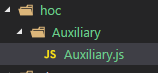
## OwnerList Component Creation, Hoc, and Routes

In the containers folder, let’s create a following folder structure with the OwnerList.js file inside:



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 it. 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 a project.

So, inside the src folder let’s create a new folder and name it hoc. In that folder, we are going to create a new one and name it Auxiliary. Finally, let’s create a new file Auxiliary.js.



Now, we need to modify the Auxiliary.js file:

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

export default auxiliary;

All we are doing 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 name owners which is going to hold all the owners from the database. Above the table, we may 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 display all the data from the owners array, even though it is an empty array for now.

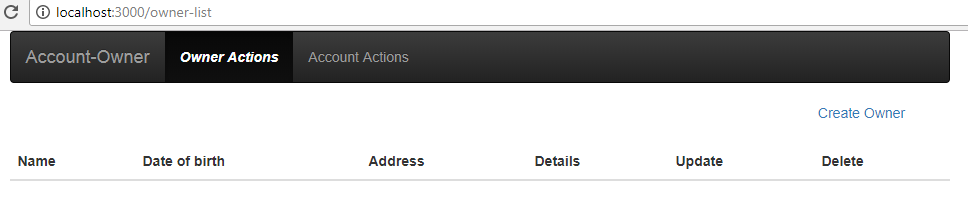
Let’s continue by modifying the App.js file to add the route to this component:

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

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

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

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

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

## Implementing Redux

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

First, 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, let’s 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, we are going to 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.

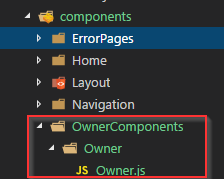
The mapDispatchToProps function is creating additional property onGetData which again we may call with the statement: this.props.onGetData. Then, it is going to dispatch the action inside the repositoryActions.js file which is going to fetch the data from the server.

*Right now if we 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 modifying 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

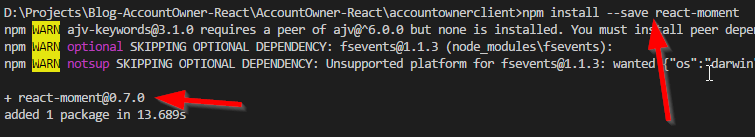
Because we are going to display multiple owners inside the OwnerList component, it is the best practice to create the Owner component and just to import it into the OwnerList component. So, let’s do exactly that.

In the components folder create the following structure:



To work 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’s data and create a row with the data and couple of buttons for the navigation to different components. All buttons are referencing to functions which enable redirection towards the details, update and delete components.

It is very important to notice 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 the 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 the Owners

Let’s import the Owner component inside the OwnerList component:

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

Then above the render() part, we are going to create a new function with the call to the OnGetData property 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 to populate 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} />

)

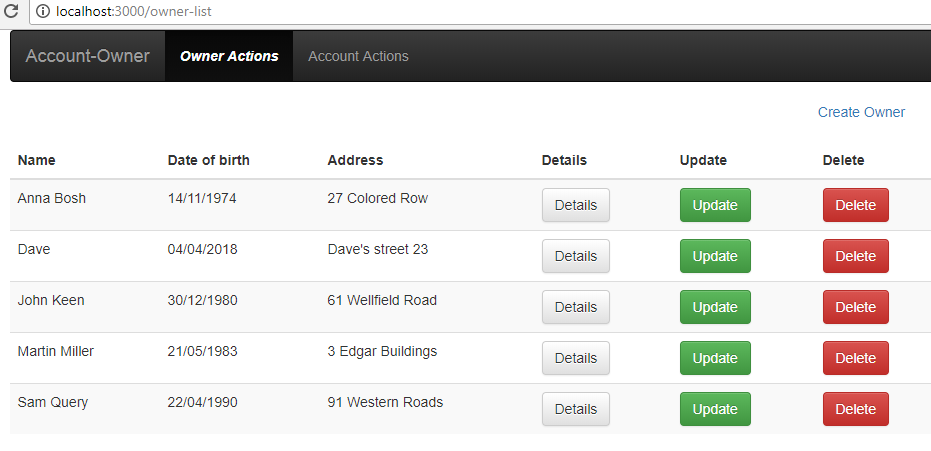
})

}

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 to having access to the “history” property of that object and a single owner object for child component to be able to use props.owner.

Excellent.

Now if you navigate to the Owner Actions menu we 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 let’s create a new folder AsyncComponent and inside let’s 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, we need to 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>

);

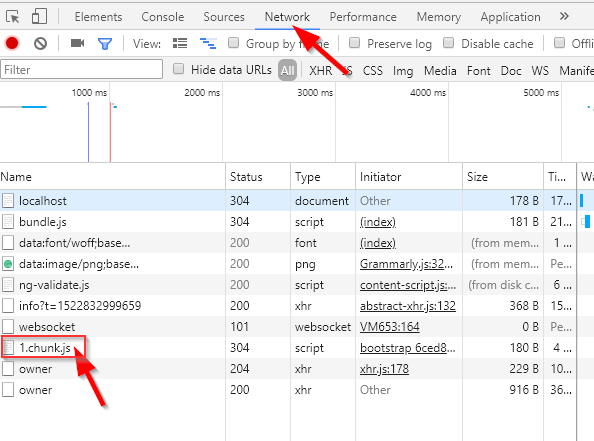
}

}

export default App;

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 we navigate, after restarting our app, to the Owner Actions page we are going to see additional chunk file loaded just for this page.

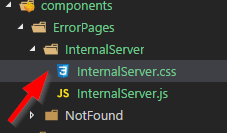


# Error Handling With Redux

Handling errors while sending HTTP requests towards our 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 errors which are different than 404 or 500 is going to be executed by using 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

Let’s create a new InternalServer folder inside the ErrorPages folder, and in there let’s create two new files and name them InternalServer.js and InternalServer.css:



We are going to 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, let’s modify the InternalServer.css file:

.internalServer{

font-weight: bold;

font-size: 50px;

text-align: center;

color: #c72d2d;

}

Finally, let’s 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 path="\*" component={NotFound} />

Excellent.

The 500 error component is ready and we may continue on.

## Implementing 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.

We may handle errors in different ways, but we want to show how to work with multiple reducers in a project. Moreover, centralizing the error handling actions is a good practice overall.

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

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 the actions folder, we have the repositoryActions.js file. Now we need to add one more action file errorHandlerActions.js inside the actions folder. So, let’s 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 the code above, we export the action handleHTTPError in which we check the status of the error and execute the corresponding function. We may notice that this is the same thing we did with the repositoryActions.js file.

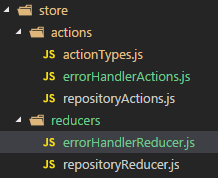
Inside the repositoryActions.js file, we need to import this errorHandlerActions.js file:

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

And to replace all the comments with this line of code:

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

Let’s continue by creating a new reducer file errorHandlerReducer.js inside the reducers folder. Right now we have this folder structure:



The errorHandlerReducer.js file should look like this:

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 create the state (initialState) object 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 import errorHandlerReducer file and also the combineReducers functions. Then with a help of the combineReducers function, we create the rootReducers object which contains all of our reducers. Finally, 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 the data property from the repositoryReducer file. Let’s 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.

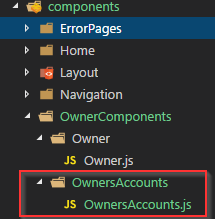
Excellent.

We can try this by modifying the code in the server’s method GetAllOwners. As a first line of a code, we may add return NotFound() or return StatusCode(500, “Some message”), and we 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 the OwnerDetails component and the child component is going to be the OwnersAccounts component. So, let’s start by creating a child component first.

For the OwnersAccounts component, we are going to create this structure:



Let’s 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>

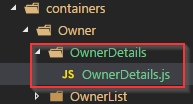
)

}

export default ownersAccounts;

We are doing great.

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



And 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, let’s 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;

In the code above, we are using the this.props.data statement but we haven’t implemented a 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 closing tag of 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';

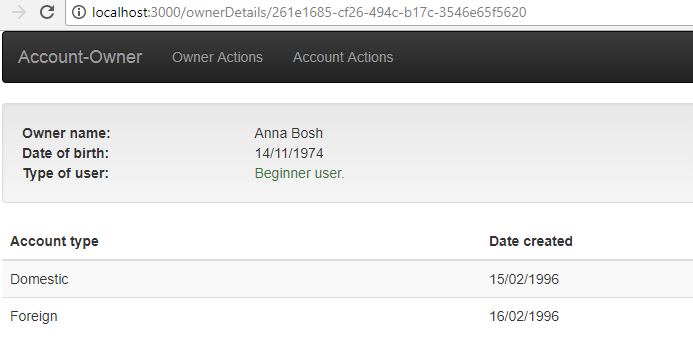
this.props.onGetData(url, { ...this.props })

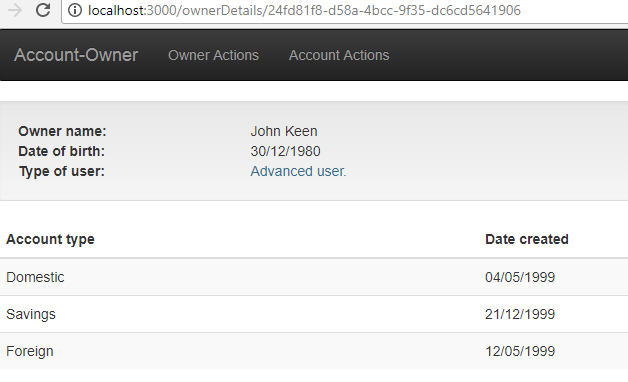
}

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:





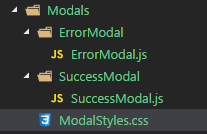
# Dynamic Form Creation and Modal Components

As the title suggests, in this post we are going to create a new form component for the post actions and modal components to display success and error messages. We are going to reuse them in every parent component that needs those modal components. Furthermore, we are going to create all the input fields from the config file dynamically, therefore extracting this part for reuse is our goal as well.

So, let’s start.

## Success and Error Modal Components

In the component folder, let’s create this folder structure:



First, we are going to modify the SuccessModal.js file:

import React from 'react';

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

import { Modal, Button } from 'react-bootstrap';

import '../ModalStyles.css';

const successModal = (props) => {

return (

<Aux>

<Modal show={props.show} backdrop='static'>

<Modal.Header>

{props.modalHeaderText}

</Modal.Header>

<Modal.Body>

<p>{props.modalBodyText}</p>

</Modal.Body>

<Modal.Footer>

<Button bsStyle="success" onClick={props.successClick}>{props.okButtonText}</Button>

</Modal.Footer>

</Modal>

</Aux>

)

}

export default successModal;

This code is pretty straightforward. We use the react-bootstrap components for the modal inside the functional component. Through the props, we send different parameters for our modal and one event as well. This event is going to close this modal once we click on the button.

Let’s modify the ErrorModal.js file in the same manner:

import React from 'react';

import { Modal, Button } from 'react-bootstrap';

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

import '../ModalStyles.css';

const errorModal = (props) => {

return (

<Aux>

<Modal show={props.show} backdrop='static'>

<Modal.Header>

{props.modalHeaderText}

</Modal.Header>

<Modal.Body>

<p>{props.modalBodyText}</p>

</Modal.Body>

<Modal.Footer>

<Button bsStyle="danger" onClick={props.closeModal}>{props.okButtonText}</Button>

</Modal.Footer>

</Modal>

</Aux>

)

}

export default errorModal;

Finally, we need to modify the ModalStyles.css file:

.modal-header {

margin: 0;

line-height: 1.42857143;

font-size: 30px;

text-align: center;

}

.modal-body p {

text-align: center;

margin-top: 10px;

}

@media (min-width: 768px){

.modal-dialog {

width: 500px !important;

margin: 20% auto !important;

}

}

Excellent.

Now we have our modal components which we can use for the Create, Update or any other component. I have deliberately created two modal components, even though they have almost the same code because it is much more readable when we use these modal components inside the parent components. You can distinguish right away what is the purpose of the modal by just looking at its name.

## Creating Input Configuration

Inside the src folder, we are going to create the Utility folder and inside it a new file InputConfiguration.js. In this file, we are going to store all of our configuration for the input elements which then we are going to use in the create and update forms (name, address, dateOfBirth).

Prior to modifying the InputConfiguration.js file, let’s install the Moment.js library because we are going to need it for the drop down control. So, let’s execute the npm command:

npm install --save moment

Now let’s modify the InputConfiguration.js file:

import moment from 'moment';

export const returnInputConfiguration = () => {

return {

name: {

element: 'input', type: 'text', value: '',

validation: { required: true }, valid: false, touched: false,

errorMessage: '', label: 'Name:'

},

address: {

element: 'input', type: 'text', value: '',

validation: { required: true, maxLength: 60 }, valid: false, touched: false,

errorMessage: '', label: 'Address:'

},

dateOfBirth: {

element: 'datePicker', type: 'text', value: moment(),

valid: true, touched: false,

errorMessage: '', label: 'Date of birth:'

}

}

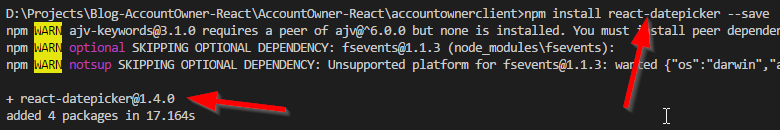
}

Later on, in our create component, we are going to transform this object into an array of objects and send it to the Input component to create all of the input fields we need on the form. That array will consist of the objects (key-value pairs) where the key is going to be the name or the address or the dateOfBirth (properties from above object) and the value is going to be the complete configuration part of the above object (type, value, element…).

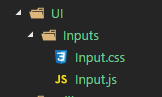
## Creating Inputs Dynamically

The first thing we need to do is to install the react-datepicker library because we are going to use it for the dateOfBirth control. Let’s execute the command:

npm install react-datepicker --save



In the src folder let’s create the UI folder. Inside it, we are going to create a new folder Input and inside it the Input.js and the Input.css files:



The Input.js file is going to be a functional component so let’s modify it accordingly:

import React from 'react';

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

import { FormGroup, Col, FormControl, ControlLabel } from 'react-bootstrap';

import DatePicker from 'react-datepicker';

import 'react-datepicker/dist/react-datepicker.css';

import './Input.css';

const input = (props) => {

let inputField = null;

let errorMessage = null;

if(props.invalid && props.shouldValidate && props.touched){

errorMessage = (<em>{props.errorMessage}</em>);

}

return (

<Aux>

{inputField}

</Aux>

)

}

export default input;

For now, we just import all the resources we need, initialize the input field and the error message and set up the value for that message if the control is invalid and if it should be validated and if it is touched (We don’t want to show the error if the user didn’t place the mouse cursor inside that component at all). Lastly, we are returning that input field (which is null for now) to the parent component.

Below the if statement and above the return block, we are going to add the code for populating the input field property:

switch (props.elementType) {

case 'input':

inputField = (

<FormGroup controlId={props.id}>

<Col componentClass={ControlLabel} sm={2}>

{props.label}

</Col>

<Col sm={6}>

<FormControl type={props.type} value={props.value} onChange={props.changed} onBlur={props.blur} />

</Col>

<Col>

<em>{errorMessage}</em>

</Col>

</FormGroup>

)

break;

case 'datePicker':

inputField = (

<FormGroup controlId={props.id}>

<Col componentClass={ControlLabel} sm={2}>

{props.label}

</Col>

<Col sm={6}>

<DatePicker selected={props.value} dateFormat="MM/DD/YYYY" readOnly

onChange={props.changed} className='datePickerControl'

showYearDropdown dropdownMode="select"/>

</Col>

<Col>

<em>{errorMessage}</em>

</Col>

</FormGroup>

)

break;

default: inputField = null;

}

So, we switch through the element type and if it is the input type we create the input field with all the properties and events it needs. We are doing the same thing for the datePicker control. We need only those two types of controls for our forms, but if you for any of your projects need more controls, just simply add additional case statement.

We have left to modify the Input.css file:

em{

color: red;

margin: 5px 0;

}

.react-datepicker\_\_day-name, .react-datepicker\_\_day, .react-datepicker\_\_time-name {

display: inline-block;

width: 30px;

height: 30px;

line-height: 30px;

text-align: center;

margin: 0.166rem;

font-size: 14px;

}

.react-datepicker\_\_month-container select {

font-size: 12px;

}

.datePickerControl {

display: block;

width: 100%;

height: 34px;

padding: 6px 12px;

font-size: 14px;

line-height: 1.42857143;

color: #555;

background-color: #fff;

background-image: none;

border: 1px solid #ccc;

border-radius: 4px;

}

.react-datepicker\_\_input-container {

position: relative;

display: inline-block;

width: 100%;

}

.react-datepicker-wrapper {

display: inline-block;

width: 100%;

}

In here we are overriding some of the datePicker native classes and adding one custom class (.datePickerControl).

That’s it, we can now continue to the CreateOwner component.

## CreateOwner Component

In the containers folder and then inside the Owner folder, let’s create a new folder and name it CreateOwner. Inside create a new file CreateOwner.js.

Let’s start modifying this file:

import React, { Component } from 'react';

import { Form, Well, Button, FormGroup, Col } from 'react-bootstrap';

import { returnInputConfiguration } from '../../../Utility/InputConfiguration';

class CreateOwner extends Component {

state = {

ownerForm: {},

isFormValid: false

}

componentWillMount = () =>{

this.setState({ ownerForm: returnInputConfiguration() });

}

render() {

return (

<Well>

</Well>

)

}

}

export default CreateOwner;

In the code above, we import all the necessary react-bootstrap components and the returnInputConfiguration function from the InputConfiguration.js file. This component is class component or stateful component and in the componentWillMount lifecycle hook, we update our local state with all the form configuration. The compnentWillMount hook is going to trigger immediately before the component mounts.

Let’s add another line of code between the render and return blocks:

const formElementsArray = formUtilityActions.convertStateToArrayOfFormObjects({ ...this.state.ownerForm });

As the name of the function indicates, we want to convert the ownerForm object into an array of objects to send it to the Input component. So, let's add that function in a separate file.

Inside the Utility folder, create a new file FormUtility.js. Modify that file by adding the function for the object transformation:

export const convertStateToArrayOfFormObjects = (formObject) => {

const formElementsArray = [];

for (let key in formObject) {

formElementsArray.push({

id: key,

config: formObject[key]

});

}

return formElementsArray;

}

Now we need to import this function inside CreateOwner component:

import \* as formUtilityActions from '../../../Utility/FormUtility';

We are going to have more actions inside FormUtility.js file therefore, we are importing all of those actions in the CreateOwnerComponent.

We have the formElementsArray populated, so let's use it to send all the properties towards the Input component.

Let’s add this code in the Well tag:

<Form horizontal onSubmit={this.createOwner}>

{

formElementsArray.map(element => {

return <Input key={element.id} elementType={element.config.element}

id={element.id} label={element.config.label}

type={element.config.type} value={element.config.value}

changed={(event) => this.handleChangeEvent(event, element.id)}

errorMessage={element.config.errorMessage}

invalid={!element.config.valid} shouldValidate={element.config.validation}

touched={element.config.touched}

blur={(event) => this.handleChangeEvent(event, element.id)} />

})

}

<br />

</Form>

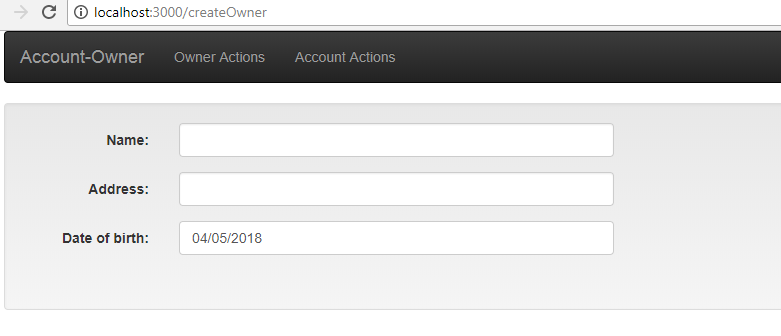
In this code, we are looping through all the objects (form control configurations) inside the formElementsArray and returning the Input component with all the necessary properties and events it requires. There is the function handleChangeEvent, and we are going to create this function to enable validation and two-way binding. We are going to do that little bit later.

To see the result of our current actions, modify the App.js file by adding another route to the CreateOwner component below the OwnerDetails route. Don’t forget the import as well.

import CreateOwner from './Owner/CreateOwner/CreateOwner';

<Route path="/createOwner" component={CreateOwner} />

If we navigate to the CreateOwner page we are going to see this result:



Fantastic.

We have our controls, even though we can’t do anything with them yet. But soon as we implement the handleChangeEvent function, we are going to be able to modify and validate our inputs.

## Finalizing the CreateOwner View

Let’s add a button to our component, to finalize the view part.

Inside the Form tag and below the <br> tag add this code:

<FormGroup>

<Col mdOffset={6} md={1}>

<Button type='submit' bsStyle='info' disabled={!this.state.isFormValid}>Create</Button>

</Col>

<Col md={1}>

<Button bsStyle='danger' onClick={this.redirectToOwnerList}>Cancel</Button>

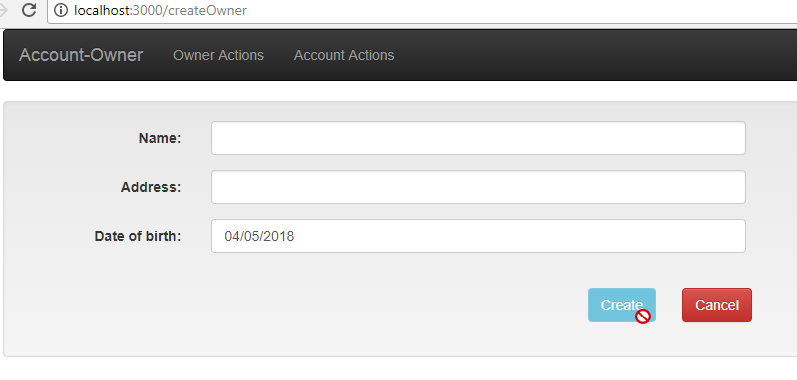
</Col>

</FormGroup>

We are adding two buttons, and the Create button is disabled as long as the form is invalid.

Excellent.

Now our form looks like this:



# Form Validation and Sending POST Request

As a continuation of the previous part, we are going to implement form validation and to create a new owner entity by sending the POST request towards our .NET Core Web API server.

In the CreateOwner component, we are passing the handleChangeEvent() function to the Input component. We are going to use this function to enable two-way binding and to validate our input fields. Therefore, its implementation and sending POST request is what we aim for in this post.

## Implementing handleChangeEvent

Right now if we type something inside the input fields nothing is going to happen. That is because, in our InputConfiguration.js file all “value” properties are set to empty string for all the inputs (except for the datePicker, it accepts the moment() function which always applies the current date). More important is that we never change the value property inside the configuration object, thus the empty input fields.

We are going to fix this and add the validation as well, by implementing the handleChangeEvent function.

Let’s modify the CreateOwner.js file by adding the handleChangeEvent function below the componentWillMount function:

handleChangeEvent = (event, id) => {

const updatedOwnerForm = { ...this.state.ownerForm };

updatedOwnerForm[id] = formUtilityActions.executeValidationAndReturnFormElement(event, updatedOwnerForm, id);

const counter = formUtilityActions.countInvalidElements(updatedOwnerForm);

this.setState({ ownerForm: updatedOwnerForm, isFormValid: counter === 0 })

}

This function accepts two parameters. The event parameter holds the value from our input fields (not only the value but this one is important for us) and the id parameter to which we are going to pass one of the configuration object keys (name, address or dateOfBirth).

We are deeply cloning the ownerForm from the state, and then updating the configuration of the currently modified input by calling the executeValidationAndReturnFormElement function. After that validation, we are counting how many fields are invalid, and if any, our complete form is going to be invalid. Finally, we are updating the state.

## Creating Additional Functions

We have all these functions called (executeValidationAndReturnFormElement and countInvalidElements) but they are not created yet.

So, let’s create them.

Let’s modify the FormUtility.js file:

export const executeValidationAndReturnFormElement = (event, updatedOwnerForm, id) => {

let formElement = { ...updatedOwnerForm[id] };

formElement.value = id === 'dateOfBirth' ? event : event.target.value;

formElement.touched = true;

const validationResponse = checkValidity(formElement.value, formElement.validation);

formElement.valid = validationResponse.isValid;

formElement.errorMessage = validationResponse.errorMessage;

return formElement;

}

First, we are cloning the exact formElement which user currently modifies. Then set up the value by using the ternary operator (If a user is changing the dateOfBirth input, then the event is the value. But for any other input field, the event is the object that has the target property which holds the value property). Furthermore, we set our control to touched and then send the value and the validation rules to the validation process. Once the process is done we store those results. Finally, we return formElement with all the changes.

So, lets implement checkValidity function just above the executeValidationAndReturnFormElement function:

const checkValidity = (value, validation) => {

let validationObject = {

isValid: true,

errorMessage: ''

};

if (validation) {

if (validation.required) {

validationObject.isValid = value.trim() !== '';

validationObject.errorMessage = validationObject.isValid ? '' : 'Field is required';

}

if (validationObject.isValid && validation.maxLength) {

validationObject.isValid = value.length <= 60;

validationObject.errorMessage = 'Not allowed more than 60 charactes';

}

return validationObject;

}

else {

return validationObject;

}

}

With the code above we check do we need to validate the input at all. If it needs to be validated then we perform validation but if it doesn’t (like dateOfBirth) we just return the same object back.

In the validation process, we check does the field has the required validation rule and validate it. Then if it is still valid we validate against the maxLength rule. Finally, we just return the validationObject.

## Validating the Entire Form

Inside the handleChangeEvent function, we are calling one additional function the countInvalidElements. So let’s implement that one just below the executeValidationAndReturnFormElement function in the FormUtility.js file:

export const countInvalidElements = (ownerForm) => {

let countInvalidElements = 0;

for (let element in ownerForm) {

if (!ownerForm[element].valid) {

countInvalidElements = countInvalidElements + 1;

break;

}

}

return countInvalidElements;

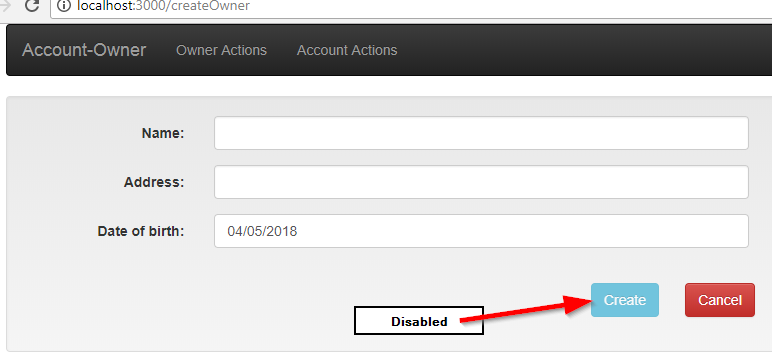
}

This function is pretty straightforward. We loop through all the elements inside the ownerForm. And if we find any element which is invalid, we just increase the countInvalidElements variable and leave the loop. Finally, we just return that variable.

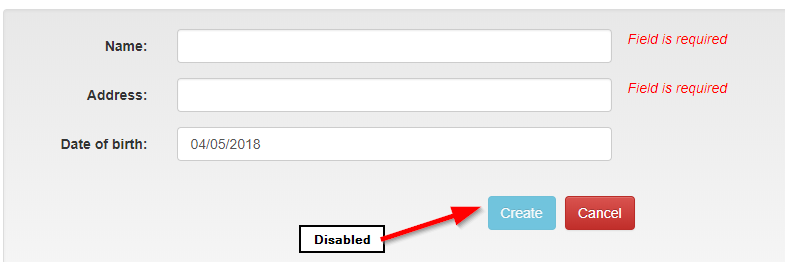
Excellent.

Now we can inspect the results.

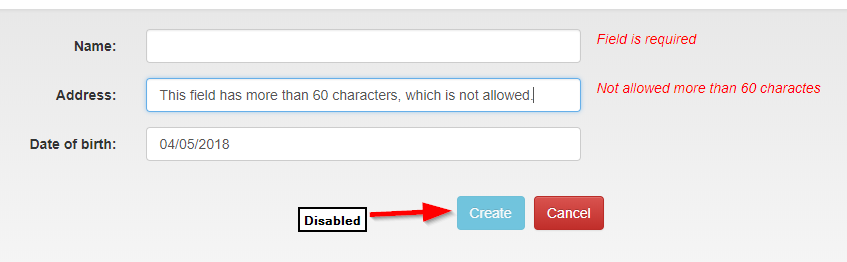
Empty form (never touched):



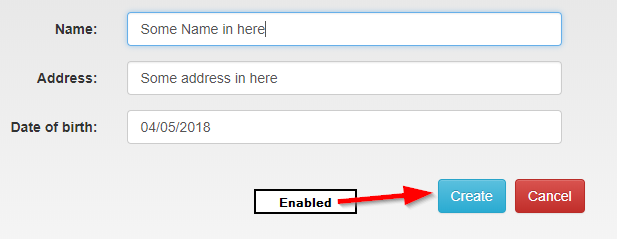
If we place our cursor inside an input field and then just leave:



Another type of error message:



Finally, the valid form:



## Connect the CreateOwner Component With the Reducer

Let’s execute the familiar actions to connect our component to the reducer file, so we could send the POST request.

First, let’s add all the necessary imports:

import { connect } from 'react-redux';

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

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

Second, let’s create mapStateToProps function bellow the component:

const mapStateToProps = (state) => {

return {

showSuccessModal: state.repository.showSuccessModal,

showErrorModal: state.errorHandler.showErrorModal,

errorMessage: state.errorHandler.errorMessage

}

}

After that, we need to create mapDispatchToProps function below the mapStateToProps:

const mapDispatchToProps = (dispatch) => {

return {

onCreateOwner: (url, owner, props) => dispatch(repositoryActions.postData(url, owner, props)),

onCloseSuccessModal: (url, props) => dispatch(repositoryActions.closeSuccessModal(props, url)),

onCloseErrorModal: () => dispatch(errorHandlerActions.closeErrorModal())

}

}

Then, let’s modify the export statement:

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

Finally, we are going to add our modal components below the FormGroup tag but still inside the Well tag:

<SuccessModal show={this.props.showSuccessModal}

modalHeaderText={'Success message'}

modalBodyText={'Action completed successfylly'}

okButtonText={'OK'}

successClick={() => this.props.onCloseSuccessModal('/owner-List', { ...this.props })} />

<ErrorModal show={this.props.showErrorModal}

modalHeaderText={'Error message'}

modalBodyText={this.props.errorMessage}

okButtonText={'OK'} closeModal={() => this.props.onCloseErrorModal()} />

And, add import statements as well:

import SuccessModal from '../../../components/Modals/SuccessModal/SuccessModal';

import ErrorModal from '../../../components/Modals/ErrorModal/ErrorModal';

The SuccessModal is calling the onCloseSuccessModal property from the mapDispatchToProps function. Moreover, the ErrorModal is calling the onCloseErrorModal property from the mpaDispatchToProps function as well. Both of those properties are dispatching functions which are not implemented. So this is something we need to fix.

## Modifying the Action and the Reducer files

We need to modify our files by adding additional functions for closing success and error modal components.

Let’s add two more action types inside the ActionTypes.js file:

export const CLOSE\_ERROR\_MODAL = 'CLOSE\_ERROR\_MODAL';

export const CLOSE\_SUCCESS\_MODAL = 'CLOSE\_SUCCESS\_MODAL';

Then let’s modify the repositoryActions.js file by adding closeSuccessModal function:

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

return {

type: actionTypes.CLOSE\_SUCCESS\_MODAL,

props: props,

url: url

}

}

We need to modify the repositoryReducer.js file as well:

case actionTypes.CLOSE\_SUCCESS\_MODAL:

return executeCloseSuccessModal(state, action)

const executeCloseSuccessModal = (state, action) => {

action.props.history.push(action.url);

return {

...state,

showSuccessModal: false

}

}

We are done with the success modal, let’s do the similar thing for the error modal.

First let’s modify the errorHandlerActions.js file by adding a new function:

export const closeErrorModal = () => {

return {

type: actionTypes.CLOSE\_ERROR\_MODAL

}

}

Then, let’s modify the errorHandlerReducer.js file:

case actionTypes.CLOSE\_ERROR\_MODAL:

return executeCloseErrorModal(state, action);

const executeCloseErrorModal = (state, action) => {

return{

...state,

showErrorModal: false,

errorMessage: ''

}

}

Now, all we have left to do is to send the POST request from the CreateOwner component.

There is onSubmit event inside the Form tag, which points to the createOwner function. Let’s implement that function below the handleChangeEvent function:

createOwner = (event) => {

event.preventDefault();

const ownerToCreate = {

name: this.state.ownerForm.name.value,

address: this.state.ownerForm.address.value,

dateOfBirth: this.state.ownerForm.dateOfBirth.value

}

const url = '/api/owner';

this.props.onCreateOwner(url, ownerToCreate, { ...this.props });

}

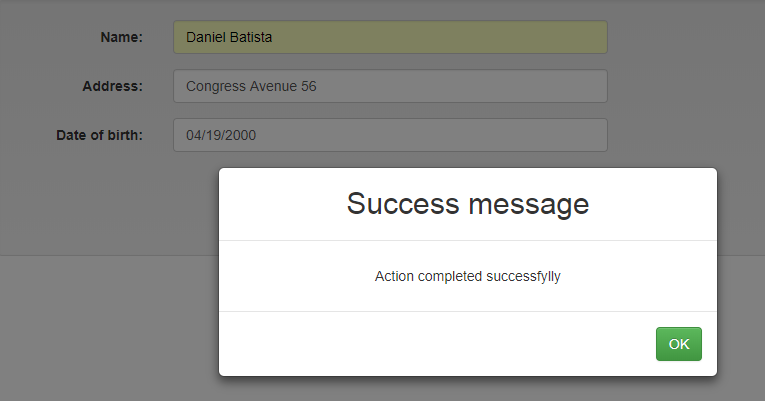
Bellow this function we need to add one more function for the Cancel button functionality:

redirectToOwnerList = () => {

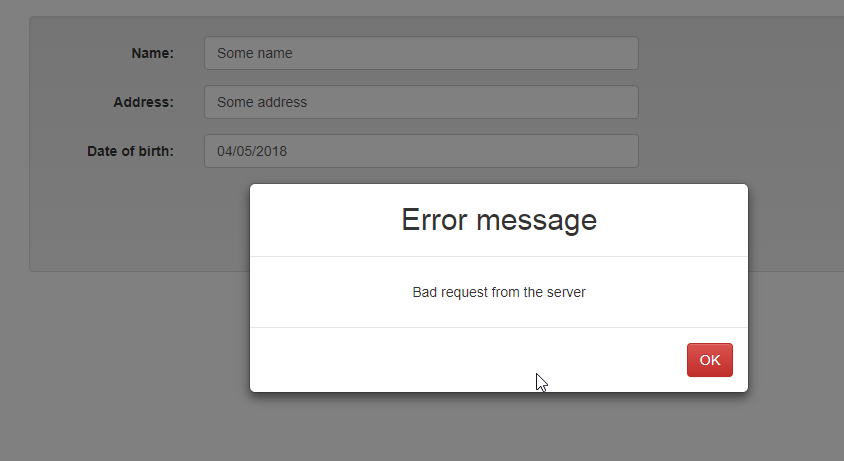
this.props.history.push('/owner-List');

}

Now we can check out the results:



If the error occurs, with the status code other than 404 or 500:



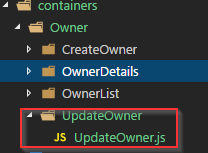
# Handling PUT Request in React Project

In the previous part, we have covered many different React features. The way to create the input controls from the configuration object, and validation of those controls are just some of many features we have covered. These mentioned features are going to be useful to us in this part of the series where we send the PUT request towards the server in order to update our entity object.

So, let’s dive right into it.

## Preparations for the Update Owner Component and the Update Route

Inside the containers folder, we can find the Owner folder. Inside that folder we are going to create a new one and name it UpdateOwner. Let’s create a new file UpdateOwner.js inside:



In this file, we are going to handle a business logic to update the owner entity by reusing many features we have already used in the CreateOwner component. Therefore, this business logic shouldn’t be too hard to understand because we already have all the necessary knowledge (from a previous part) about input fields, validation, and a state object.

Let’s add the basic logic to the UpdateOwner.js file:

import React, { Component } from 'react';

import { Form, Well, Button, FormGroup, Col } from 'react-bootstrap';

import { returnInputConfiguration } from '../../../Utility/InputConfiguration';

class UpdateOwner extends Component {

state = {

ownerForm: {},

isFormValid: true

}

componentWillMount = () => {

this.setState({ ownerForm: returnInputConfiguration() });

}

render() {

return (

<Well>

</Well>

)

}

}

export default UpdateOwner;

So, we create the class component with its local state. This state has the same properties as the CreateOwner component: the ownerForm, and isFormValid. For the UpdateOwner component, we need to set the isFormValid to true because all the fields on the form are going to be populated as soon as the component mounts. Therefore, our fields are not invalid when form mounts as they were in the CreateOwner component.

In the compnentWillMount lifecycle hook, we just fetch all the input configuration and update the state.

To enable navigation to this component we need to modify the App.js file:

import UpdateOwner from './Owner/UpdateOwner/UpdateOwner';

<Route path="/updateOwner/:id" component={UpdateOwner} />

Now we are able to navigate to this component.

## Adding Input Elements, Validation, and Two-Way Binding

Right below the render function and above the return block we are going to add this line of code to convert the ownerForm object to an array of objects:

const formElementsArray = formUtilityActions.convertStateToArrayOfFormObjects({ ...this.state.ownerForm });

Don’t forget to add the import statement for the formUtilityActions and the Input component as well:

import \* as formUtilityActions from '../../../Utility/FormUtility';

import Input from '../../../UI/Inputs/Input';

Then, inside the Well tag, let’s add this code to display the input elements and buttons:

<Form horizontal onSubmit={this.updateOwner}>

{

formElementsArray.map(element => {

return <Input key={element.id} elementType={element.config.element}

id={element.id} label={element.config.label}

type={element.config.type} value={element.config.value}

changed={(event) => this.handleChangeEvent(event, element.id)}

errorMessage={element.config.errorMessage} invalid={!element.config.valid}

shouldValidate={element.config.validation}

touched={element.config.touched}

blur={(event) => this.handleChangeEvent(event, element.id)} />

})

}

<br />

<FormGroup>

<Col mdOffset={6} md={1}>

<Button type='submit' bsStyle='info' disabled={!this.state.isFormValid}>Update</Button>

</Col>

<Col md={1}>

<Button bsStyle='danger' onClick={this.redirectToOwnerList}>Cancel</Button>

</Col>

</FormGroup>

</Form>

This is familiar code which we have used in a previous post. Moreover, let’s add the handleChangeEvent function (below the componentWillMount lifecycle hook) to enable validation and the two-way binding:

handleChangeEvent = (event, id) => {

const updatedOwnerForm = { ...this.state.ownerForm };

updatedOwnerForm[id] = formUtilityActions.executeValidationAndReturnFormElement(event, updatedOwnerForm, id);

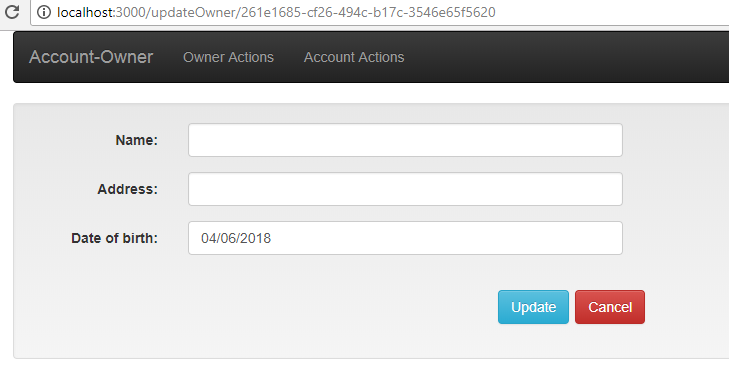
const counter = formUtilityActions.countInvalidElements(updatedOwnerForm);

this.setState({ ownerForm: updatedOwnerForm, isFormValid: counter === 0 })

}

Again this is familiar code where we reuse the code for the validation and the two-way binding.

At this moment when we click on the Update button in the OwnerList component, we are going to see the UpdateOwner view:



Of course, this is not enough for us. We want to populate all the fields with correct owner’s data.

So, let’s do exactly that.

## Connecting the Reducer With the Component

Let’s add all the necessary import statements:

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

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

import { connect } from 'react-redux';

import moment from 'moment';

Then we are going to add the mapStateToProps function below the component:

const mapStateToProps = (state) => {

return {

data: state.repository.data,

showSuccessModal: state.repository.showSuccessModal,

showErrorModal: state.errorHandler.showErrorModal,

errorMessage: state.errorHandler.errorMessage

}

}

Just below the mapStateToProps function, we need to add the mapDispatchToProps function:

const mapDispatchToProps = (dispatch) => {

return {

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

onUpdateOwner: (url, owner, props) => dispatch(repositoryActions.putData(url, owner, props)),

onCloseSuccessModal: (url, props) => dispatch(repositoryActions.closeSuccessModal(props, url)),

onCloseErrorModal: () => dispatch(errorHandlerActions.closeErrorModal())

}

}

Finally, let’s modify the export statement:

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

There it is.

We have now the connection between the reducers and our component. Now we can add our modal components to show success or error messages.

First, let’s add the import statements:

import SuccessModal from '../../../components/Modals/SuccessModal/SuccessModal';

import ErrorModal from '../../../components/Modals/ErrorModal/ErrorModal';

Then we need to add the components below the Form tag but inside the Well tag:

<SuccessModal show={this.props.showSuccessModal} modalHeaderText={'Success message'}

modalBodyText={'Action completed successfully'}

okButtonText={'OK'}

successClick={() => this.props.onCloseSuccessModal('/owner-List', { ...this.props })} />

<ErrorModal show={this.props.showErrorModal} modalHeaderText={'Error message'}

modalBodyText={this.props.errorMessage}

okButtonText={'OK'}

closeModal={() => this.props.onCloseErrorModal()} />

## Fetching the Data

Below the compnentWillMount lifecycle hook, we are going to add another hook to fetch the data from the server:

componentDidMount = () => {

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

const url = '/api/owner/' + id;

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

}

In this function, we take the id of the owner we want to update and then we fetch that owner from the server. Now we need to update our local state (to be more precise the ownerForm object inside that state) to show the data from the owner object in the input fields.

The componentWillMount and the componentDidMount are the **creation** lifecycle hooks. But React has the **update** lifecycle hooks which triggers as soon as a new property arrives at the component or we update the state.

So, if we take a look at the diagram where we explained the Redux flow, we will see that after the reducer updates the state, the central store is going to propagate that state as props inside the component. In that very moment with the update lifecycle hook, we can catch that change and update our ownerForm object inside our local state.

## Displaying Data on the Screen

Therefore, let’s add a new update lifecycle hook below the componentDidMount function:

componentWillReceiveProps = (nextProps) => {

const updatedOwnerForm = { ...this.state.ownerForm };

let nameObject = { ...updatedOwnerForm.name };

let dateObject = { ...updatedOwnerForm.dateOfBirth };

let addressObject = { ...updatedOwnerForm.address };

nameObject.value = nextProps.data.name;

nameObject.valid = true;

dateObject.value = moment(nextProps.data.dateOfBirth);

addressObject.value = nextProps.data.address;

addressObject.valid = true;

updatedOwnerForm['name'] = nameObject;

updatedOwnerForm['dateOfBirth'] = dateObject;

updatedOwnerForm['address'] = addressObject;

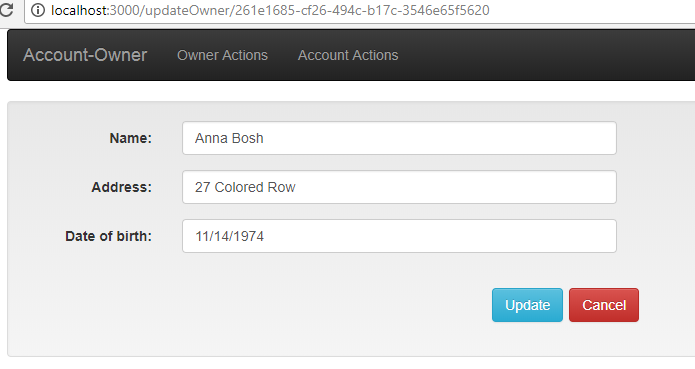
this.setState({ ownerForm: updatedOwnerForm });

}

The componentWillReceiveProps hook is called only when a component receives a new props object **and not** when we update the state. Why is this important to know? Well, some of the update lifecycle hooks are going to be triggered when new props arrive and when we update the state as well, therefore if we update the state inside that hook and not provide the exit condition (some type of if statement) we are going to end up with the infinite loop. But that’s not the case with the componentWillReceiveProps hook.

Inside the componentWillReceiveProps hook, we are not using this.props statement but the nextProps parameter because it contains our new props object. So, all we do in this function is extract the ownerForm from the state immutably inside updatedOwnerform. Then from the updatedOwnerForm, we extract all the other objects immutably as well (name, address, and dateOfBirth). Then we change the property values inside those objects and return those objects inside the updatedOwnerForm. Finally, we update the state with the new updatedOwnerForm.

Now our form has a different look once we navigate to it:



## Executing the Update Action

At the end let’s add two functions to redirect to the OwnerList component if we click the Cancel button, and to update the entity:

redirectToOwnerList = () => {

this.props.history.push('/owner-List');

}

updateOwner = (event) => {

event.preventDefault();

const ownerToUpdate = {

name: this.state.ownerForm.name.value,

dateOfBirth: this.state.ownerForm.dateOfBirth.value,

address: this.state.ownerForm.address.value

}

const url = "/api/owner/" + this.props.data.id;

this.props.onUpdateOwner(url, ownerToUpdate, {...this.props});

}

There we go. Now we can test the update functionality.

We can modify the server code to respond with the success and the error responses to test our modal components. Furthermore, we can test the form validation by emptying the input fields or typing more than 60 characters in the address input field.

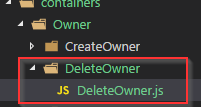
# Delete Owner

In this post, we are going to write a business logic for the DeleteOwner component in which we are going to delete the entity from the server. After we finish our job with this component, we are going to finish all the CRUD operations for this series.

Without further ado, let’s get into it.

## Preparations for the Delete Owner Component and the Update Route

Inside the Owner folder, create a new one and name it DeleteOwner. Inside create a new file DeleteOwner.js:



Now let’s modify the DeleteOwner.js file with basic setup code:

import React, { Component } from 'react';

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

class DeleteOwner extends Component {

render() {

return (

<Aux>

</Aux>

)

}

}

export default DeleteOwner;

To enable navigation to this component, we need to modify the App.js file:

import DeleteOwner from './Owner/DeleteOwner/DeleteOwner';

<Route path="/deleteOwner/:id" component={DeleteOwner} />

Right now if we click on the Delete button in the OwnerList component, the DeleteOwner page is going to open (even though it has no content inside).

## Connecting Our Component with the Reducer

First of all, we need to add the import statements:

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

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

import { connect } from 'react-redux';

The next thing we need to do is to add the mapStateToProps function right below the class:

const mapStateToProps = (state) => {

return {

data: state.repository.data,

showSuccessModal: state.repository.showSuccessModal,

showErrorModal: state.errorHandler.showErrorModal,

errorMessage: state.errorHandler.errorMessage

}

}

Then, let’s add the mapDispatchToProps function:

const mapDispatchToProps = (dispatch) => {

return {

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

onDeleteOwner: (url, props) => dispatch(repositoryActions.deleteData(url, props)),

onCloseSuccessModal: (url, props) => dispatch(repositoryActions.closeSuccessModal(props, url)),

onCloseErrorModal: () => dispatch(errorHandlerActions.closeErrorModal())

}

}

In this function, all the properties are self-explanatory and they dispatch the functions that are self-explanatory as well.

Finally, let’s modify the export statement:

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

## Implementing JSX in the DeleteOwner Component

To create our view of this component, first we need to add the import statements:

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

import Moment from 'react-moment';

Right between the render and return blocks, we need to add this code:

render() {

let owner = {...this.props.data};

return (

Inside the Aux tag, let’s add the content that we are going to display on the page:

<Row>

<Col md={10}>

<Well>

<Row>

<Col md={3}>

<ControlLabel htmlFor='name'>Owners name:</ControlLabel>

</Col>

<Col md={7}>

<span name='name'>{owner.name}</span>

</Col>

</Row>

<Row>

<Col md={3}>

<ControlLabel htmlFor='dateOfBirth'>Date of birth:</ControlLabel>

</Col>

<Col md={7}>

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

</Col>

</Row>

<Row>

<Col md={3}>

<ControlLabel htmlFor='address'>Address:</ControlLabel>

</Col>

<Col md={7}>

<span name='address'>{owner.address}</span>

</Col>

</Row>

</Well>

</Col>

</Row>

Below the Row tag add the following code, to display the buttons on the page:

<Row>

<Col mdOffset={8} md={1}>

<Button type="submit" bsStyle="info" onClick={this.deleteOwner}>Delete</Button>

</Col>

<Col md={1}>

<Button bsStyle='danger' onClick={this.redirectToOwnerList}>Cancel</Button>

</Col>

</Row>

Finally, add the import statements for the modal components and below the Row tag and right above the Aux tag, add the code for displaying our modal components:

import SuccessModal from '../../../components/Modals/SuccessModal/SuccessModal';

import ErrorModal from '../../../components/Modals/ErrorModal/ErrorModal';

<SuccessModal show={this.props.showSuccessModal} modalHeaderText={'Success message'}

modalBodyText={'Action completed successfylly'}

okButtonText={'OK'}

successClick={() => this.props.onCloseSuccessModal('/owner-List', { ...this.props })} />

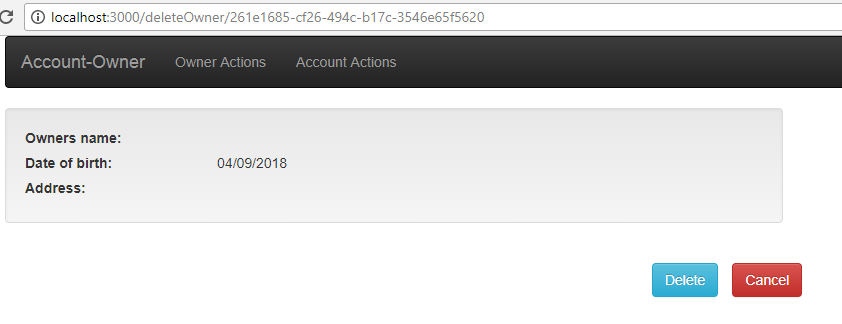
<ErrorModal show={this.props.showErrorModal} modalHeaderText={'Error message'}

modalBodyText={this.props.errorMessage}

okButtonText={'OK'}

closeModal={() => this.props.onCloseErrorModal()} />

Now, we can inspect the result:



As you may notice from the picture above, we don’t have the data on the page, but we are about to fix that.

## Implementing the Business Logic For the Delete Action

To fetch the owner object from the server we are going to add componentDidMount function right above the render function:

componentDidMount = () => {

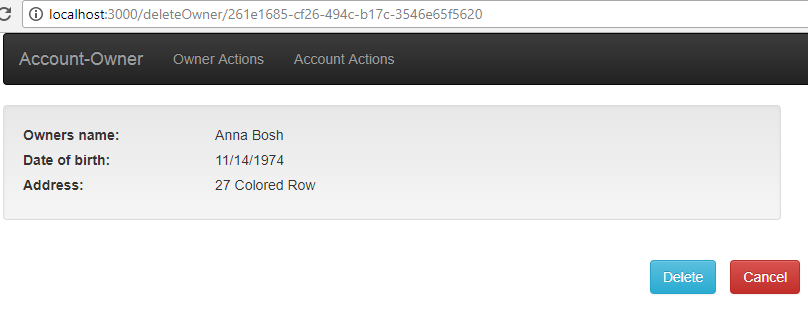
const id = this.props.match.params.id;

const url = '/api/owner/' + id;

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

}

Now if we navigate to the OwnerDelete component, we are going to see the page with the correct data:



For the Cancel button to work, let’s add this function:

redirectToOwnerList = () => {

this.props.history.push('/owner-List');

}

Finally, let’s implement the delete action:

deleteOwner = (event) => {

event.preventDefault();

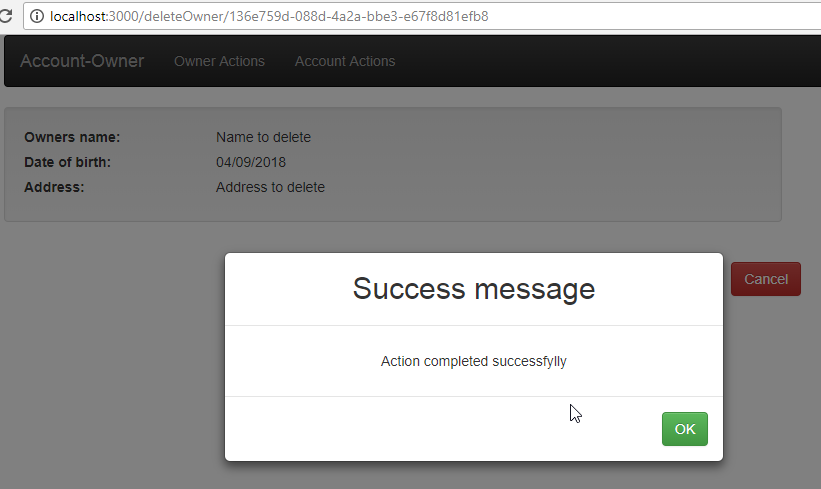
const url = "/api/owner/" + this.props.data.id;

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

}

Great job.

We can check the result:



If we click on the OK button, we are going to be directed to the OwnerList component.