|  |
| --- |
| WEB-XXX ReactJS |
| Workbook  (version 1.0, 05.22.2016) |

## Workbook tasks

### [Kick Off](#KickOff)

### [Hello World](#HelloWorld)

### [JSX](#JSX_new)

### [Grid Development](#Grid)

### [State vs Props](#StateVsProps)

### [Children](#Children_new)

### [React Router](#Router)

### [Redux](#Redux_new)

### [Testing React](#Testing)

Kick off. Install everything required to start development.

1. Install node.js: https://nodejs.org/en/
2. Mkdir project && cd project
3. Npm init and answer the questions
4. Go to terminal and install react and react-dom: npm install --save react react-dom

Hello World.

1. Create index.html page with:

<!doctype html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>  
<body>  
  
</body>  
</html>

1. Include react and react-dom scripts to the page and create react mount dom element

<!doctype html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <script src="node\_modules/react/dist/react.js"></script>  
 <script src="node\_modules/react-dom/dist/react-dom.js"></script>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>  
<body>  
 <app id="app">  
   
 </app>

<script src="js/app.js"></script>

</body>  
</html>

1. Create your js scripts folder and add initial js file: app.js (mkdir js && cd js)
2. Add to app.js next content:

ReactDOM.render(  
 React.createElement('h1', **null**, 'Hello, world!'),  
 document.getElementById('app')  
);

1. Open index.html in browser. Congrats! We just created our first React Component using pure JavaScript: 
2. Okay, nice, but lets make things a bit more complicated and create some very simple markup with div > h1 ul > li li > h2:

**var** app = React.createElement('div', {},  
 React.createElement('h1', {}, "Hi, I'm header inside div"),  
 React.createElement('ul', {},  
 React.createElement('li', {},  
 React.createElement('h2', {}, "Hi, I'm list item inside list inside div")  
 ),  
 React.createElement('li', {},  
 React.createElement('h2', {}, "Hi, I'm list item inside list inside div")  
 )  
 )  
);  
  
ReactDOM.render(  
 app,  
 document.getElementById('app')  
);

# Task 1: Create React.js Component with given HTML in pure JS

## Have some fun with Pure JS and React. Create next html structure with it:

<div>  
 <h1>I'm page header</h1>  
 <div>  
 <p>I'm staying at the begining of the page content</p>  
 <div>  
 <span>I'm user logo container</span>  
 <span>I'm user name container</span>  
 </div>  
 <h2>I'm next section header</h2>  
 <section>  
 <article>I'm awesome article</article>  
 <ul>  
 <li>I'm article item</li>  
 <li>I'm article item with <b>bold element</b></li>  
 </ul>  
 </section>  
 </div>  
</div>

JSX

1. Do you think it’s comfortable to write and read React.js pure JavaScript code? – No, No, No, No!
2. Lets move to jsx. For that we need to install WebPack and JSX transformer:

npm i webpack -g

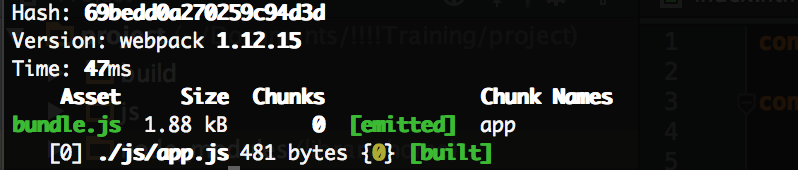
npm i babel-loader babel-core babel-preset-es2015 babel-preset-react --save-dev

1. Let me explain what happened here. We installed webpack globally (-g flag) to make sure it’s available everywhere (we could also point it every time to node\_modules/webpack/.bin/xxx). Why do we need it? Let’s go step by step:
   1. **Webpack**: Module bundler, with react used also as prebundler compiler tool from JSX to JS, ES6 to ES2015
   2. **Babel**-**core:** JSX -> JS, ES6 -> ES5 compiler itself
   3. **Webpack Babel loader:** Used by webpack to operate babel
   4. **Babel Presets:** Different preset for different kind of compilation target you are looking for. We gonna use React preset (JSX -> JS) and ES2015 (ES6-> ES5)
2. So far so good! Let’s setup webpack config. Create file webpack.config.js at root of you project and put it there:

**const** path = require('path');  
  
**const** PATHS = {  
 app: "./js/app",  
 build: path.join(\_\_dirname, 'build')  
};  
  
module.exports = {  
 entry: {  
 app: PATHS.app  
 },  
 output: {  
 path: PATHS.build,  
 filename: 'bundle.js'  
 }  
};

1. What’s happening here is that we say webpack to start bundling from our entry point and put bundle at output path with name bundle.js. Okay. We don’t have bundle.js yet, but let’s assume we got it. Let update our index.html:

<!doctype html>  
<html>  
<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <script src="node\_modules/react/dist/react.js"></script>  
 <script src="node\_modules/react-dom/dist/react-dom.js"></script>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>  
<body>  
 <app id="app">  
  
 </app>  
   
<script src="build/bundle.js"></script>  
</body>  
</html>

1. Okay, looks like we are ready to bundle the file. Type webpack in root of your project. You will see something like that:
2. Open index.html in browser. Nothing changed? Awesome! That’s exactly what we were looking for.
3. Okay, I’m assuming when you typed webpack 10 more times to rebuild you get tired of it. You can improve this process, just add build script in your package.json:

"scripts": {

"build": "webpack"

}

1. As of now package.json is created as readonly, it’s better to make it writable:

chmod a+rwx package.json

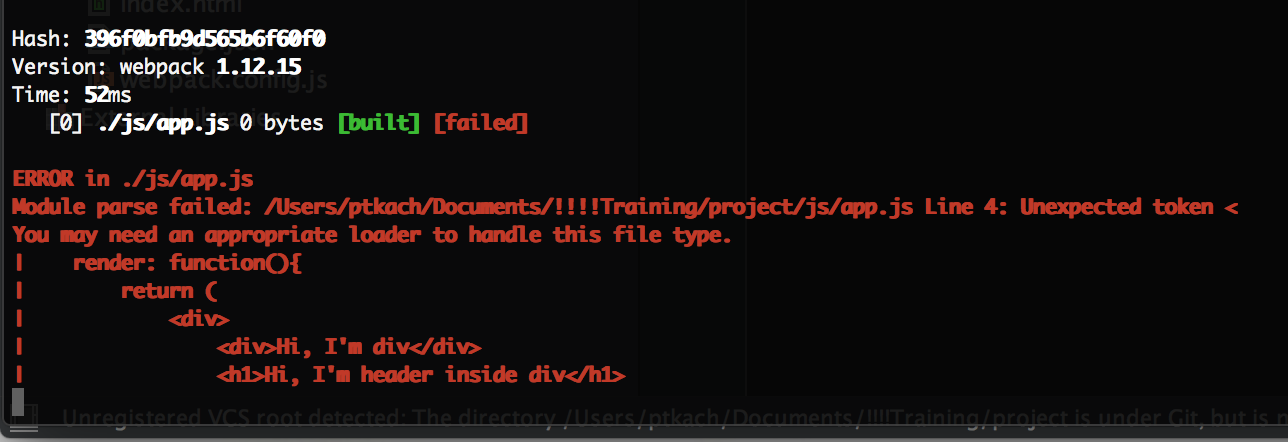
1. Now when you type “npm run build” in terminal you will build your JS. But what did we achieve? Instead of typing webpack we now type npm run build. Let’s make it more comfortable:

"build": "webpack --progress --colors --watch"

1. Okay, we configured webpack to watch for any changes in files and automatically rebuild the project. Since now you can type “npm run build” just before developing and webpack will do everything for you
2. How comfortable are you with with React.js markup? Does it look weird? Yes, I think so. Lets change it to JSX to make it more familiar:

**var** JsxApp = React.createClass({  
 render: **function**(){  
 **return** (  
 <div>  
 <div>Hi, I'm div</div>  
 <h1>Hi, I'm header inside div</h1>  
 <ul>  
 <li>Hi, I'm list item inside list inside div</li>  
 <li>Hi, I'm list item inside list inside div</li>  
 </ul>  
 </div>  
 )  
 }  
});  
ReactDOM.render(  
 <JsxApp></JsxApp>,  
 document.getElementById('app')  
);

1. Look into console. What you see supposed to look like:



1. Looks like we need some loader to let webpack work with JSX. Do you remember we installed one, lets set it up, go to webpack.config.js and put it there:

**const** path = require('path');  
  
**const** PATHS = {  
 app: "./js/app",  
 build: path.join(\_\_dirname, 'build')  
};  
  
module.exports = {  
 devtool: 'source-map',  
 entry: {  
 app: PATHS.app  
 },  
 module: {  
 loaders: [  
 {  
 test: /\.js$/,  
 exclude: /node\_modules/,  
 loader: 'babel',  
 query: {  
 presets: ['react']  
 }  
 }  
 ]  
 },  
 output: {  
 path: PATHS.build,  
 filename: 'bundle.js'  
 }  
};

1. Let me explained what happened here. We added 2 things:
   1. Devtool: we force webpack to build complete source-map for us (Just in case we need to debug something)
   2. Module: This is the most interesting part. We say webpack to use babel loader (which we installed earlier) with react preset (which we installed earlier) for any files \*.js excluding node\_modules folder. Why do we exclude node\_modules? Because according to npm policy any lib should be completely compatible with es5, so we don’t need to waste time on building node\_modules dependencies also
2. Okay lets restart webpack and open index.html in browser:



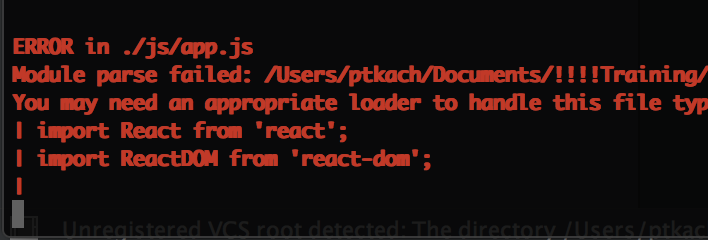
1. Awesome it works. So for now we have setup webpack to use JSX with Babel loader, we changes our JS to JSX (Which look much better) and transformed JSX to JS with webpack.
2. But what is wrong here? I think since we use webpack which gives us all power of the modules, we don’t need to load React and ReactDOM as a global scripts:

<head>  
 <meta charset="utf-8">  
 <title>Awesome React Project</title>  
 <script src="node\_modules/react/dist/react.js"></script>  
 <script src="node\_modules/react-dom/dist/react-dom.js"></script>  
 <!--<link rel="stylesheet" href="css/main.css">-->  
</head>

1. Let’s remove that and load it with webpack. Add that to your app.js:

**import** React **from** 'react';  
**import** ReactDOM **from** 'react-dom';

1. In console you might see:



1. It’s happening because we are trying to use ES6 import/export. We need separate loader for that:

presets: ['es2015', 'react']

1. And wooala:



1. So we connected webpack, babel + babel es2015, react presets, react and reactDOM modules and we are all set!

Grid Development

1. We will start development from moving to ES6 completely. So lets rewrite our app to it:

**import** React **from** 'react';  
**import** {render} **from** 'react-dom';  
  
**class** App **extends** React.Component {  
 render(){  
 **return** (  
 <div>  
 <div>Hi, I'm div</div>  
 <h1>Hi, I'm header inside div</h1>  
 <ul>  
 <li>Hi, I'm list item inside list inside div</li>  
 <li>Hi, I'm list item inside list inside div</li>  
 </ul>  
 </div>  
 )  
 }  
}  
  
render(  
 <App></App>,  
 document.getElementById('app')  
);

1. Let’s create something useful. We will start from creating Grid using Bootstrap. For that we need to install bootstrap and webpack css loader (Yes, webpack can bundle css =))

npm install --save bootstrap

npm install file-loader css-loader style-loader url-loader --save-dev

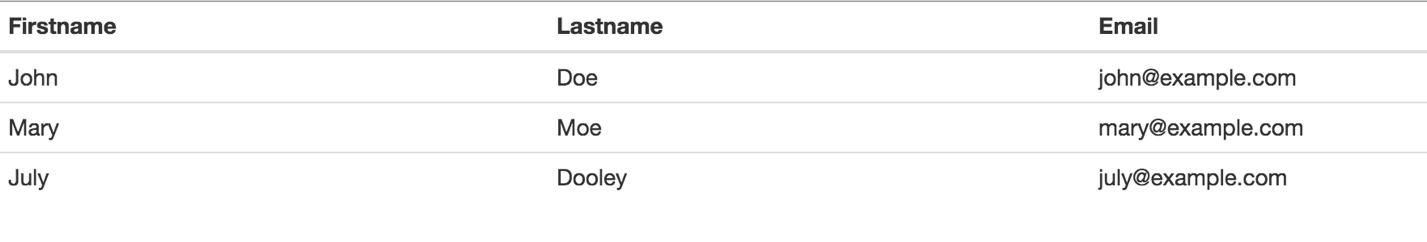
1. Update you webpack.config with new loaders:

loaders: [  
 {  
 test: /\.js$/,  
 exclude: /node\_modules/,  
 loader: 'babel',  
 query: {  
 presets: ['es2015', 'react']  
 }  
 },{  
 test: /\.css$/,  
 loaders: ['style', 'css']  
 },  
 { test: /\.(png|woff|woff2|eot|ttf|svg)$/, loader: 'url-loader?limit=100000' }  
]

1. Update your component appropriately:

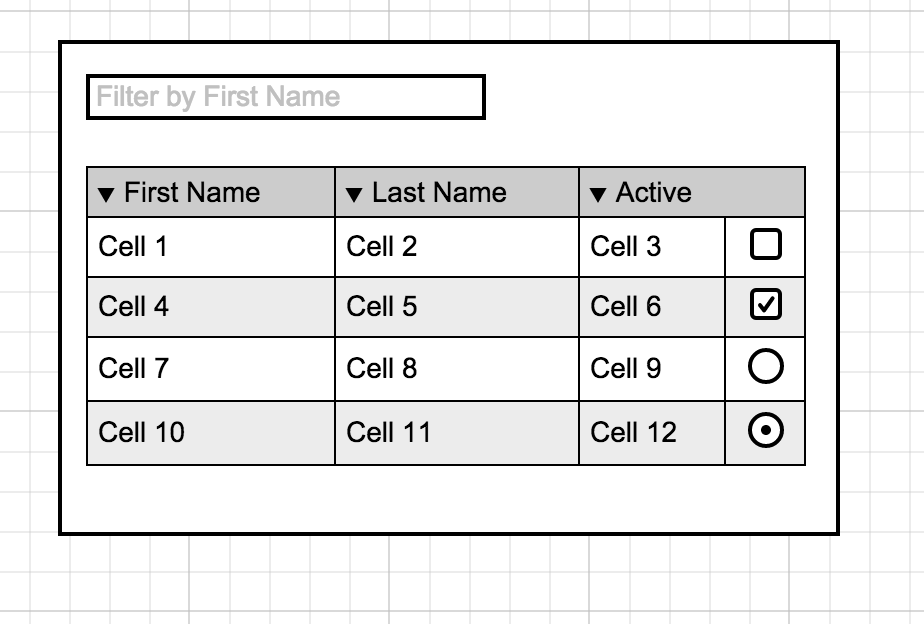
require("bootstrap/dist/css/bootstrap.css");  
**import** React **from** 'react';  
**import** {render} **from** 'react-dom';  
  
**class** GridComponent **extends** React.Component {  
 render(){  
 **return** (  
 <table className="table table-condensed">  
 <thead>  
 <tr>  
 <th>Firstname</th>  
 <th>Lastname</th>  
 <th>Email</th>  
 </tr>  
 </thead>  
 <tbody>  
 <tr>  
 <td>John</td>  
 <td>Doe</td>  
 <td>john@example.com</td>  
 </tr>  
 <tr>  
 <td>Mary</td>  
 <td>Moe</td>  
 <td>mary@example.com</td>  
 </tr>  
 <tr>  
 <td>July</td>  
 <td>Dooley</td>  
 <td>july@example.com</td>  
 </tr>  
 </tbody>  
 </table>  
 )  
 }  
}  
  
render(  
 <GridComponent/>,  
 document.getElementById('app')  
);

1. Open browser:

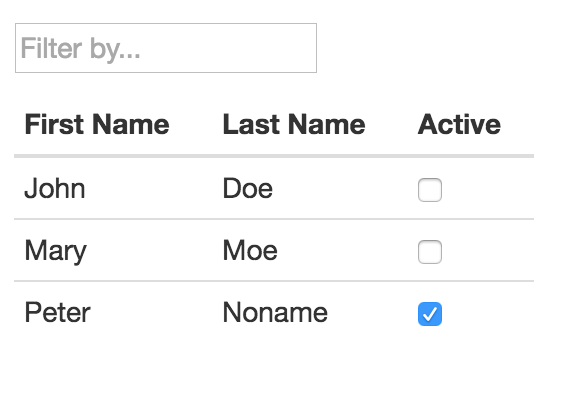


1. Very good! Now we have first UI component written in JSX, styles from bootstrap are loading with webpack and it all works together like a charm.

# Task 2: Make markup with JSX for given design:



1. I assume we all now see something like this in the browser now:



State VS Props

1. Now let’s add some state and props and remove those hardcoded table values. First of all lets create object which will emulate our data source. Add this to your app.js:

**const** dataSource = [  
 {firstName: "John", lastName: "Doe", active: **false**},  
 {firstName: "Mary", lastName: "Moe", active: **false**},  
 {firstName: "Peter", lastName: "Noname", active: **true**}  
 ]

1. Lets pass this data as a state to our GridComponent:

constructor(){  
 **super**();  
 **this**.state = {  
 records:[]  
 }  
}  
componentDidMount(){  
 **this**.setState({  
 records:dataSource  
 })  
}

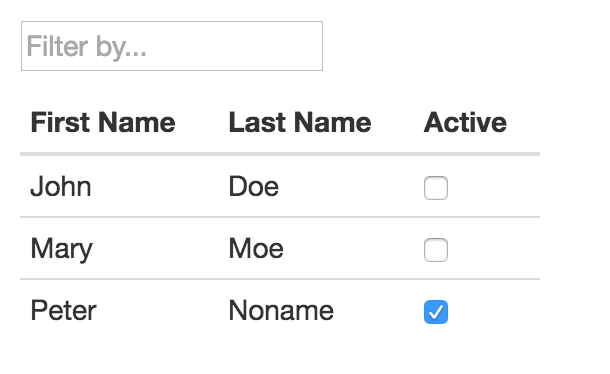
1. Awesome it’s there. Now lets build our component based on this props it receives. For that first of all lets extract Grid Record from the component and consider it also going to get data it needs as a properties:

**class** GridRecord **extends** React.Component {  
 render(){  
 **let** {record} = **this**.props;  
 **return** <tr>  
 <th>{record.firstName}</th>  
 <th>{record.lastName}</th>  
 <th><input type="checkbox" checked={record.active}/></th>  
 </tr>  
 }  
}

1. Now update GridComponent render method to render GridRecords instead of hardcoded markup:

render(){  
 **let** records = **this**.state.records.map((record)=>{  
 **return** <GridRecord record={record}/>  
 });  
 return (  
 <div style={{width:300, height: 300, padding: 20}}>  
 <p>  
 <input type="text" placeholder="Filter by..."/>  
 </p>  
 <table className="table table-condensed">  
 <thead>  
 <tr>  
 <th>First Name</th>  
 <th>Last Name</th>  
 <th>Active</th>  
 </tr>  
 </thead>  
 <tbody>  
 {records}  
 </tbody>  
 </table>  
 </div>  
 )  
 }  
}

1. Okay, make sure you understand what happened here. We passed data to GridComponent, after that iterated records from data and passed each of record as a property to GridRecord component, after that we just simply added all records to records array which we rendered in GridComponent. Open it in the browser and make sure that nothing changed so far:



1. But what actually happened is that we separated responsibilities and improved scalability of our component. And you can see one direction Data Flow here. You can also do it like that:

**return** (  
 <div style={{width:300, height: 300, padding: 20}}>  
 <p>  
 <input type="text" placeholder="Filter by..."/>  
 </p>  
 <table className="table table-condensed">  
 <thead>  
 <tr>  
 <th>First Name</th>  
 <th>Last Name</th>  
 <th>Active</th>  
 </tr>  
 </thead>  
 <tbody>  
 {**this**.state.records.map((record)=>{  
 **return** <GridRecord record={record}/>  
 })}  
 </tbody>  
 </table>  
 )

1. Now you might see in console something like that:

../../Desktop/Screen%20Shot%202016-04-14%20at%202.27.06%20PM.p

1. This is happening because for internal optimization in react if you create components dynamically with iterators you need to provide each of the component some unique identifier:

{**this**.state.records.map((record, index)=>{  
 **return** <GridRecord record={record} key={index}/>  
})}

1. Now we need to give user availability to change by clicking on checkboxes. Hypothetically, first thing which comes to mind is something like that:

toggleActive(){  
 **this**.props.record.active = !**this**.props.record.active;  
 **this**.replaceProps({  
 record: **this**.props.record  
 })  
}

1. This solution is not correct, because properties should only come from parent component and state is only what can be changed inside the component. Changing properties inside the component is considered as a bad practice and replaceProps method is already deprecated in react. Remove wrong solution and to make it right we need to follow principle Data goes down, Actions go up. Pass Action Handler to the GridRecord Component:

<GridRecord record={record} key={index} toggleActive={**this**.toggleActive.bind(**this**, index)}/>

1. If you noticed we used bind method here. I assume you know what bind does and will explain why are we using it in React. With help of bind we create Partial Application effect to make sure we pass reference to the function under GridComponent context which will receive index as a first argument.
2. Create action handler method in your GridComponent:

toggleActive(index){  
 **let** {records} = **this**.state;  
 records[index].active = !records[index].active;  
 **this**.setState({  
 records:records  
 })  
}

1. And activate it at GridRecord:

<th><input type="checkbox" checked={record.active} onChange={**this**.props.toggleActive}/></th>

1. To summarize:
   1. **GridComponent** is responsible for data. It know how to change it and since we are working with state – it know how to update dom after data is changed
   2. **GridRecord** knows how to create markup based on incoming props. It know what it should call when you click on it. But if at some day you decide to change toggleActive method he doesn’t need to be aware of it.

# Task 3: Make editable lastname in the grid.

## Acceptance criteria:

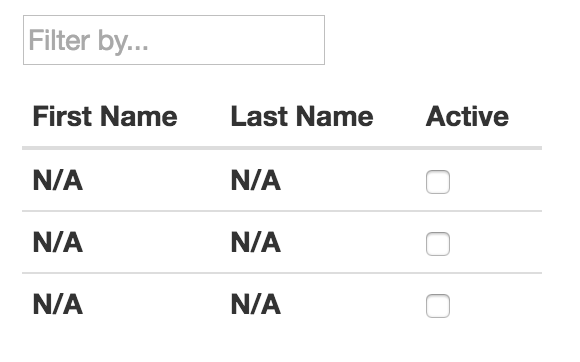
1. Column can be editable
2. After editing and moving cursor to any other columns or field value remains as edited.
3. Lets make our components more reusable. First we will add defaultProps in case something goes wrong and our GridRecord component will get not really what he expects:

GridRecord.defaultProps = {  
 record: {firstName: "N/A", lastName: "N/A", active: **false**}  
};

1. Let now check it:

**return** <GridRecord record={undefined} key={index} toggleActive={**this**.toggleActive.bind(**this**, index)}/>

1. So now instead of errors in console and empty grid you will see:



1. Cool, second thing is propTypes:

GridRecord.propTypes = {  
 record: React.PropTypes.shape({  
 firstName: React.PropTypes.string.isRequired,  
 lastName: React.PropTypes.string.isRequired,  
 active:React.PropTypes.bool.isRequired  
 })  
};

1. Let’s try to modify our dataSource, we will update firstName to 123:

**const** dataSource = [  
 {firstName: 123, lastName: "Doe", active: **false**},  
 {firstName: "Mary", lastName: "Moe", active: **false**},  
 {firstName: "Peter", lastName: "Noname", active: **true**}  
];

1. Open browser console:

../../Desktop/Screen%20Shot%202016-04-16%20at%2012.20.44%20PM.p

1. You can remove replace 123 in dataSource back to John.
2. It’s very useful thing. Okay, lets now finish with our filter. We need to add new listener to input field in GridComponent class:

onChange={**this**.handleFilterChange.bind(**this**)}

1. And method itself in GridComponent:

handleFilterChange(e){  
 **let** value = e.target.value,  
 records = dataSource.filter((record) => record.firstName.toUpperCase().includes(value.toUpperCase()));  
 **this**.setState({  
 records:records  
 });  
}

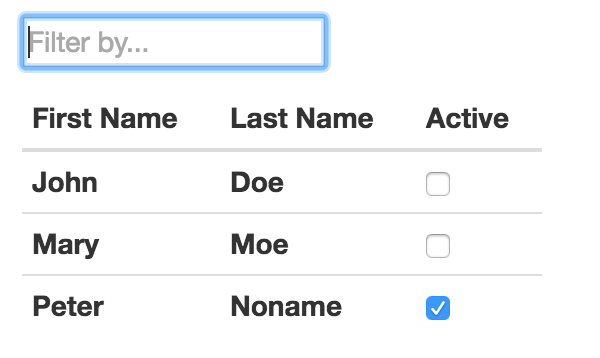
1. Very good, but lets say that according to new requirements we need to let user type in this field right after page is loaded. We need to focus it somehow. How can we do it? We need ref:

<input type="text" ref="filterInput" placeholder="Filter by..." onChange={**this**.handleFilterChange.bind(**this**)}/>

and

componentDidMount(){  
 **this**.refs.filterInput **&& this**.refs.filterInput.focus();  
 **this**.setState({  
 records:dataSource  
 })  
}

That’s it:



React Children

1. Ok, it works and we are on the half of the way. Now lets imagine you need to develop numerous grids like this one and you need to provide different rows at the bottom of the grid, with different kinds of information – show how many active users and how many users at all. What will you do? I would say it makes sense to use children from props:

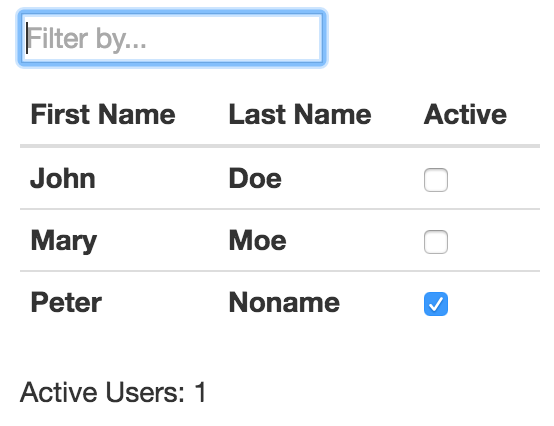
**class** SummaryActive **extends** React.Component {  
 render(){  
 **return** (  
 <div>Active Users: {**this**.props.records.filter((record)=>record.active).length}</div>  
 )  
 }  
}  
  
**class** SummaryUsers **extends** React.Component {  
 render(){  
 **return** (  
 <div>Users Count: {**this**.props.records.length}</div>  
 )  
 }  
}

render(  
 <GridComponent>  
 <SummaryActive/>  
 </GridComponent>,  
 document.getElementById('app')  
);

And add it as a child to GridComponent:

</table>  
 <div>{**this**.props.children &&

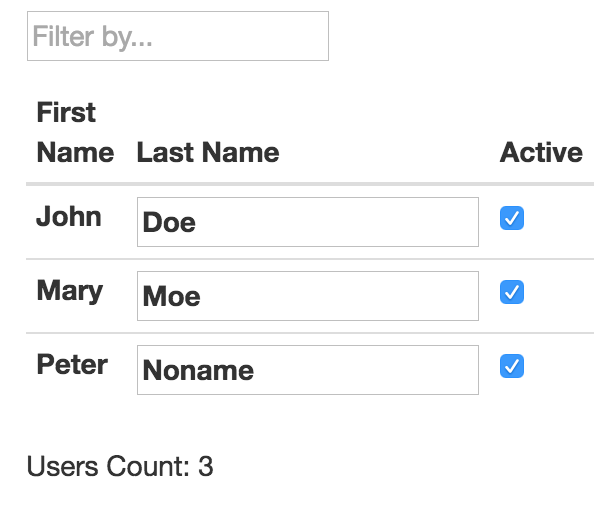
React.cloneElement(**this**.props.children, { records: **this**.state.records })}</div>  
 </div>  
)



1. And lets say for some other grid we will need to know how many users we have. You do it simply like that:

render(  
 <GridComponent>  
 <SummaryUsers/>  
 </GridComponent>,  
 document.getElementById('app')  
);

1. No more changes required in GridComponent. It’s very convenient, because now GridComponent is completely untied from bottom rows.



1. Lets now make our app.js more clear. First we will remove SummaryActive and SummaryUsers from it and put it in some separate file. Let’s create file summaries.js and put it there
2. Now lets create grid.js and put GridComponent and GridRecord and dataSource in this file.
3. If you will try to load app now you will see:



1. It’s expected, don’t worry. We removed parts of our app to separate files, but we didn’t manage to connect them. Lets do it now. We will add React to all files we created:

**import** React **from** 'react';

1. We will add export to components we are planning to export:

**export class** SummaryActive **extends** React.Component {

**export class** SummaryUsers **extends** React.Component {

**export default class** GridComponent **extends** React.Component {

1. We will add import in app.js where we going to use GridComponent and Summaries

**import** GridComponent **from** './grid';  
**import** {SummaryActive, SummaryUsers} **from** './summaries';

1. After all those changes your app.js will look like that:

require("bootstrap/dist/css/bootstrap.css");  
**import** React **from** 'react';  
**import** {render} **from** 'react-dom';  
**import** GridComponent **from** './grid';  
**import** {SummaryActive, SummaryUsers} **from** './summaries';  
  
  
render(  
 <GridComponent>  
 <SummaryUsers/>  
 </GridComponent>,  
 document.getElementById('app')  
);

1. Very clean and structured app.

# Task 4: Create UserDetailsComponent which will render give html

1. Create file user-details.js under js folder
2. Create react component to render this html

<div class="container">  
 <div class="row">  
 <div class="col-md-offset-2 col-md-8 col-lg-offset-3 col-lg-6">  
 <div class="well profile">  
 <div class="col-sm-12">  
 <div class="col-xs-12 col-sm-8">  
 <h2>Nicole Pearson</h2>  
 <p><strong>About: </strong> Web Designer / UI. </p>  
 <p><strong>Hobbies: </strong> Read, out with friends, listen to music, draw and learn new things. </p>  
 <p><strong>Skills: </strong>  
 <span class="tags">html5</span>  
 <span class="tags">css3</span>  
 <span class="tags">jquery</span>  
 <span class="tags">bootstrap3</span>  
 </p>  
 </div>  
 <div class="col-xs-12 col-sm-4 text-center">  
 <figure>  
 <img src="http://www.bitrebels.com/wp-content/uploads/2011/02/Original-Facebook-Geek-Profile-Avatar-2.jpg" alt="" class="img-circle img-responsive"/>  
 </figure>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
</div>

1. Add css file (you can find in additional files for this task to style this component)
2. And render this newly created component instead of GridComponent

React Router

1. Now when you have users grid and user details components we need to make some sort of router to let user switch between them. For that install Router:

npm install --save react-router

1. And add this to your app.js:

**import** { Router, Route, Link, hashHistory} from 'react-router'

**class** App **extends** React.Component {  
 render(){  
 **return** (  
 <div>  
 <h1>Our awesome app</h1>  
 <ul role="nav">  
 <li><Link to="/grid">Grid</Link></li>  
 <li><Link to="/details">Details</Link></li>  
 </ul>  
 {**this**.props.children}  
 </div>  
 )  
 }  
}  
  
render(  
 <Router history={hashHistory}>  
 <Route path="/" component={App}>  
 <Route path="grid" component={GridComponent}/>  
 <Route path="details" component={UserDetail}/>  
 </Route>  
 </Router>,  
 document.getElementById('app')  
);

1. At this moment Details page contains just a mocked markup we put there in JSX. We will fix it later
2. When we have both routes working, let make things more complicated. Let’s pass parameter from grid to user-details. Lets say we want to see details for specific user. First we will need to make a handler:

**class** GridRecord **extends** React.Component {  
 showUserDetails(e){  
 e.preventDefault();  
 hashHistory.push(`/details/${**this**.props.record.id}`);  
 }

render(){  
 **let** {record} = **this**.props;  
 **return** <tr>  
 <th onClick={**this**.showUserDetails.bind(**this**)}><a href="#">{record.id}</a></th>

<th>{record.firstName}</th>  
 <th>{record.lastName}</th>  
 <th><input type="checkbox" checked={record.active} onChange={**this**.props.toggleActive}/></th>  
 </tr>  
 }  
}

1. Add import to grid.js

**import** {hashHistory} **from** 'react-router'

1. Then we will update grid table to show id:

<thead>  
<tr>  
 <th>Id</th>  
 <th>First Name</th>  
 <th>Last Name</th>  
 <th>Active</th>  
</tr>  
</thead>

1. Lets add new route to our routes:

render(  
 <Router history={hashHistory}>  
 <Route path="/" component={App}>  
 <Route path="grid" component={GridComponent}/>  
 <Route path="details" component={UserDetail}>  
 <Route path="/details/:id" component={UserDetail}/>  
 </Route>  
 </Route>  
 </Router>,  
 document.getElementById('app')  
);

1. And finally update dataSource:

**const** dataSource = [  
 {firstName: "John", lastName: "Doe", active: **false**, id: 1},  
 {firstName: "Mary", lastName: "Moe", active: **false**, id: 2},  
 {firstName: "Peter", lastName: "Noname", active: **true**, id: 3}  
];

1. Now we can get id at grid component as easy as (Add it to your UserDetails):

<h1>THIS IS PARAM FROM GRIDCOMPONENT: {**this**.props.params.id}</h1>

# Task 5: Create dataSource for user-details and based on id from grid show correct user-details.

We have developed GridComponent with mocked data for now and when it loads it shows data from the mock. As of now user-details shows hardcoded html in jsx. We need to make it more dynamic for that replace hardcoded html with data from the detailsRecords object (described below) and connect it with GridComponent parameter.

## Acceptance criteria:

1. User-details.js component shows data base on mocked data object.
2. User-details component can show as many details as many objects specified in source data
3. Source data should be accepted as this:

detailsRecords = [{  
 id:1,  
 name:"John Doe",  
 about:"Nice guy",  
 hobby:"Likes drinking wine",  
 skills:["html", "javascript", "redux"]  
},{  
 id:2,  
 name:"Mary Moe",  
 about:"Cute girl",  
 hobby:"Likes playing xbox whole days long",  
 skills:["Fortran", "Lua", "R#"]  
}];

1. If user-details retrieves id from gridComponent it need to be able to show only specific data from data source

Redux.

1. As of now we have built nice app which actually does very common things at UI. But we will try to switch from vanilla React to React + Redux to show you how you easy it is.
2. First thing you need to do is: “npm install --save redux react-redux”
3. Create several folders: Actions, Reducer, Store, Constants, Components, Containers. You can do it with

mkdir Actions && mkdir Reducer && mkdir Store && mkdir Constants && mkdir Components && mkdir Containers

1. Move grid.js and user-details.js and summaries.js from js folder to components and app.js move to containers
2. Update imports in app.js for UserDetail, GridComponent and Summaries to import them from the components folder
3. Create index.js in reducer, actions, store, constants folders
4. Ok, we are ready to start. Now first of all update your webpack.config:

**const** PATHS = {  
 app: "./containers/app",  
 build: path.join(\_\_dirname, 'build')  
};

1. Update app.js:

**import** { Provider } **from** 'react-redux'

**import** configureStore **from** '../store/index'

**const** store = configureStore();

render(  
 <Provider store={store}>  
 <Router history={hashHistory}>  
 <Route path="/" component={App}>  
 <Route path="grid" component={GridComponent}/>  
 <Route path="details" component={UserDetail}>  
 <Route path="/details/:id" component={UserDetail}/>  
 </Route>  
 </Route>  
 </Router>  
 </Provider>,  
 document.getElementById('app')  
);

1. Let me explain what we did so far:

We have removed js/app.js and added containers/app.js. To make webpack build work again we needed to fix entry point. This is first.

Redux app needs to be wrapped in Redux Provider, that’s what we did – wrapped our app, to let redux manage it as a child

We called configure store (Not yet created, but we will do it in a few moments)

Applied our store (Don’t forget that we will have single instance single store) to Provider

1. Now lets figure out what we gonna do with store:

Goto: store/index.js and put it there:

**import** { createStore,applyMiddleware } **from** 'redux'  
**import** {rootReducer} **from** '../reducer'  
  
**export default function** configureStore(initialState) {  
 **return** createStore(rootReducer);  
}

Nothing about store here, right? It’s correct. We don’t need to create any objects or structures here; we just need to point redux’s createStore function to our rootReducer. Everything else redux will do.

1. Ok, now let’s go to reducers, the place where all magic happens:

First of all we need to create separate reducers for grid and details. For that put this in reducer/index.js:

**export function** grid(state = gridRecords, action){  
 **switch** (action.type) {  
 **default**:  
 **return** state  
 }  
}  
  
**export function** details(state = detailsRecords, action){  
 **switch** (action.type) {  
 **default**:  
 **return** state  
 }  
}

Do you see this state = gridRecords and state = detailsRecords? It’s ES6 syntax for value in case it’s not provided in parameters. Why do we need it? We need it to set value for a first call to reducer. When you just start you app redux will need to fill store with some values and since store at this moment is undefined we set default values directly in reducers. Lets define them:

**let** gridRecords = [  
 {firstName: "John", lastName: "Doe", active: **false**, id: 1},  
 {firstName: "Mary", lastName: "Moe", active: **false**, id: 2},  
 {firstName: "Peter", lastName: "Noname", active: **true**, id: 3}  
],  
 detailsRecords = [{  
 id:1,  
 name:"John Doe",  
 about:"Nice guy",  
 hobby:"Likes drinking wine",  
 skills:["html", "javascript", "redux"]  
 },{  
 id:2,  
 name:"Mary Moe",  
 about:"Cute girl",  
 hobby:"Likes playing xbox whole days long",  
 skills:["Fortran", "Lua", "R#"]  
 }];

1. Okay we have reducer for each of screen. Now lets merge them in one reducer:

**import** { combineReducers } **from** 'redux'

**export const** rootReducer = combineReducers({  
 details,  
 grid  
});

You must be curious why do we do it like that? The answer is simple: since we are having only one store, the easiest way would be to modify data in this store with one reducer. But just try to imagine the size and structure of this reducer for a huge apps. You won’t be able to support it and understand. That’s why reducers are the thing which separates logical parts changes for store. As many logical parts you have in your app – as many reducers you gonna have. And just simply.

You can split reducers as many times as you want.

Reducers reminds Responsibility Chain pattern meaning reducers will be called one by one to change state. You can even do something like that:

**export function** grid(state = gridRecords, action){  
 **switch** (action.type) {  
 **case** "FILTER":  
//I also do something on filter action  
 **return** state;  
 **default**:  
 **return** state  
 }  
}  
  
**export function** details(state = detailsRecords, action){  
 **switch** (action.type) {  
 **case** "FILTER":  
//I also do something on filter action  
 **return** state;  
 **default**:  
 **return** state  
 }  
}

1. Ok, lets move on for now. But we will come back to them. Let summarize what we did so far:
   1. We created reducer which merges two other reducer and just simply provide default state as of now
   2. We created store with this reducer
   3. We wrapped our existing app in Redux wrapper container
2. Now we can provide data to the components. Open grid.js and do the following:
   1. Remove hardcoded data
   2. Remove export from the GridComponent
   3. Add this:

**import** { connect } **from** 'react-redux'

GridComponent.propTypes = {  
 records: React.PropTypes.array.isRequired  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 records: state.grid  
 }  
}  
  
**export default** connect(  
 mapStateToProps  
)(GridComponent)

1. What we just did? We have provided our GridComponent with data from our store. Since now data comes as a property we need to replace state with props everywhere in GridComponent:

{**this**.props.records.map((record, index)=>{

1. And we need to remove this and dataSource object:

**this**.setState({  
 records:dataSource  
})

1. If you will refresh the page now you supposed to see the same component, but it’s now driven by React + Redux
2. But for now we pass just a half of the way. We can read from store, but how can we update the store? Lets do it. First of all we need to modify toggleActive method:

toggleActive(index){  
 **let** {dispatch} = **this**.props;  
 dispatch({  
 type:"TOGGLE\_ACTIVE",  
 value:index  
 });  
}

1. After that lets update reducer:

**export function** grid(state = gridRecords, action){  
 **switch** (action.type) {  
 **case** "TOGGLE\_ACTIVE":  
 **let** newState = [...state];  
 newState[action.value].active = !newState[action.value].active;  
 **return** newState;  
 **case** "FILTER":  
 //Filter will be implemented later  
 **default**:  
 **return** state  
 }  
}

1. That’s it reload the page and you will see data is updated.

# Task 6: Make filter by textfield.

## Acceptance criteria:

1. With use of dispatch, action, reducer filter records in grid according to firstname matching data in textfield
2. Now lets figure out what to do with details. First we remove all hardcoded values from components. Then we map data from store to the properties and pass them to component:

**import** { connect } **from** 'react-redux'  
  
**import** UserDetail **from** './user-detail';

**class** UserDetails **extends** React.Component {  
 render(){  
 **return** (  
 <div>  
 <h1>THIS IS PARAM FROM GRIDCOMPONENT: {**this**.props.params.id}</h1>  
 {**this**.props.details.map((detail, i)=>{  
 **return** <UserDetail key={i} detail={detail}/>  
 })}  
 </div>  
 )  
 }  
}  
  
UserDetails.propTypes = {  
 details: React.PropTypes.array.isRequired  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 details: state.details  
 }  
}  
  
**export default** connect(  
 mapStateToProps  
)(UserDetails)

and we create user-details.js and put component template in it:

**export default class** UserDetail **extends** React.Component {  
 render(){  
 **let** {detail} = **this**.props;  
 **return** (  
 <div className="container">  
 <div className="row">  
  
 <div className="col-md-offset-2 col-md-8 col-lg-offset-3 col-lg-6">  
 <div className="well profile">  
 <div className="col-sm-12">  
 <div className="col-xs-12 col-sm-8">  
 <h2>{detail.name}</h2>  
 <p><strong>About: </strong> {detail.about} </p>  
 <p><strong>Hobbies: </strong> {detail.hobbies} </p>  
 <p><strong>Skills: </strong>  
 {detail.skills.map((skill, i)=>{  
 **return** <span key={i} className="tags">{skill}</span>  
 })}  
 </p>  
 </div>  
 <div className="col-xs-12 col-sm-4 text-center">  
 <figure>  
 <img src="http://www.bitrebels.com/wp-content/uploads/2011/02/Original-Facebook-Geek-Profile-Avatar-2.jpg" alt="" className="img-circle img-responsive"/>  
 </figure>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 )  
 }  
}

That’s it, it works.

# Task 7: If user clicks on grid – show only appropriate user details.

## Acceptance criteria:

1. Property that comes to user-details able to sort source records
2. If in source records there’re 3 records with ids – 1,2,3 and property comes as 2 then only record with id equals 2 should be placed at details page
3. Okay, that’s good. But lets say you will need to rename FILTER\_DETAILs to SEARCH\_DETAILS. What you will need to do is change it in reducer and component? It’s not very convenient. WE can replace action with action creators. But first lets define them in one place. Go to constants index.js and put it there:

//DETAILS PAGE ACTIONS  
**export const** FILTER\_DETAILS = 'FILTER\_DETAILS';  
  
//GRID ACTIONS  
**export const** TOGGLE\_ACTIVE = 'TOGGLE\_ACTIVE';  
**export const** FILTER = 'FILTER';

1. Now we have actions constants defined and we can create action creators. Go to actions index.js and put it there:

**import** \* **as** types **from** '../constants'  
  
**export function** filterDetails(value) {  
 **return** {  
 type: types.FILTER\_DETAILS,  
 value  
 }  
}  
  
**export function** filterGrid(value) {  
 **return** {  
 type: types.FILTER,  
 value  
 }  
}  
  
**export function** toggleActive(value) {  
 **return** {  
 type: types.TOGGLE\_ACTIVE,  
 value  
 }  
}

1. We need to update reducer after that:

**import** {TOGGLE\_ACTIVE, FILTER, FILTER\_DETAILS} **from** '../constants'

**case** TOGGLE\_ACTIVE:

**case** FILTER:

**case** FILTER\_DETAILS:

1. We just defined all actions as constants in one place and action creators which depend on value and action are also separate. Lets finally update dispatch:

Grid.js

**import** {filterGrid, toggleActive} **from** '../actions'

toggleActive(index){  
 **let** {dispatch} = **this**.props;  
 dispatch(toggleActive(index));  
}  
handleFilterChange(e){  
 **let** {dispatch} = **this**.props;  
 dispatch(filterGrid(e.target.value));  
}

user-details.js

**import** {filterDetails} **from** '../actions'

componentDidMount(){  
 **let** {dispatch} = **this**.props;  
 dispatch(filterDetails(**this**.props.params.id));  
}

1. Awesome, code is clear and data flow is predictable. Let’s now imagine that we need to load data in grid from server not from hardcoded variable. Please find server folder at additional training resources and hit:

node server.js

1. Now, we have dummy server which does only one thing on any request it generated json data with gridRecords and detailsRecords in it. Lets figure out how can we connect it to our application.
2. Okay now we need to fire action creator for it. But how can we do it? Let start first create actions we will need. Put it in your constants:

**export const** START\_LOADING = 'START\_LOADING';  
**export const** STOP\_LOADING = 'STOP\_LOADING';  
**export const** ADD\_DATA= 'ADD\_DATA';

1. Now we need to define action creators:

**export function** startLoading() {  
 **return** {  
 type: types.START\_LOADING  
 }  
}  
  
**export function** stopLoading() {  
 **return** {  
 type: types.STOP\_LOADING  
 }  
}  
  
**export function** addData(value) {  
 **return** {  
 type: types.ADD\_DATA,  
 value  
 }  
}

1. Since we are emulating loading process in the grid, we need to update grid state shape with new field:

**let** gridState = {  
 records:[],  
 filtered: [],  
 loading:**false** }

**export function** grid(state = gridState, action){

1. Let’s add couple new reducers and constants for them:

**import** {TOGGLE\_ACTIVE, FILTER, FILTER\_DETAILS, START\_LOADING, STOP\_LOADING, ADD\_DATA} **from** '../constants'

**case** START\_LOADING:  
 **return** Object.assign({}, state, {loading: **true**});  
**case** STOP\_LOADING:  
 **return** Object.assign({}, state, {loading: **false**});  
**case** ADD\_DATA:  
 **return** Object.assign({}, state, {  
 records:[...action.value]  
 });

1. Since we updated grid state shape we need to update connect and mapStateToProps:

GridComponent.propTypes = {  
 records: React.PropTypes.array.isRequired,  
 filtered: React.PropTypes.array.isRequired,  
 loading: React.PropTypes.bool.isRequired  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 records: state.grid.records,  
 filtered: state.grid.filtered,  
 loading: state.grid.loading  
 }  
}

1. And finally we get to the dispatching. Here’s how you can do it:

componentDidMount(){  
 **this**.refs.filterInput && **this**.refs.filterInput.focus();  
 **this**.loadData();  
}  
loadData(){  
 **let** {dispatch} = **this**.props;  
 dispatch(startLoading());  
 fetch('http://localhost:4730')  
 .then(**function**(response) {  
 **return** response.json();  
 }).then(**function**(json) {  
 dispatch(addData(json.gridRecords))  
 }).then(**function**(){  
 dispatch(stopLoading());  
 })  
}

1. If you refresh the page now you supposed to see data in the grid which comes from server.
2. That’s the easiest way to handle async actions or I would say async sequence of actions in redux. But as for me that’s not very convenient way. I look at this loadData method and see 3 dispatchers but generally only one action – LOAD\_DATA. But wait a second… We already have action creators who create action for us, so let them do it also. But here’s the problem: action creators have no idea about dispatcher. Redux middleware can help us here
3. Redux Thunk middleware allows you to write action creators that return a function instead of an action. In this case we are returning function from loadDataInGrid() which allows to do asynchronous processing
4. To use any kind of redux middleware (thunk, saga, logs) we need to initialize our app with it and install it, because thunk or saga or logs are separate modules. We will use thunk:

npm install --save redux-thunk

1. Go to your store index.js and replace your code with this one:

**import** { createStore,applyMiddleware } **from** 'redux'  
**import** thunk **from** 'redux-thunk';  
**import** {rootReducer} **from** '../reducer'  
  
**export default function** configureStore(initialState) {  
 **const** createStoreWithMiddleware = applyMiddleware(  
 thunk  
 )(createStore);  
  
 **const** store = createStoreWithMiddleware(rootReducer);  
 **return** store;  
}

1. What we do here is we are basically saying redux to create store with dispatch method being able go to any middlewares you provide in it before it actually reaches the reducer. Exactly what we need. Ok, lets define action creator for it:

**export function** loadDataInGrid(){  
 **return** (dispatch)=>{  
 dispatch(startLoading());  
 fetch('http://localhost:4730')  
 .then(**function**(response) {  
 **return** response.json();  
 }).then(**function**(json) {  
 dispatch(addData(json.gridRecords))  
 }).then(**function**(){  
 dispatch(stopLoading());  
 })  
 }  
}

1. Now we can remove startLoading, stopLoading, addData action creators from grid.js and add only one:

**import** {filterGrid, toggleActive, loadDataInGrid} **from** '../actions'

1. loadData method will look also quite more accurate:

loadData(){  
 **let** {dispatch} = **this**.props;  
 dispatch(loadDataInGrid());  
}

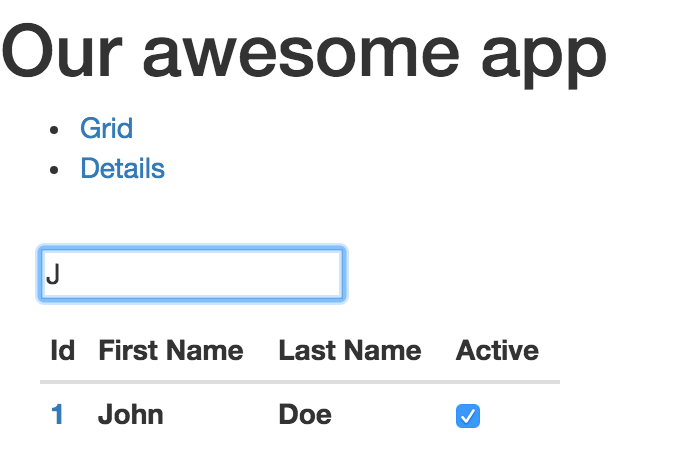
1. If you reload your page now it should work.

# Task 8: Need to fix TOGGLE\_ACTIVE and FILTER reducers for grid

While we were making changes and integrating Action Creators, Server and reducers we completely forgot about Toggle Active in the grid and Filter.

## Acceptance Criteria:

1. Toggle active and filter should work again, after refactoring with Action creators, reducers



# Task 9: Need to add loading state to GridComponent

## Acceptance criteria:

1. Loading actions are dispatched
2. Loading changes from loading: false to loading: true
3. When loading equals true GridComponent renders:

<div style={{width:300, height: 300, padding: 20}}>Loading...</div>

instead of existing markup

1. When loading equals false GridComponent renders current GridComponent markup again

# Task 10: Using redux thunk need to load and filter data in the user details

## Acceptance criteria:

1. When user details has opened from the general link – it needs to show all data which comes from server
2. When user details has opened from the grid – it needs to get last data from server and filter it by id from the grid

Testing React.js

1. We are done with our coding, now lets write some tests. First of all we need to create to install everything we gonna use:

npm install --save-dev mocha babel-register react-addons-test-utils jsdom enzyme chai

1. Lets create folder test with one file in it: index.spec.js
2. We need to create file .babelrc in root of our application
3. Place this in .babelrc:

{  
 "presets": ["es2015", "react"]  
}

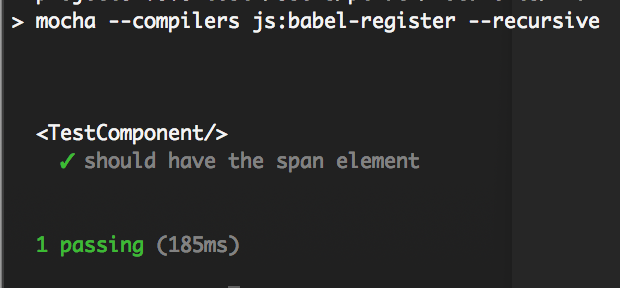
1. And finally update your package.json scripts section with new script:

"test": "mocha --compilers js:babel-register --recursive",

1. Let me finalize. This script runs mocha test runner with small remark that any file placed under test folder with \*.js extension needs to be compiled (preprocessed) with babel-register recursively (including sub dirs). And since we are using babel-register here we need to have .babelrc file to let babel know which presets he needs to use. Before we configured them in webpack.conf.js
2. Lets now create some very simple test in test/test.spec.js:

**import** React **from** 'react';  
**import** { mount, shallow } **from** 'enzyme';  
**import** {expect} **from** 'chai';  
  
**export class** TestComponent **extends** React.Component {  
 render(){  
 **return** <div><span>Test Component</span></div>  
 }  
}  
describe('<TestComponent/>', **function** () {  
 it('should have the span element', **function** () {  
 **const** wrapper = shallow(<TestComponent/>);  
 expect(wrapper.find('span')).to.have.length(1);  
 });  
});

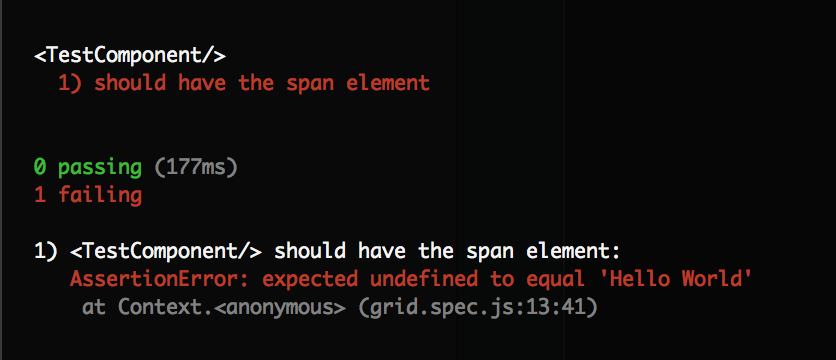
1. Now type npm test in your console and you will something like that:



1. Lets play around with it a little bit:

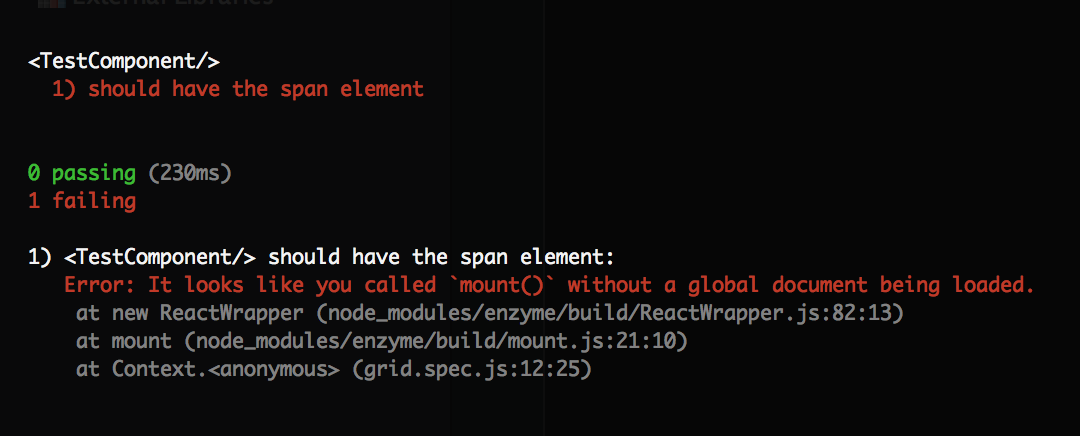
**export class** TestComponent **extends** React.Component {  
 render(){  
 **return** <div><span>{**this**.props.text}</span></div>  
 }  
}  
describe('<TestComponent/>', **function** () {  
 it('should have the span element', **function** () {  
 **const** wrapper = shallow(<TestComponent text="Hello World"/>);  
 expect(wrapper.props().text).to.equal('Hello World')  
 });  
});

run npm test again. What do you in console? I think something like this:



1. You might be curious why do we get this error? We get it because we use shallow render which actually very useful when you are testing your component as a units which are not interact with other component, DOM. But here we actually need to interact with other world, we need to get props at least. Let replace our shallow with mount method:

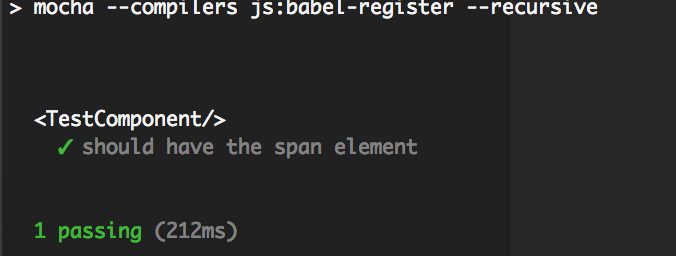
**const** wrapper = mount(<TestComponent text="Hello World"/>);  
expect(wrapper.props().text).to.*equal*('Hello World')



1. Oh no, test not pass again. Now it’s saying that we are trying to mount component, which actually means put it in DOM, pass all lifecycle methods and so on when we don’t have any document. Let’s create document for it. Create file setup.js at test folder and put this code there:

**import** { jsdom } **from** 'jsdom'  
  
global.document = jsdom('<!doctype html><html><body></body></html>');  
global.window = document.defaultView;  
global.navigator = global.window.navigator;

1. Rerun the tests:



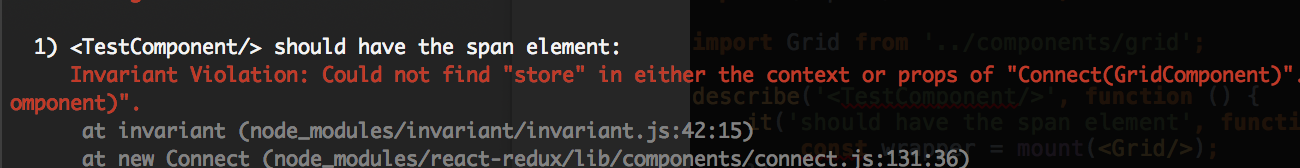
1. Cool it pass. By the way do you still type npm test every time you need to rerun your tests? Lets fix it, add this to your package.json scripts, and when your will type “npm run test:watch” mocha will rerun test every time you change your code

"test:watch": "npm test -- --watch",

1. We played with React.js testing a little bit, now lets cover with tests our application:
2. Create grid.spec.js and put it there:

**import** React **from** 'react';  
**import** { mount, shallow } **from** 'enzyme';  
**import** {expect} **from** 'chai';  
  
**import** Grid **from** '../components/grid';  
  
describe('<TestComponent/>', **function** () {  
 it('should have the span element', **function** () {  
 **const** wrapper = mount(<Grid/>);  
 });  
});

in console you will get:



1. That’s happening because of the way we defined our GridComponent. I would say it’s rather Container then Component. It uses connect function to pick properties from store, but here we are not providing any store to it and it fails. We need to wrap GridComponent in Container and make it fully independent from Redux store to let us mock it. To make it we just need to add two classes to each of the files correspondingly:

**let** GridContainer = **class extends** React.Component {  
 render(){  
 **return** <GridComponent {...**this**.props}/>  
 }  
};

**let** UserDetailsContainer = **class extends** React.Component {  
 render(){  
 **return** <UserDetails {...**this**.props}/>  
 }  
};

and replace with old one:

**export default** connect(  
 mapStateToProps  
)(GridContainer)

**export default** connect(  
 mapStateToProps  
)(UserDetailsContainer)

1. Let’s also move Containers out of the components folder:
2. Create containers/grid.js containers/user-details.js and place it there:

**import** React **from** 'react';  
**import** { connect } **from** 'react-redux'  
**import** GridComponent **from** '../components/grid'  
  
  
**let** GridContainer = **class extends** React.Component {  
 render(){  
 **return** <GridComponent {...**this**.props}/>  
 }  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 records: state.grid.records,  
 filtered: state.grid.filtered,  
 loading: state.grid.loading  
 }  
}  
  
**export default** connect(  
 mapStateToProps  
)(GridContainer)

**import** React **from** 'react';  
**import** { connect } **from** 'react-redux'  
**import** UserDetails **from** '../components/user-details'  
  
**let** UserDetailsContainer = **class extends** React.Component {  
 render(){  
 **return** <UserDetails {...**this**.props}/>  
 }  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 details: state.details  
 }  
}  
  
**export default** connect(  
 mapStateToProps  
)(UserDetailsContainer)

1. Now update your app.js:

**import** GridContainer **from** './grid';  
**import** UserDetailsContainer **from** './user-details';

render(  
 <Provider store={store}>  
 <Router history={hashHistory}>  
 <Route path="/" component={App}>  
 <Route path="grid" component={GridContainer}/>  
 <Route path="details" component={UserDetailsContainer}>  
 <Route path="/details/:id" component={UserDetailsContainer}/>  
 </Route>  
 </Route>  
 </Router>  
 </Provider>,  
 document.getElementById('app')  
);

1. Remove connect and mapStateToProps from component grid and user-details, remove import and it should works.
2. Now we can easily import GridComponent in our test:

**import** React **from** 'react';  
**import** { mount, shallow } **from** 'enzyme';  
**import** {expect} **from** 'chai';  
  
**import** GridComponent **from** '../components/grid';  
  
**function** setup(propOverrides) {  
 **const** props = Object.assign({  
 records:[  
 {firstName: "John", lastName: "Doe", active: **false**, id: 1},  
 {firstName: "Mary", lastName: "Moe", active: **false**, id: 2},  
 {firstName: "Peter", lastName: "Noname", active: **true**, id: 3}  
 ],  
 filtered: [],  
 loading:**false**,  
 dispatch: **function**(arg1, arg2){  
 }  
 }, propOverrides);  
  
 **const** Grid = mount(<GridComponent {...props} />);  
  
 **return** {  
 component:Grid,  
 rows:Grid.find('tbody').children()  
 }  
}  
  
describe('<TestComponent/>', **function** () {  
it('should render GridComponent with 3 records by default', **function** () {  
 **let** {rows} = setup();  
 expect(rows).to.have.length(3);  
});  
  
it('should render GridComponent with 2 records with filter', **function** () {  
 **let** {rows} = setup({  
 filtered: [1]  
 });  
 expect(rows).to.have.length(2);  
});  
});

# Task 11: Create test to cover loading state.

## Acceptance criteria:

1. Loading state is covered with test