WEB-014
ReactJS

Workbook
(version 1.0, 26.01.2018)

Workbook tasks

Kick off. Install everything required to start development.

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

Hello World.

1) Create index.html page with:

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

</body>
</html>
```

2) Include react and react-dom scripts to the page and create react mount dom element

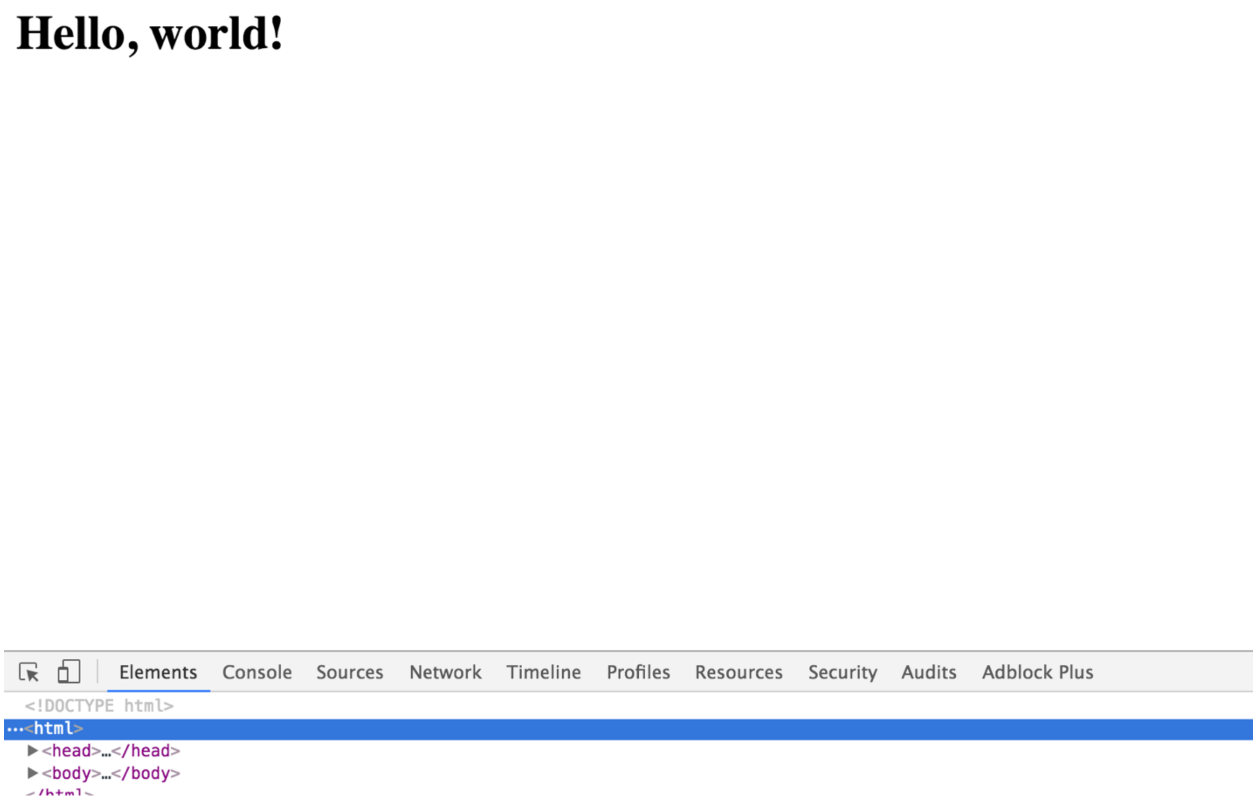
```html
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Awesome React Project</title>
    <script src="node_modules/react/umd/react.development.js"></script>
    <script src="node_modules/react-dom/umd/react-dom.development.js"></script>
    <!--<link rel="stylesheet" href="css/main.css">-->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

    </app>
<script src="js/app.js"></script>
</body>
</html>
```

3) Create your js scripts folder and add initial js file: app.js (mkdir js && cd js)

4) Add to app.js next content:

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

5) Open index.html in browser. Congrats! We just created our first React Component using pure JavaScript:


Hello, world!

```
<!DOCTYPE html>
…<html>
  ▶<head>…</head>
  ▶<body>…</body>
```

6) Okay, nice, but let 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')
);
```

7) What happened here? Does it look normal to you? Do you think it’s comfortable to read this code? – No, No, No, No!

8) Let’s move to jsx. For that we require to have Babel dependency:

```
npm install @babel/standalone --save
```

9) Then, we can rewrite app.js code with plain JSX:

```
var app = <div>
    <h1>Hi, I'm header inside div</h1>
    <ul>
        <li>
            <h2>Hi, I'm list item inside list inside div</h2>
        </li>
        <li>
            <h2>Hi, I'm list item inside list inside div</h2>
        </li>
    </ul>
</div>

ReactDOM.render(
    app,
    document.getElementById('app')
);
```

10) Since we use babel.js to render JSX components, we have to mark script type as "babel/text"

```
<script type="text/babel" src="js/app.js"></script>
```

11) Open index.html in browser. Nothing changed? Awesome! That’s exactly what we were looking for.

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:

```html
<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>
```

Create React App

We have just used plain HTML file to create React components. Babel.js can render JSX content in fly, but it is not efficient for medium, or large-scale applications.

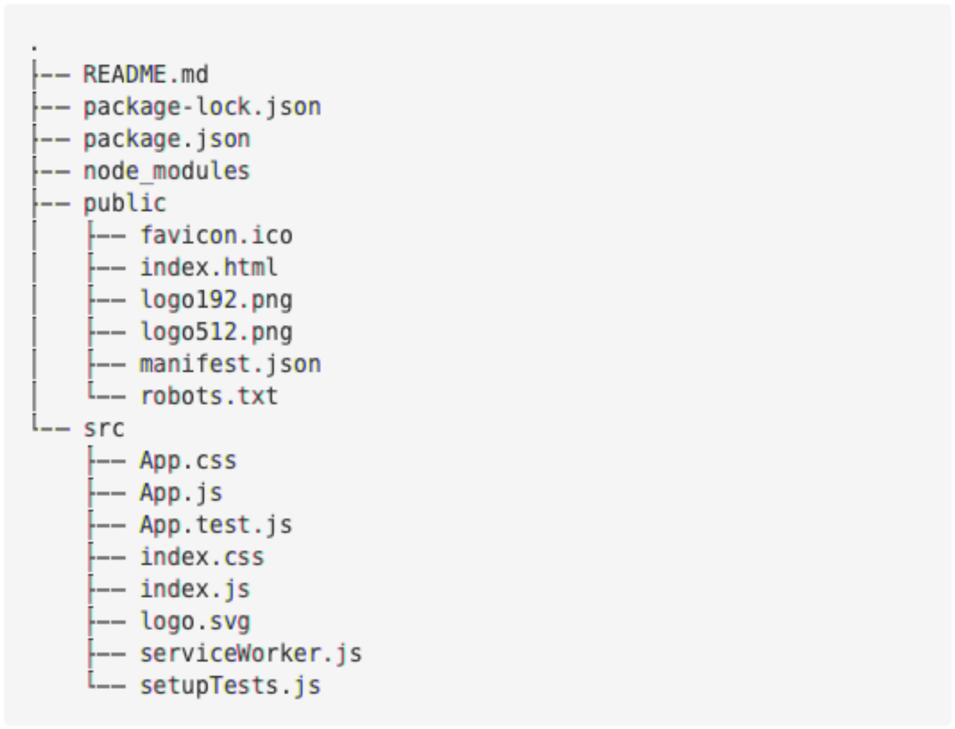
To that purpose, we can use any project starters which support transpiling JSX content to pure JS content.

Create React App is a React.js project starter project created by Facebook.

12) Let's install create-react-app, and continue with the new structure.

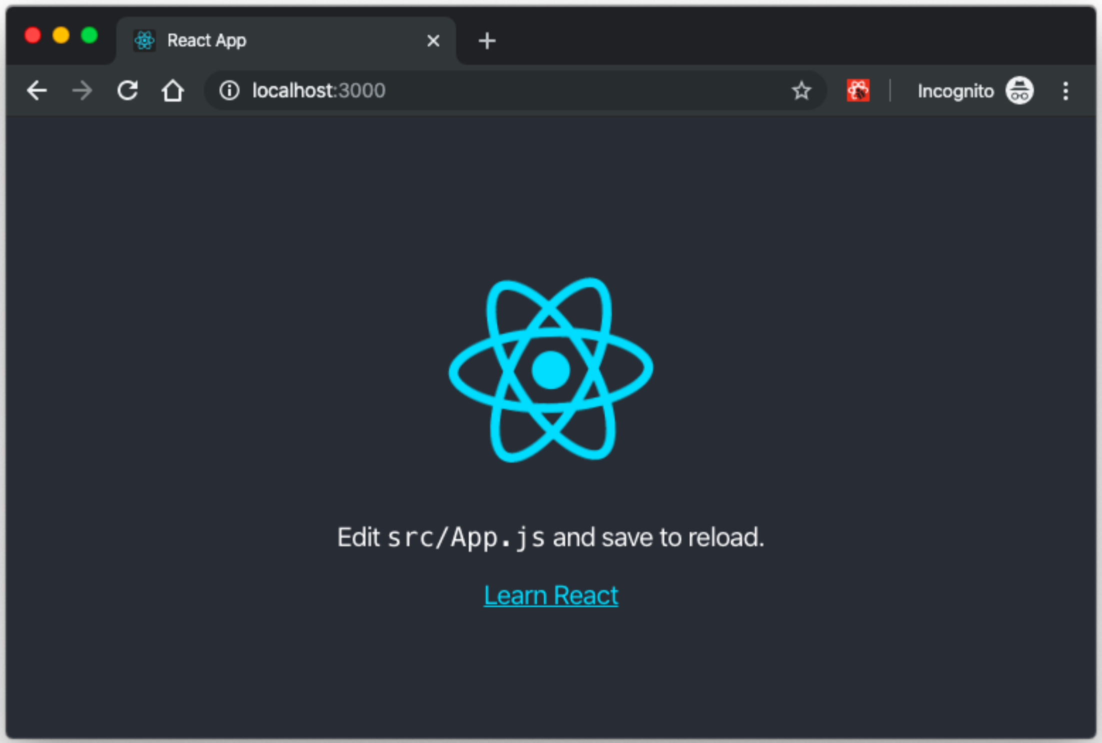
```
npx create-react-app my-project
cd my-project
```

13) You will see such project structure inside the my-project folder.



```
.
├-- README.md
├-- package-lock.json
├-- package.json
├-- node_modules
├-- public
│    ├-- favicon.ico
│    ├-- index.html
│    ├-- logo192.png
│    ├-- logo512.png
│    ├-- manifest.json
│    └-- robots.txt
└-- src
     ├-- App.css
     ├-- App.js
     ├-- App.test.js
     ├-- index.css
     ├-- index.js
     ├-- logo.svg
     ├-- serviceWorker.js
     └-- setupTests.js
```

14) To start such project, use the following command, and open localhost:3000. Initial form of the UI can be seen below:



Edit `src/App.js` and save to reload.

15) That’s great! Now investigate what is inside of the src/App.js file!

Grid Development

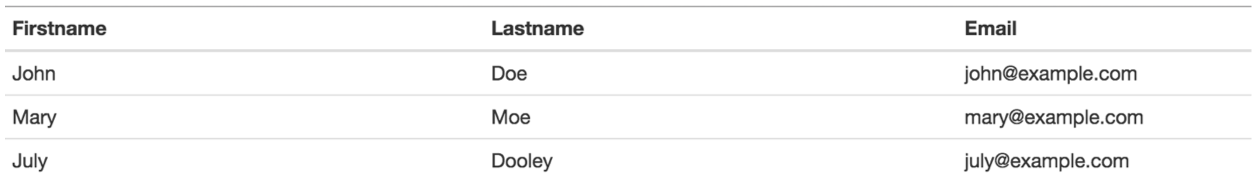
Let's create something useful. We will start from creating a Grid component using Twitter's CSS Framework: Bootstrap. For that reason, we need to install the bootstrap dependency:

```
npm install --save bootstrap
```

16) 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')
);
```

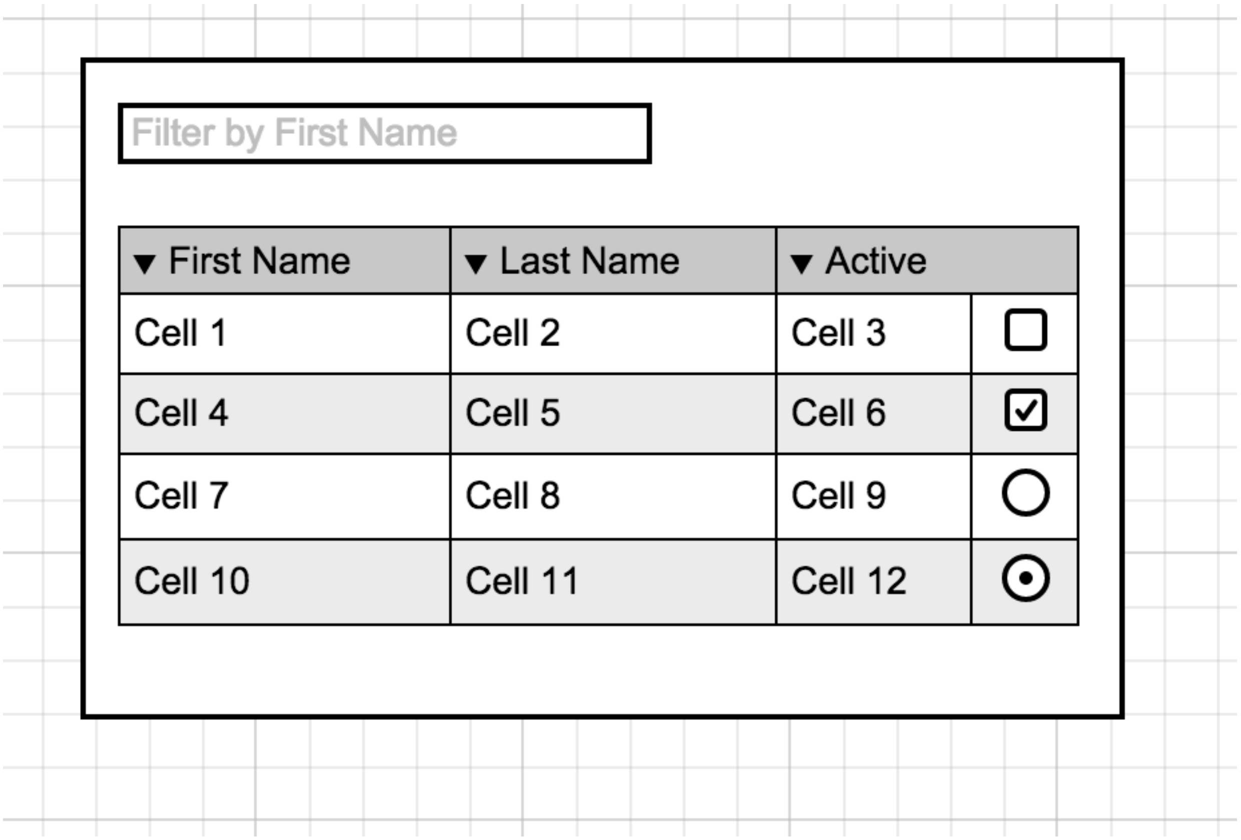
17) Open browser:



Firstname	Lastname	Email
John	Doe	john@example.com
Mary	Moe	mary@example.com
July	Dooley	july@example.com

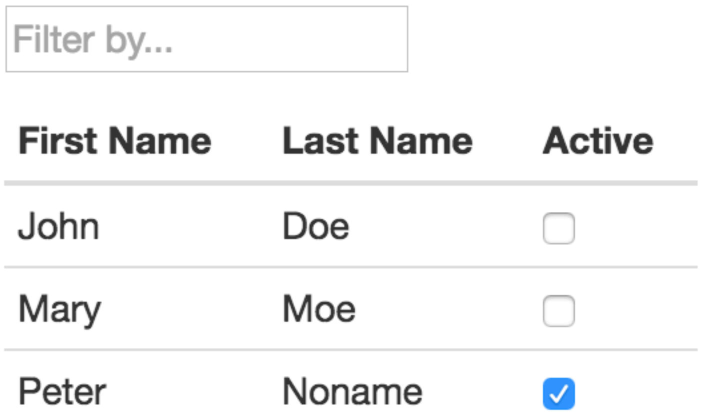
18) Very good! Now we have written our first UI component in JSX, styled it with bootstrap.css.

Task 2: Make markup with JSX for given design:



▼ First Name	▼ Last Name	▼ Active	
Cell 1	Cell 2	Cell 3	☐
Cell 4	Cell 5	Cell 6	☑
Cell 7	Cell 8	Cell 9	○
Cell 10	Cell 11	Cell 12	⦿

19) I assume we all now see something like this in the browser now:



Filter by...

First Name	Last Name	Active
John	Doe	☐
Mary	Moe	☐
Peter	Noname	✓

State VS Props

20) Now let’s add some state and props and remove those hardcoded table values. First of all lets create an 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}
    ]
```

21) Lets pass this data as a state to our GridComponent:

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

22) 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 props:

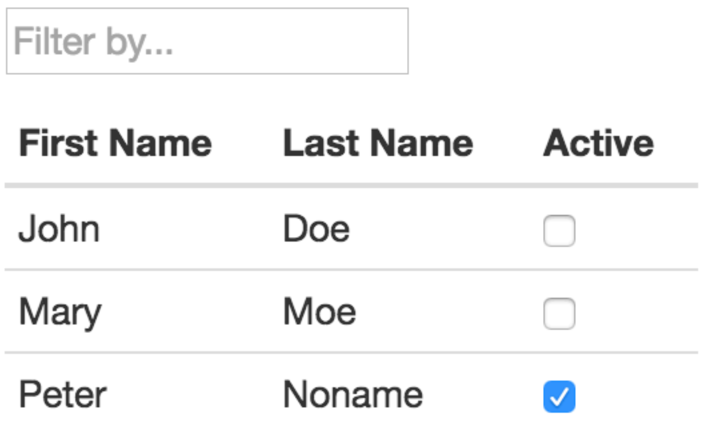
```
class GridRecord extends React.Component {
    render(){
        let {record} = this.props;
        return <tr>
            <th>{record.firstName}</th>
            <th>{record.lastName}</th>
            <th><input type="checkbox" checked={record.active}/></th>
        </tr>
    }
}
```

23) Now update GridComponent’s 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>
        )
    }
}
```

24) 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 we have the same UI view after the update:



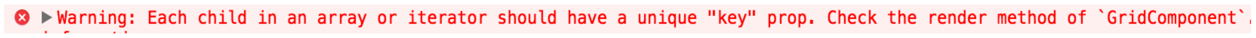
Filter by...

First Name	Last Name	Active
John	Doe	☐
Mary	Moe	☐
Peter	Noname	✓

25) 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>
      </div>
  )
```

26) Now you might see in console something like that:



```
Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `GridComponent`.
```

27) This is happening because of internal optimization in react, if you create components dynamically with iterators you must provide a unique identifier to the key props of iterated element:

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

28) 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
    })
}
```

29) 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)}/>
```

30) If you noticed, we used bind method here. I assume you know what bind does and will explain why we are 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.

31) Create action handler method in your GridComponent:

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

32) And activate it at GridRecord:

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

33) To summarize:

   a. **GridComponent** is responsible for data. It knows how to change it and since we are working with state – it knows how to update dom after data is changed
   b. **GridRecord** knows how to create markup based on incoming props. It knows 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.

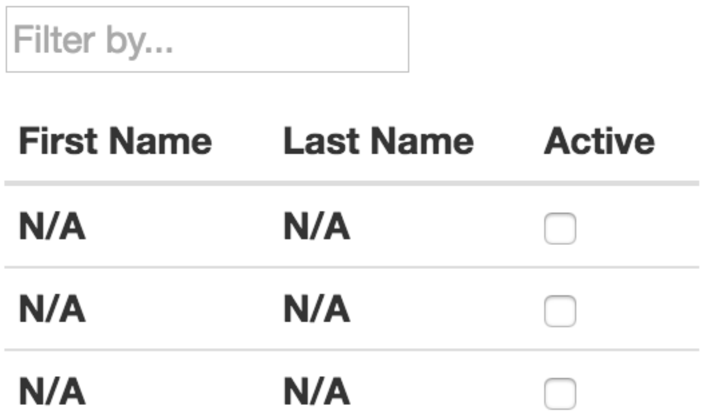
34) Let’s make our components more reusable. First, we will add defaultProps in case something goes wrong and our GridRecord component will get not really what he expects:

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

35) Let now check it:

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

36) So now instead of errors in console and empty grid you will see:



First Name	Last Name	Active
N/A	N/A	☐
N/A	N/A	☐
N/A	N/A	☐

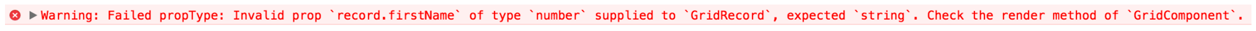
37) Cool, second thing is propTypes:
Install prop-types module: `npm i --save prop-types`

```
import PropTypes from 'prop-types';

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

38) 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}
];
```

39) Open browser console:



```
Warning: Failed propType: Invalid prop `record.firstName` of type `number` supplied to `GridRecord`, expected `string`. Check the render method of `GridComponent`.
```

40) You can remove replace 123 in dataSource back to John.

41) 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)}
```

42) 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
    });
}
```

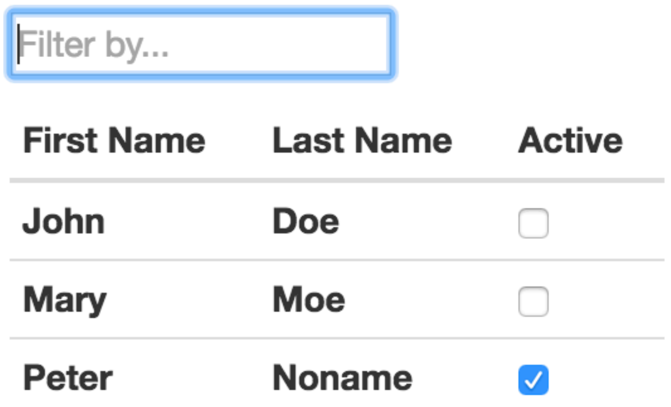
43) Very good but let's say that according to new requirements we need to let user type in this field right after page is loaded. We need to focus it somehow. How can we do it? We need ref:

```
<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:



First Name	Last Name	Active
John	Doe	☐
Mary	Moe	☐
Peter	Noname	✓

React Children

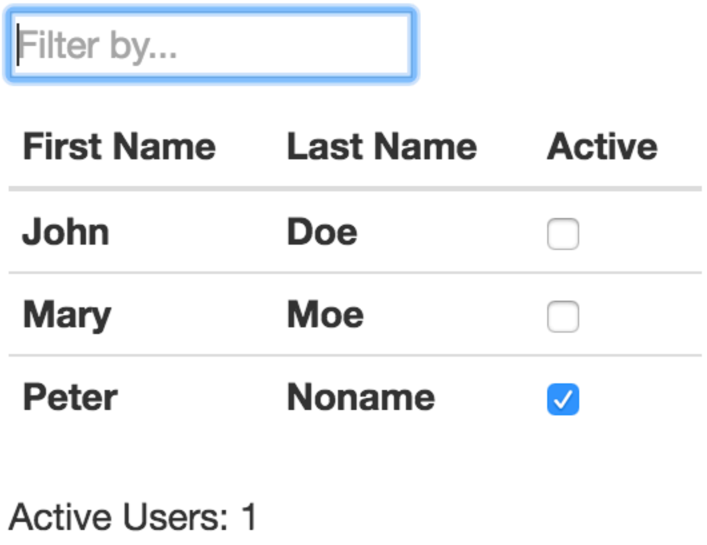
44) Ok, it works, and we are on the half of the way. Now let’s imagine you need to develop numerous grids like this one and you need to provide different rows at the bottom of the grid, with different kinds of information – show how many active users and how many users at all. What will you do? I would 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>
)
```



Filter by...

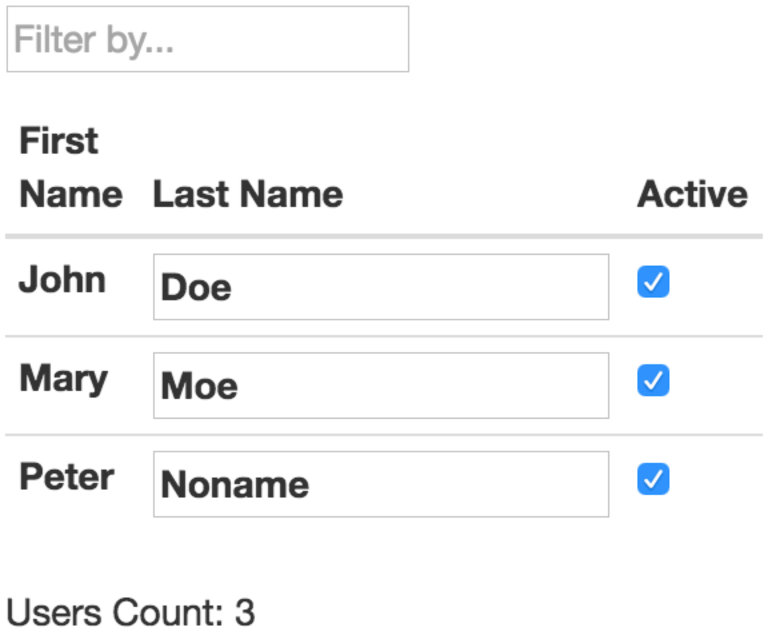
First Name	Last Name	Active
John	Doe	☐
Mary	Moe	☐
Peter	Noname	✓

Active Users: 1

45) And let's 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')
);
```

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



Filter by...

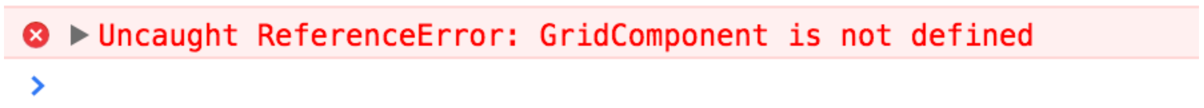
First Name	Last Name	Active
John	Doe	✓
Mary	Moe	✓
Peter	Noname	✓

Users Count: 3

47) Let’s 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

48) Now let’s create grid.js and put GridComponent and GridRecord and dataSource in this file.

49) If you will try to load app now you will see:



```
Uncaught ReferenceError: GridComponent is not defined
```

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

```
import React from 'react';
```

51) 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 {
```

52) We will add import in App.js where we going to use GridComponent and Summaries

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

53) 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')
);
```

54) Now, it is clean, and structured app.

Task 4: Create UserDetail component which will render details of user

1) Create file user-details.js under src folder
2) Create react component to render user details based on 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>html5</span>
                            <span>css3</span>
                            <span>jquery</span>
                            <span>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>
```

Component should show data from this variable:

```
var user = {name:"John Doe",
  about:"Nice guy",
  hobby:"Likes drinking wine",
  skills:["html", "javascript", "redux"]}
```

3) Render this newly created component instead of GridComponent, and see the result in the browser!

Task 5: Create UserDetails component to show all user details.

We have developed GridComponent with mocked data for now and when it loads it shows data from the mock. We will use UserDetails component to show all detailed information about each user, by using UserDetail to show information about each user.

Acceptance criteria:

1) UserDetails component shows data based on mocked data object detailsRecords.
2) UserDetails component can show as many details as many objects specified in source data
3) Source data should be accepted like the following:

```
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#"]
}];
```

React Router 4

1) Install it with

npm install --save react-router-dom

2) Import necessary classes from react-router-dom:

```
import {HashRouter, Switch, Route, Link} from "react-router-dom";
```

3) Update App.js with router definitions. Use this configuration for the router:

```
<HashRouter>
    <div>
        <Header />
        <Switch>
            <Route path="/grid" component={GridComponent}/>
            <Route exact path="/details" component={UserDetails}/>
            <Route path="/details/:id" component={UserDetails}/>
        </Switch>
    </div>
</HashRouter>
```

Note that links <Link> can be used only inside router (<HashRouter> or <BrowserRouter>).

4) Define App component – it will be the main menu of our application:

```
const Header = ({children}) =>
            <div>
                <h1>Our awesome app</h1>
                <ul role="nav">
                    <li><Link to="/grid">Grid</Link></li>
                    <li><Link to="/details">Details</Link></li>
                </ul>
                {children}
            </div>;
```

5) Update dataSource by adding the ids:

```
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}
];
```

Also update table in GridComponent:

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

6) Define method in UserDetails component which will filter details by id, leaving only details with id from URL parameter or showing all details otherwise:

```
filterDetails(id) {
    if (id) {
        this.setState({
            details: detailsRecords.filter((record) => {
                return record.id == id;
            })
        })
    } else {
        this.setState({
            details: detailsRecords
        })
    }
}
```

7) We should perform filtering on component mount, so that UserDetails contain only details with certain id:

```
componentWillMount() {
    this.filterDetails(this.props.match.params.id);
}
```

8) Also note that component UserDetails is not recreated when we go from URL "/details" to "/details/:id" and back – it is just updated. Method `componentWillMount`() will not be executed in this case. Also we cannot use `componentWillUpdate()` because it's not allowed to use setState() inside this method to avoid endless loops. To retrieve id and rerun filter in this case, we can use `componentWillReceiveProps`() method, which doesn't execute when we call setState(), so it's safe to use setState() from this method. We can do it this way:

```
componentWillReceiveProps(props) {
    this.filterDetails(props.match.params.id);
}
```

9) Also, you shoud take history object to push updated URL when user clicks the link. History is available in the component used in a router via this.props.history, so it's available in GridComponent. We need to pass it to GridRecord:

```
<GridRecord record={record} history={this.props.history} … />
```

Then in GridRecord component update `showUserDetails` method:

```
showUserDetails(e){
    e.preventDefault();
    this.props.history.push(`/details/${this.props.record.id}`);
}
```

Add first column to GridRecord:

```
<th onClick={this.showUserDetails.bind(this)}>
        <a href="#">{record.id}</a>
</th>
```

10) Now you can check the routing with React Router 4.

Redux.

55) As of now we have built nice app which actually does very common things at UI. But we want to try to switch from vanilla React to React + Redux to show you how easy it is.

56) First thing you need to do is: “npm install --save redux react-redux”

57) 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

58) Move grid.js and user-details.js and summaries.js from src folder to Components and app.js move to Containers

59) Update imports in App.js for UserDetail, GridComponent and Summaries to import them from the Components folder

60) Create index.js in Reducer, Actions, Store, Constants folders

61) Ok, now we are ready to start. Update the App.js:

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

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

```
const store = configureStore();
```

```
render(
    <Provider store={store}>
        <HashRouter>…</HashRouter>
    </Provider>,
    document.getElementById('app')
);
```

62) Let me explain what we did so far:

Redux app needs to be wrapped in Redux Provider, that’s what we did – wrapped our app, to let redux manage our components.

We have applied the store (don’t forget that we will have single instance of the store) to Provider.

63) Now let’s figure out what we going to 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.

64) 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 expressions? It is default parameter values comes from ES6 syntax. Why do we need it? We need it to set value for a first call to reducer. When you just start your app, redux will require to fill the store with some initial value and since store state is undefined at first call, we set default values directly in reducers. Let’s define the initial value objects:

```
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#"]
    }];
```

65) Okay now, we have reducers for the grid, and user-detail screens. Now let’s merge reducers into 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 app. 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 going to 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
    }
}
```

66) Ok, let's move on for now. But we will come back to them. Let summarize what we did so far:
   a. We created reducer which merges two other reducer and just simply provide default state as of now
   b. We created store with this reducer
   c. We wrapped our existing app in Redux wrapper container

67) Now we can provide data to the components. Open grid.js and do the following:
   d. Remove hardcoded data
   e. Remove export from the GridComponent
   f. Add this:

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

```
GridComponent.propTypes = {
    records: PropTypes.array.isRequired
};

function mapStateToProps(state) {
    return {
        records: state.grid
    }
}

export default connect(
    mapStateToProps
)(GridComponent)
```

68) 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)=>{
```

69) And we need to remove this and dataSource object:

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

70) If you will refresh the page now you supposed to see the same component, but it's now driven by React + Redux

71) But for now, we pass just a half of the way. We can read from store, but how can we update the store? Let's do it. First of all, we need to modify toggleActive method:

```
toggleActive(index){
    let {dispatch} = this.props;
    dispatch({
        type:"TOGGLE_ACTIVE",
        value:index
    });
}
```

72) 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
  }
}
```

73) That’s it reloads the page and you will see data is updated.

Task 6: Make filter by textfield.

Acceptance criteria:

1) With use of dispatch, action, and reducer filter records in the grid according to firstname matching data in textfield

74) Now let’s 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';
import PropTypes from 'prop-types';
```

```
class UserDetails extends React.Component {
    render(){
        return (
            <div>
                {this.props.details.map((detail, i)=>{
                    return <UserDetail key={i} detail={detail}/>
                })}
            </div>
        )
    }
}

UserDetails.propTypes = {
    details: 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 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: Show only the appropriate user details when user clicks on the grid

---

Acceptance criteria:

1) Property that comes to user-details able to sort source records
2) In the source records, If 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

75) Okay, that's good. But let's say you will need to rename FILTER_DETAILs to SEARCH_DETAILS. What you will need to do is to change it in reducer and component? It's not very convenient. We can replace action with action creators. But first let's 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';
```

76) 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
    }
}
```

77) We should update reducer after that:

```
import {TOGGLE_ACTIVE, FILTER, FILTER_DETAILS} from '../Constants'
```

```
case TOGGLE_ACTIVE:
case FILTER:
case FILTER_DETAILS:
```

78) We just defined all actions as constants in one place and action creators which depend on value and action are also separate. Let's finally update the dispatches:

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.match.params.id));
}
```

79) 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

80) Now, we have dummy server which does only one thing on any request it generated json data with gridRecords and detailsRecords in it. Let's figure out how can we connect it to our application.

81) Okay now we need to fire action creator for it. But how can we do it? Let start first creating 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';
```

82) 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
    }
}
```

83) 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){
```

84) 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]
    });
```

85) Since we updated grid state shape we need to update connect and mapStateToProps:

```
GridComponent.propTypes = {
    records: PropTypes.array.isRequired,
    filtered: PropTypes.array.isRequired,
    loading: PropTypes.bool.isRequired
};

function mapStateToProps(state) {
    return {
        records: state.grid.records,
        filtered: state.grid.filtered,
        loading: state.grid.loading
    }
}
```

86) 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());
    })
}
```

87) If you refresh the page now you supposed to see data in the grid which comes from server.

88) 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

89) 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

90) 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

91) Go to your Store/index.js and change 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;
}
```

92) 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());
        })
    }
}
```

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

```
import {filterGrid, toggleActive, loadDataInGrid} from '../Actions'
```

94) loadData method will look also quite more accurate:

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

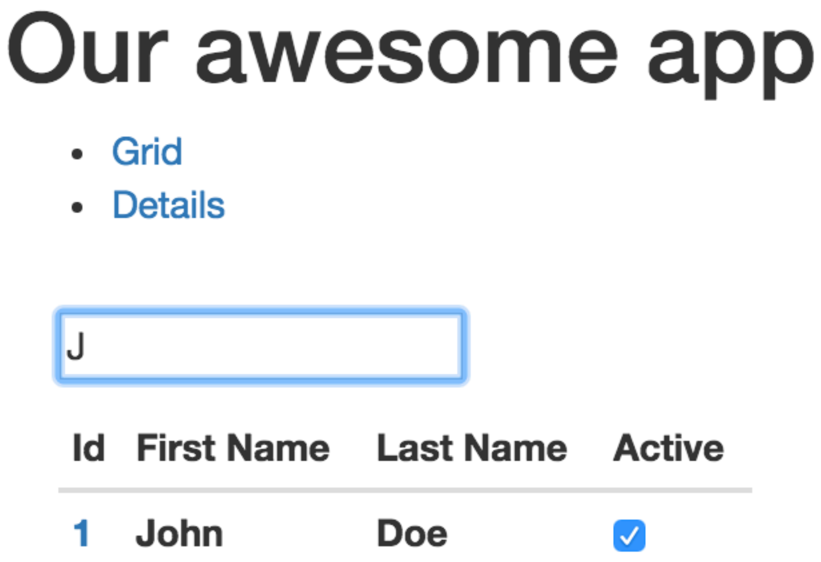
95) If you reload your page now it should work.

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



Our awesome app

- Grid
- Details

J

Id	First Name	Last Name	Active
1	John	Doe	✓

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

4) When loading equals false GridComponent renders current GridComponent markup again

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

Acceptance criteria:

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

Testing React.js

96) We are done with our coding, now let's write some tests. First of all, we need to create to install everything we going to use:

```
npm install --save-dev enzyme enzyme-adapter-react-16 react-test-renderer jsdom enzyme
chai
```

97) Install enzyme-adapter for react:

```
npm i --save-dev enzyme enzyme-adapter-react-16
```

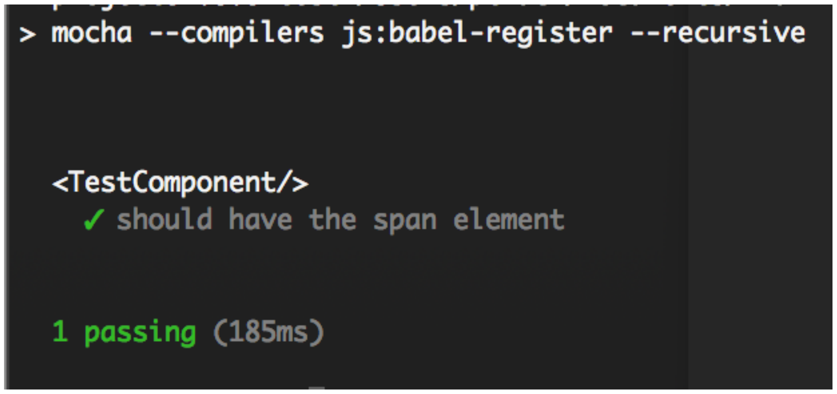
98) Add the following setup into src/setupTests.js:

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({ adapter: new Adapter() });
```

99) Let's now create some very simple test in src/tests/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);
    });
});
```

100) Now type npm test in your console and you will something like that:

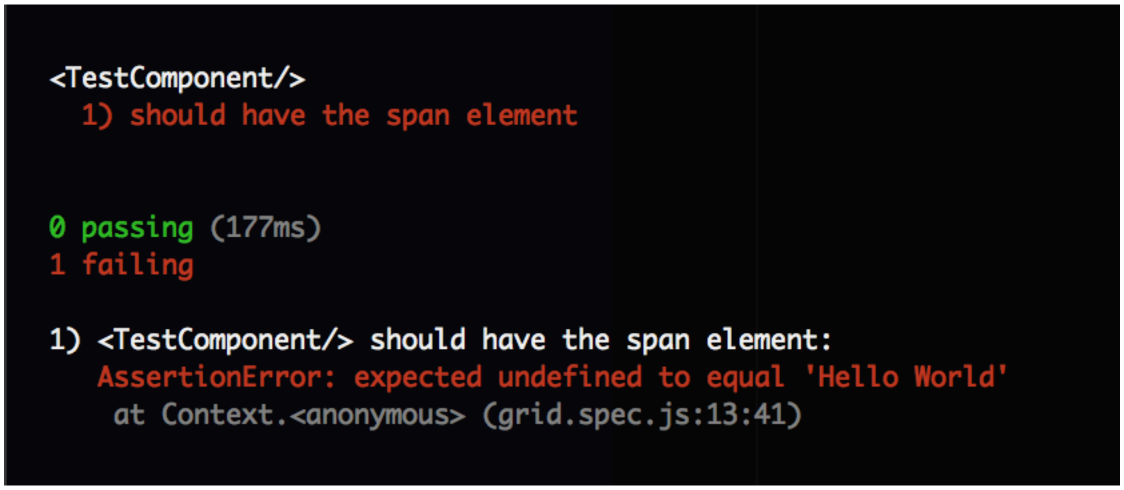


```
> mocha --compilers js:babel-register --recursive


  <TestComponent/>
    ✓ should have the span element


  1 passing (185ms)
```

101) Let’s 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 see in the console? I think something like this:

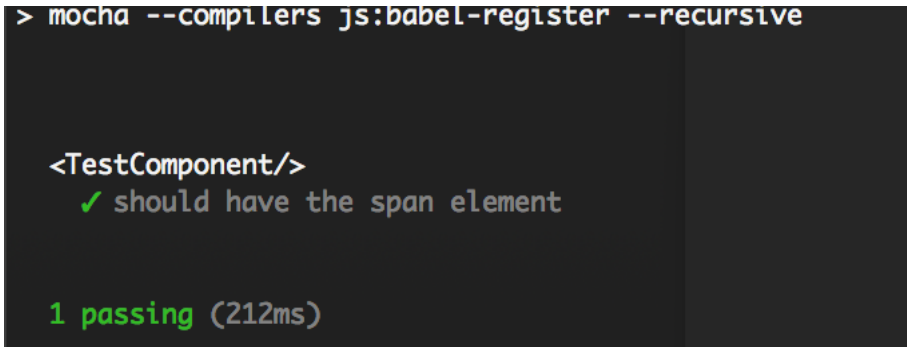


```
  <TestComponent/>
    1) should have the span element


  0 passing (177ms)
  1 failing

  1) <TestComponent/> should have the span element:
     AssertionError: expected undefined to equal 'Hello World'
      at Context.<anonymous> (grid.spec.js:13:41)
```

102) You might be curious why do we get this error? We get it because we use shallow render which is actually very useful when we are testing our components as a unit, but here shallow doesn’t receive actual component’s props, so we should replace our shallow with mount method:

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

103) Now rerun the test:



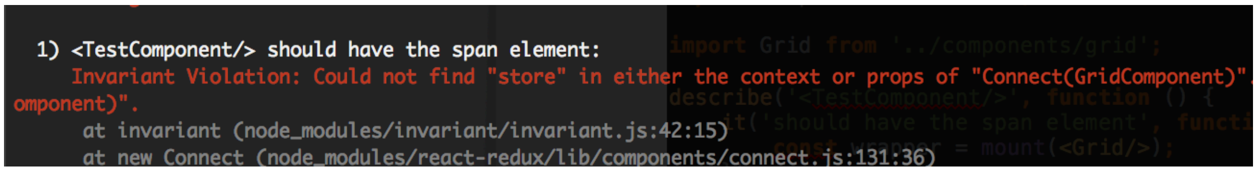
```
> mocha --compilers js:babel-register --recursive



  <TestComponent/>
    ✓ should have the span element


  1 passing (212ms)
```

104) We played with React.js testing a little bit, now let’s cover with tests our application:

105) 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) <TestComponent/> should have the span element:
     Invariant Violation: Could not find "store" in either the context or props of "Connect(GridComponent)",
omponent)".
      at invariant (node_modules/invariant/invariant.js:42:15)
      at new Connect (node_modules/react-redux/lib/components/connect.js:131:36)
```

106) 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)
```

107) Now extract Containers from the Components folder, and move them to Containers folder:

108) 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)
```

109) Now update your App.js:

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

```
render(
    <Provider store={store}>
     <HashRouter>
        <div>
            <Header />
            <Switch>
                <Route path="/grid" component={GridContainer}/>
                <Route exact path="/details" component={UserDetailsContainer}/>
                <Route path="/details/:id" component={UserDetailsContainer}/>
            </Switch>
        </div>
     </HashRouter>
    </Provider>,
    document.getElementById('app')
);
```

110) Remove connect and mapStateToProps from component grid and user-details, remove import and it should work.

111) Now our GridComponent is testable, and we can easily import 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('<GridComponent/>', 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