formation-react

Labs

© Copyright 2019 Zenika. All rights reserved

Foreword

In order to practice concepts seen during the course, we will develop an application which list "golden rules" for developers. These rules are the best practices that every developers should follow!

We will start from a single static HTML page, that will be enriched bit by bit, after each chapter. The result will be a modern React application, rich and reactive.

It uses the CSS Twitter Bootstrap framework to have clean CSS base code.

It doesn't embed any JavaScript code. The React application will be generated with `create-react-app` and external libraries will be installed through npm (don't download them manually).

© Copyright 2019 Zenika. All rights reserved

Prerequisite

Preparation

You received an archive called TP.zip that once unzipped looks like :

```
├── resources
│   ├── data.json
│   ├── edition.html
│   ├── navigation.html
│   ├── rule.html
│   └── superagent.js
└── server
    └── ...
```

Installation

We will use `create-react-app` to bootstrap the application. It is based on this tools :

- ES2015.
- Babel for ES2015 to ES5 transpilation.
- Webpack for generating a final bundle, runnable in the browser.
- Webpack-dev-server to have a lightweight web server and live reload.
- hot reloading with Webpack.
- And a lot more

Note: Developing with ES5 is possible (Babel converts only ES2015 code).

The installation will be done through *Node.js* (which must be available on the workstation), available on https://nodejs.org.

Information: For Windows users only, If *Node.js* is already installed on your workstation, take care on the version used (check with `node --version` command). For the following labs, *Node.js* >= v4 will be needed (don't use v0.12).

Install `create-react-app` :

```
npm install -g create-react-app
```

Then, use this module to create a `client` application.

```
create-react-app client
```

Finally, you can start the application with the command:

© Copyright 2019 Zenika. All rights reserved

```
cd client
npm start
```

If everything works well, you can now open the browser at `http://localhost:3000` URL, you should see a welcome message and a spinning React logo.

To see the hot-reload in action, open the `App.js` file and change one of the text string: the text is updated in the browser immediately, without manual refresh.

© Copyright 2019 Zenika. All rights reserved

Lab 1: React and JSX setup

For this first lab, we will create our first components and replace the default one generated by `create-react-app`.

Objectives:

- Create a component with plain JavaScript.
- Create a component using plain JSX.
- Display React components.

React setup

Even if React is already up and running, it is important to understand how React is bootstrapped in a Web application. That is why we will start by replacing the generated app by a hand-made "Hello World" component.

Open the `index.js` file and replace the existing `ReactDOM.render` instruction by these lines:

```
const reactElement = React.createElement('div', null, 'Hello World');
const domElement = document.getElementById('root');

ReactDOM.render(reactElement, domElement);
```

The application should display "Hello World" in the browser.

Bootstrap setup

`create-react-app` doesn't embed Twitter Bootstrap by default. It must be installed manually:

```
npm install bootstrap@3.x.x --save
```

Then load the main CSS file in `index.js` using the `import` keyword:

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

Note: Importing CSS files is allowed by the Webpack configuration used by `create-react-app` under the hood, it is not part of the EcmaScript standard.

© Copyright 2019 Zenika. All rights reserved

First components

The application will display rules that any developers should respect.

The HTML code to display a rule, ready to be copy-pasted, lies in the `resources/rule.html` file.

Be aware: When copying HTML code in the `render` method, think about the syntax differences between HTML and JSX (`class` attribute must be replaced by `className`).

Displaying the list

We will start by creating a list in a React component:

- In `src` folder, create a file named `RuleList.js` .
- In this file, create a class inheriting from `React.Component` for this new component and then export it by default (see annexes).
- Implement the `render` method:
    - The rules to display will be provided as props.
    - The function must return a root JSX element (`<div>` for instance) containing a JSX block (equivalent to the HTML code mentioned above) for each rule.

Note: From React 16, the render function can now return an array of elements without any wraping element. Each element in the array needs a `key` prop uniquely defined. You also can render multiple elements with the `React.Fragment` component.

Tip: To create a React list from a JavaScript array, use the `map` function:

```
const elements = array.map(item => React.DOM.div(null, 'Hello ' + item));
```

Then bootstrap the application:

- In the file `src/index.js` , import the component previously created.
- Provide the rules to display: drag the file `data.json` from the `resources` directory into `src` and import it with ES2015 import:

```
import rules from './data';

console.log('rules = ', rules);
```

- Call `ReactDOM.render` method to render the element inside the DOM element with the id `root` .
- Check if the application is working well.

Externalize a component

To display the rules, we need to duplicate the HTML code for each element: this is a typical use case to create a new component:

- In `src` folder, create a file named `Rule.js` : it will contain the code of our new component.
- In this file, create a class inheriting from `React.Component` and export it by default.
- Implement the `render` method:
    - The rule to display will be provided as props.

Now, use this component:

- In the file `RuleList.js` , import the `Rule` component.
- Update `render` method to call it.
- Check if the application is working well.

Bonus - Custom CSS

Since the title panel is clickable, it could be a good idea to give the user a hint with a pointer (hand) cursor.

They are many approaches about handling styles in React, the one used in `create-react-app` consists to declare a small CSS file for each component that contains the specific styles.

- Create a `Rule.css` file sibling to `Rule.js` .
- Add a CSS property to display the "hand" cursor when the user moves the mouse over the title panel.
- Import the CSS file in `Rule.js` :

```
import './Rule.css';
```

© Copyright 2019 Zenika. All rights reserved

Lab 2: State

Objectives

We will add two new features to the application to make it more dynamic.

Each rule must be fold/unfold to hide/display the description:

- By default, the description is displayed.
- When the user clicks on the title, the description is hidden or displayed depending on its current state.

We will also add a feature to count "likes" or "dislikes" on each rule.

Objectives:

- Initialize default state.
- Update component according to the state.
- Understand the differences between props and state.

Handle component state

By default, each rule displayed must be "unfold":

- Update `Rule.js` file:
    - Implement the constructor to initialize the default component state. The default state is a simple JavaScript object containing a `folded` boolean property. By default, this property must be `false` to display the description.
    - Be aware, use `super` to initialize props properly, for instance:

```
import React from 'react';

class MyComponent extends React.Component {
    constructor(props) {
        super(props);
        // Your stuff
    }
}
```

- Update the `render` method:
    - Depending on the `folded` value, display or hide the description (tip: use the `hidden` CSS class to hide a DOM element).
    - Depending on the `folded` value, update CSS class of icon in the title: if description is visible, the icon must have `glyphicon-chevron-down` class, `glyphicon-chevron-up` otherwise (see annexes to update *React* CSS classes with ease).

- Test the component behavior (it must work as before).

Now, the state will be modified depending on user actions:

- Create a function which toggle the `folded` value (be aware of using `setState` method).
- When the user clicks on the title of a rule, call that function to display / hide the description.
- Check if the application is working well.

Bonus: To not display useless elements, make sure that the description is hidden by default if it is empty.

"likes" feature

To handle numbers of "likes" or "dislikes", we will start by creating a new component:

- Create a file named `LikeBtn.js` .
- Create a class inheriting from `React.Component` and export it by default.
- Both buttons are almost equivalent (visually and semantically), so we will use the same button for "like" and "dislike": button type will be provided as props to generate the appropriate HTML code.
- Implement the constructor:
  - Initial counter value will be provided as props. This value will be updated by user actions (click on the button), the component state must be initialized with that value.
  - Don't forget to call `super` method with props.
- Implement `render` method by using JSX.
- Create a method to increment the counter and call that method when clicking on the button.
- In the component displaying a rule, use that new component.

Note: for now, counters values start from 0 after each page refresh: there is no persistence mechanism for the moment (it will be added with the following labs).

Props validation

As the application is growing, it is interesting to use the React prop-types validation feature.

- Install the `prop-types` module.
- For each component file:
  - Import the module: `import PropTypes from 'prop-types';`
  - Attach a `propTypes` object property the class variable. Example: `Rule.propTypes = {}`
  - Define a type for each props used in the component.
- **Bonus**: configure your IDE to enjoy props autocompletion thanks to the propTypes definition.

© Copyright 2019 Zenika. All rights reserved

Lab 3 - Tests

The main goal of this lab is to write unit tests for the components by using the official React test framework : *Jest*. The configuration is already done by `create-react-app` .

First test

We will first write tests for the component displaying a rule:

- In the `src` folder, create a folder named `__tests__` (as a reminder, *Jest* scans folder with that name and run tests).
- In this folder, create a file named `Rule.test.js` .
- Implement the first test to check that a rule is displayed correctly:
  - Import React and utilities tool (import the `react-dom/test-utils` module).
  - Import the rule component.
  - Create a test suite with `describe` .
  - In this test suite, create a first test case with `it` .
  - In this test case, instantiate the component and use the `renderIntoDocument` method to render the component in the DOM.
  - Add assertions to check that the component renders well (for instance, check that the title and description panels contain the expected values).

Tip : To check the text value of a DOM element, use the following code :

```
const element = React.createElement(MyComponent);
const component = TestUtils.renderIntoDocument(element);

const subElement = TestUtils.findRenderedDOMComponentWithClass(component, 'foo');
const domNode = ReactDOM.findDOMNode(subElement);
expect(domNode.textContent).toEqual(rule.title);
```

Run the following command to run the tests:

```
npm test
```

You can leave the process running as long as you develop. Jest watches the sources and re-run the tests if anything is changed.

Shallow Rendering

Let us try *shallow rendering* :

- Install module `react-test-renderer` with :
  `npm install react-test-renderer --save-dev`

© Copyright 2019 Zenika. All rights reserved

- Import it in the test file : `import ShallowRenderer from 'react-test-renderer/shallow';`

- Add a new test with `it` .

- In this test, create a new renderer.

- Use it to render the rule component.

- Get the renderer output and test the component is working well, for instance:
  - Check css classes.

  - Check children.

Tip: You can access to css classes and children through `props` :

```
const result = renderer.getRenderOutput();
expect(result.props.className).toBe('foo');
expect(result.props.children).toEqual(<p>Hello World</p>)
```

Fake click event

In the second lab, we have created a component to increment a counter. The goal of the exercise is to use the React utilities tool to fake user actions:

- In the folder `src/__tests__` , create a file named `LikeBtn.test.js` .

- In this file, specify that the component should not be mocked.

- Import React and the counter component.

- Create a new test:
  - Instantiate the component and check that the initial counter value displayed is 0.

  - Use `TestUtils.Simulate.click` method to simulate a click on the component.

  - Check that the counter value has been incremented.

© Copyright 2019 Zenika. All rights reserved

Lab 4: Redux

For this lab, we will change our application to use Redux.

Note: `redux` and `react-redux` libraries are already available (see `package.json` file) and installed thanks to the `npm install` command.

Load rules

For the sake of separation of concerns, we will load the rules in a dedicated action instead of importing them in `index.js`.

Action creators

Create the `RULES_LOADED` action:

- In a new folder called `actions`, create a file named `rules-actions.js`.
- In this file:
  - Import the rules from the `data.json` file.
  - Create a function named `loadRules` which returns an action named `RULES_LOADED` containing the rules.
  - Export the function.
  - Export the action name as a constant too.
- Bonus: Write unit tests for this action.

Reducer

Create a reducer to manage the rules:

- Inside the `src` folder, create a folder named `reducers` and then create a file named `rules-reducer.js`.
- In this file create a function:
  - With state as first parameter (initialized by default with `[]`) and an action as second parameter.
  - Handle the `RULES_LOADED` action:
    - Save the rules from the `RULES_LOADED` action into the state.
  - Export the reducer
  - /!\ Be aware to not mutate the `state`
- Bonus: Write unit tests for this reducer.

© Copyright 2019 Zenika. All rights reserved

Store

Configuration of the Redux store:

- Inside the `src` folder, create a folder named `store` and then create a file named `app-store.js`.
- We have only one reducer but this is rarely true, create a future-proof global reducer using `combineReducers`:

```
import { combineReducers } from 'redux';
import rulesReducer from '../reducers/rules-reducer';

const reducer = combineReducers({
  rules: rulesReducer
});
```

- In this file, use `createStore` from the Redux API to create the store. Give the global reducer as parameter.
- Export the store.
- Bonus: Install the `redux-dev-tools` chrome extension and enable it in your code:

```
export default createStore(
  reducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Update React components

We will now connect our component to Redux:

- Create a container for the `RuleList` component:
  - Create a `mapStateToProps` function: it must return an object with a key named `rules` containing the rules available in the state
  - Create a `mapDispatchToProps` function: it must return an object containing a key named `fetchRules` that is a function that dispatch the `loadRules` action
  - Call `this.props.fetchRules` in the `componentDidMount` function of the `RuleList` component.
  - Use the `connect` function from Redux to connect the component and the store (don't forget `mapStateToProps` and `mapDispatchToProps`).
  - Export the container.
- In `index.js`, provide the store to the application thanks to the `Provider` component.
- Use the `RuleListContainer` instead of the `RuleList` component.

© Copyright 2019 Zenika. All rights reserved

- Bonus: Write unit test.

Likes and Dislikes

Action creators

Create two actions `DO_LIKE` and `DO_DISLIKE` :

- In the `actions` folder, create a file named `likes-actions.js` .
- In this file, implement a function named `doLike` :
  - This function has one argument, the rule identifier
  - It must dispatch an action named `DO_LIKE` , containing the rule identifier
- Create a function named `doDislike` :
  - This function has one argument, the rule identifier
  - It must dispatch an action named `DO_DISLIKE` , containing the rule identifier
- Export the functions.
- Bonus: Write unit tests for these actions.

Reducers

Update the reducer that manage the rules:

- Handle `DO_LIKE` and `DO_DISLIKE` actions:
  - `DO_LIKE` action increments `likes` value for the rule whose identifier is given to the action.
  - `DO_DISLIKE` action increments `dislikes` value for the rule whose identifier is given to the action.
- /!\ Be aware to not mutate the `state`
- Bonus: Update unit tests for this reducer.

Tip: To find an object in an array, you can use the `findIndex` function provided by `lodash` :

```
import findIndex from 'lodash/findIndex';

...

const index = findIndex(array, {'id': value})
const newObject = { /** do something **/ };
const newArray = [...array]
newArray[index] = newObject;
```

Update React components

© Copyright 2019 Zenika. All rights reserved

- Create a container for the `LikeBtn` component:
  - Provide the rule to `doLike` and `doDislike` functions in the `LikeBtn` component.
  - Trigger the corresponding actions when clicking on the button.
  - Use the `connect` function from Redux to connect the component and the store.
- Use the container in the `Rule` component
- Bonus: Write unit test.

© Copyright 2019 Zenika. All rights reserved

Lab 5 - REST Architecture

The goal of this lab is to use the REST API provided to get the data.

The server provides a REST API with the following endpoints:

- `GET` `/rest/rules` : Get all the rules.
- `GET` `/rest/rules/:id` : Get the rule with the id specified in the URL.
- `POST` `/rest/rules` : Create a new rule.
- `PUT` `/rest/rules/:id` : Update rule with the id specified in the URL.

In order to increment "likes" and "dislikes", the server also provides the following endpoints:

- `POST` `/rest/rules/:id/likes` : Increment "likes" number for the rule identified with the id in the URL.
- `POST` `/rest/rules/:id/dislikes` : Increment "dislikes" number for the rule identified with the id in the URL.

To start the server, open a new terminal and run the following command:

```
cd server
npm install
npm start
```

In development, the backend server is often separated from the web server, it could lead to errors related to cross-origin (CORS) when calling the backend. Fortunately, `create-react-app` is able to proxify requests to a particular host:

- Add this line in the `package.json` file:

```
  "proxy": {
    "/rest": {
      "target": "http://localhost:4000"
    }
  }
```

- Restart `create-react-app` .

Both URLs must return the same thing now:

- http://localhost:3000/rest/rules
- http://localhost:4000/rest/rules

Dependencies

© Copyright 2019 Zenika. All rights reserved

Redux doesn't handle asynchronous actions, for this purpose, install `redux-thunk` and configure it when creating the store.

To display rules to users, we will not import the rules from a JS file anymore, we will call the REST API instead. For this lab, we will use `superagent` (see examples in slides). Install `superagent`

Fetch data

As Redux provides a excellent separation of concerns, the only file we have to update is `rules-actions.js`:

- Import the `superagent` library.
- In the `loadRules` function, use `superagent` to call `/rest/rules` URL:
  - Get data using the `end` callback.
  - Dispatch a `RULES_LOADED` action once data received.
  - The function must return a function in order to dispatch the action manually thanks to `redux-thunk`.
- Check that the application is working well.

Handle likes & dislikes

Finally, update `LikeBtn` to call REST API when clicking on the buttons:

- In `likes-actions.js` file, import `superagent` library.
- In `doLike` function, use `superagent` to call `/rest/rules/:id/likes` URL:
  - Add a parameter to the function, that is the id of the rule to update
  - Get data using `end` callback.
  - Dispatch `DO_LIKE` action once data received.
  - The function must return a function in order to dispatch the action manually thanks to `redux-thunk`.
- In `doUnlike` function, use `superagent` to call `/rest/rules/:id/dislikes` URL:
- Add a parameter to the function, that is the id of the rule to update
- Get data using `end` callback.
- Dispatch `DO_UNLIKE` action once data received.
- The function must return a function in order to dispatch the action manually thanks to `redux-thunk`.
- In `LikeBtn.js` file, add the rule id as parameter when calling `doLike` and `doUnlike`

© Copyright 2019 Zenika. All rights reserved

Bonus - Unit tests

We will now update the unit tests:

- Mock `superagent` :
    - Create the directory `__mocks__` in the `client` directory.
    - Drag the provided mock `superagent.js` from the `resources` folder into this directory

This mock will be automatically loaded by `Jest` . It emulates the `superagent` chainable API.

- In `rules-actions.test.js` file, mock the implementation of the `end` function: for convenience, the function can called the callback synchronously.

Example:

```
import superagent from 'superagent';

describe('GET Rules', function() {
    beforeEach(function() {
        superagent.end.mockImplementation(function(callback) {
            callback(null, {
                ok: true,
                body: [
                    { id: 1, title: 'My Rule' }
                ]
            });
        });
    });
});
```

- Some changes must be done in the `likes-actions.test.js` and `rules-actions.test.js` files, because they return a function:
    - Create a mock with jest: `const dispatch = jest.fn();`
    - Call the function return by the actions with the mock: `actions.doLike(5)(dispatch);`
    - Check that mock is called with the expected parameters
- Re-run tests and make sure that everything is working well (you can add assertions to make sure that `superagent` is called).

© Copyright 2019 Zenika. All rights reserved

Lab 6 - Routing

Until now, our application contains only one page: the rules list. We will now add a routing system for the users to display the form to add new rules.

Objectives:

- Setup a real Single Page Application
- Use the router to react to URL changes.
- Navigate through the application.

Setup the router

Install the router:

```
npm install react-router-dom
```

At first we will update the `index.js` file to start the application with the router provided by the `react-router-dom` library:

- In the `index.js` file, import the `BrowserRouter` and `Route` from `react-router-dom`.
- Instead of instantiating a `RuleListContainer` directly, define a route that will instantiate a `RuleListContainer` on `/` path.
- Check that the application is still working.

Navigation bar

We need to enrich the current page with some pieces of layout like a navigation bar: that menu will contain all the links to navigate through the application. Those links will have an `active` css class when they will be active.

Let us manage the navigation bar as a React component:

- Create a file named `Header.js` containing the code of the new component.
- Create a class named `Header` (default export) and implement `render` method:
  - Get HTML code from `resources/navigation.html` file (`nav` part).
  - Be careful with JSX (Change `class` attributes by `className` for instance).
  - Replace links (`a` tag) by the `Link` component from the `react-router-dom` library.

Create the application layout:

- Create a file named `Layout.js`

© Copyright 2019 Zenika. All rights reserved

- Create a new React component (named `Layout`) that displays the navigation menu (`Header` component). JSX code must look like:

```
<div>
  <Header />
  <div className="container-fluid">
    <div className="container">

    </div>
  </div>
</div>
```

- Change the router configuration to use `Layout` component on `/` path.
- In the div with the class `container`, define a route that match the exact `/` path and instantiate a `RuleListContainer` when active.
- Check that the application is working well.

Navigate to rule creation page

We will now create a new component to add rules:

- Inside the `src` folder, create a file named `RuleForm.js`.
- In this file, display the form. The HTML code is available in the `resources/edition.html` file, be aware when converting to JSX).

Bootstrap uses a bunch of HTML tags to display a form field nicely. A best practice consists in splitting each field in a single component. It will be useful when we will deal with validation (next lab).

- Create a file named `RuleTitleField.js`.
- Externalize the field that displays the title (including the `div` with the class `form-group`) in a component named `RuleTitleField`.
- Do the same for the description.
- Use both `RuleTitleField` and `RuleDescriptionField` in the `RuleForm` component.
- In `Layout.js` file, add a new route with `/new` path: it must display the form to add rules.
- Update links in navigation bar to navigate through the application.

Navigate to rules modification page

To complete the application, we need to be able to modify a rule by navigating to the form page and fill the form with the rule properties (the submission of the form will be done in the next lab).

- In `Layout.js` file, add a new route:
  - The route path is `/edit/:id` (`id` is a dynamic value depending on the rule to update).

© Copyright 2019 Zenika. All rights reserved

    - Update the `Rule` component to navigate to the form by providing the rule identifier (use the the `Link` component from `react-router-dom` library).
    - Check that the application is working well.

The navigation should be working now. In order to modify a rule, we have to fill the form with the current rule values:

- Create a container in a new file named `RuleFormContainer.js` that connect `RuleForm` to the Redux store.
- In this container, get the rule identifier from the properties:
    - The properties are available as the second parameter in the `mapStateToProps` function.
    - Think about the rule creation when no parameter is provided.
    - Get the rule to update from that identifier.
- In the component, use the `defaultValue` property on each field to fill the form.
- Use the container instead of the component in the route definition.
- Check that the application is working well.

© Copyright 2019 Zenika. All rights reserved

Lab 7 - Forms

In this lab we will:

- Setup Redux-form
- Validate user input when saving rules
- Distinguish the edition and the creation
- Submit the form to the server

Objectives:

- Manage forms with Redux-form.
- Create reusable form field component.
- Validate forms.
- Give feedback to users.
- Handle the submitted values

First, we have to install the library:

```
npm install redux-form
```

Form binding

With Redux-form, each form has a specific entry in the store that makes its management easy. For that purpose, we need to decorate the form component with the Redux-form container.

- Import the `reduxForm` function in the `RuleFormContainer.js` file.
- Wrap the `RuleForm` component using that function.
    - **Note:** the `RuleForm` component is already wrapped by `connect`. A component could be wrapped by multiple HOCs, that's perfectly fine in the container/component pattern. Redux even provides a `compose` function for that purpose.
- Integrate the Redux-form reducer when creating the store (we are already using `combineReducers`).
- In the `RuleForm` component, use the Redux-form `Field` component. This component needs 2 properties:
    - `name` that has to matches the key in the `rule` object.
    - `component` that renders the whole field. We can use the `RuleTitleField` and `RuleDescriptionField` components but they need a simple modification: we let Redux-form handle the value. Remove the input `value` attribute and add all the properties provided

by Redux-form in the property `input` (you can use the destructuring syntax `<textarea {...this.props.input} other props />`).

To prefill the fields when editing a rule, we just need to provides a `initialValues` property: in the `mapStateToProps` function of the `RuleFormContainer.js` file, add a property `initialValues` that contains the rule to edit.

Check that the form fields are prefilled when editing a rule.

Validation

We must respect this policy:

- Title:
  - Mandatory.
  - Up to 50 characters.
- Description is optional, but if filled:
  - At least 5 characters.
  - Up to 100 characters.

Each `Field` component accepts a `validate` property. In `RuleForm.js`, write the 2 `validate` functions (for the title and the description). They both should:

- Accept the value of the field as parameter
- Return an error message in case of invalid value, return nothing otherwise.

Then, provide these functions using the `validate` property to the `Field` component.

Now, it is time to give feedback to the user in case of wrong values. Errors are readable at the field level in real time:

- In the `RuleTitleField` and `RuleDescriptionField` components:
  - Use the `meta` property (provided by Redux-form as well as the `input` one) to display an error message.
  - **Note**: Bootstrap handles textual context for form fields: simply add a `span` with a `help-block` class sibling of the input/textarea.
  - Add the class `has-error` to the root div (which has the class `form-group`) in case of error.

Try to trigger the errors to check that the validation is working.

Submission

- Create two action creators in `rules-actions.js`: `addRule` and `updateRule` depending on the presence of an id. Both actions:

© Copyright 2019 Zenika. All rights reserved

    - Get the rule as argument
    - Perform the corresponding call to the backend (reminder: backend endpoints are detailed in Lab 5).
    - Dispatch the response from the server.
- In the `RuleForm` container, add a `mapDispatchToProps` function that exposes the two action creators to the underlying component.
- In the `RuleForm` component, define a `onSubmit` event on the form that call the property `handleSubmit` with the adequate action creator.

Notes: A lot of properties are set in motion in a single line. To be clear:

- `onSubmit` is a standard property from the React event API.
- `handleSubmit` is provided by the Redux-form wrapper `reduxForm`.
- `addRule` and `updateRule` are our action creators exposed through the Redux wrapper `connect`.
- Handle the resulting actions in the reducer:
    - The updated rule must be replaced in the current state.
    - The created rule must be appended to the current state.

Check that both rule creation and edition are working well.

We will add a last feature: even if Redux-form is smart enough to not submit the form in case of error, the submit button could be disabled for a better user experience:

- Add a `disabled` property on the submit button: this property must be activated only if the form is `invalid` or `submitting` or `pristine`, all these characteristics are available as properties at the form level.

© Copyright 2019 Zenika. All rights reserved

Lab 8 - Performance

In this lab, we are going to analyze the app with Chrome Dev tools and improve performance using the `shouldComponentUpdate` method.

- Open Chrome Dev tools, select "Performances" tab and click "Record"
- Reload the application.
- Watch "User timing" part in Chrome Dev tools report.

Avoid useless rendering

We will now use the tool to avoid useless rendering:

- In devtools, "Clear" previous results and click "Record"
- Click on one of the Like/Dislike button.
- Click "Stop" on devtools
- Watch "User timing" part in Chrome Dev tools report.
- What can you see ?
- Avoid useless rendering using the `shouldComponentUpdate` method or the `PureComponent` class.
- Try again...

-

© Copyright 2019 Zenika. All rights reserved

Annexes

ES2015 SYNTAX

Here are some example of ES2015 syntax that might be useful for labs.

This annex doesn't cover the whole ES2015 features (a lot of blog articles and official documentation may be more useful).

New keywords: `const` et `let`

Introduction of two new keywords to create variables:

- `const` to define constants (immutable reference).
- `let` to define variable with block scope (as a reminder `var` defines variable with function scope).

Classes

Easy class creation:

```
class Player {
    constructor(name) {
        this.name = name;
    }

    play() {
        console.log(this.name + ' is playing');
    }
}

let player = new Player('John Doe');
console.log(player.play());
```

To note:

- No need to use `function` keyword to define functions.
- The class constructor is a function named `constructor` (optional).

Can use inheritance with `extends` keyword:

```
class Person {
    constructor(name) {
        this.name = name;
    }
}

class Player extends Person {
    constructor(name) {
        super(name);
```

© Copyright 2019 Zenika. All rights reserved

```
    }

    play() {
        return this.name + ' is playing';
    }
}

let player = new Player('John Doe');
console.log(player.play());
```

ES2015 modules

An ES2015 module can be imported with `import` keyword and exported with `export` keyword.

A module can be exported by name (and so must be imported with that name):

```
// Fichier foo.js
export function foo() {
    return 'foo';
};
```

```
// Fichier bar.js
import { foo } from './foo';

console.log(foo()).
```

To ease imports, use the default export:

```
// Fichier player.js
export default class Player {
    play() {
        console.log('play !');
    }
}
```

```
// Fichier team.js
import Player from './player';

export default class Team {
    constructor() {
        this.players = [
            new Player('John Doe'),
            new Player('Jane Doe')
        ];
    }

    play() {
        this.players.forEach((player) => {
            player.play();
        });
    }
}
```

Note: It is possible to mix default export and named exports.

React & JSX tips

CSS classes manipulation

With React, we can manipulate directly CSS classes to apply on element with "inline" style:

```
import React from 'react';

export default class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    let hidden = this.props.hidden;
    return <div className={'class1 ' + (hidden ? 'hidden': 'visible')}></div>;
  }
}
```

To avoid the code to be hard to read and maintain, use `classnames` library (https://www.npmjs.com/package/classnames):

```
import React from 'react';
import classNames from 'classnames';

export default class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return <div className={classNames('class1', { hidden: hidden, visible: !hidden
  }
}
```

HTML5 form validation

Check form validity

HTML5 defines new attributes to check fields validity. One of the most used attribute is `required` to check a field has been filled by users.

For instance:

```
<form>
  <div class="form-group">
    <label for="input-title">Title:</label>
    <input type="text" placeholder="Title" required />
  </div>
</div>
```

It is easy to check field validity with plain JavaScript (by using the following function with `onBlur` event):

```
function validate() {
  let input = document.getElementById('input-title');
  console.log('Is valid: ', input.checkValidity());
}
```

React example

Use HTML5 API to check field validity is easy: we can directly access to native DOM element.

Example:

```
import React from 'react';

export default class MyForm extends React.Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() {
    this.validate();
  }

  render() {
    return (
      <form>
        <div className="form-group">
          <label htmlFor="input-title">Title:</label>
          <input type="text" ref="inputTitle" placeholder="Title"
                 required onBlur={this.validate.bind(this)} />
        </div>
      </div>
    );
  }

  validate() {
    console.log('Is valid: ', this.refs.inputTitle.checkValidity());
  }
}
```

Notes:

- Access to React element with `ref` property.
- Access to DOM element with `ReactDOM.findDOMNode` function (with React element as first parameter).

To validate the form when instantiating a component, call validation in `componentDidMount` function: React makes sure DOM is created and available.

© Copyright 2019 Zenika. All rights reserved