Formation React

Labs

zenika

ARCHITECTURE INFORMATIQUE

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).

Prerequisite

Preparation

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

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

The installation will be done through *Node.js* (which must be available on the workstation), available on https://nodejs.org. Make sure you have at least the latest LTS version installed 14.17.0).

You can make sure of it using the following command on your terminal:

```
node --version
```

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:

```
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.

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@3.1.1
```

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.

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';
```

Lab 2: Hooks

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:
  - Using the useState hook, create a `folded` state initialized with the value false by default.
  - Depending on the `folded` value, display or hide the description (tip: use the `hidden` CSS class to hide a DOM element. To conditionnaly add a class with React, have a look at the `classnames` library).
  - 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:

- When the user clicks on the title of a rule, reverse the value of the `folded` state.
- 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 (tips: you can use the useEffect hook for that).

"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 functional component `LikeBtn` 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.
- Using the useState hook, create a `counter` state value initialized to 0. This value will be updated by user actions (click on the button).
- Using the useEffect hook, make sure that if there is a props `counter`, it's value is inserted inside the `counter` state value.
- Using JSX, add the markup of the LikeBtn component.
- Increment the counter 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 to your component. Example: `Rule.propTypes = {}`
  - Define a type for each props used in the component.

Lab 3: Context

- Create a file `ThemeContext.js` and define a `ThemeContext` with a variable `theme` (default to undefined) and a function `updateTheme` (default to empty function).
- Create a `Header` component displaying a header with a theme button.
- Import the `ThemeContext` in the `Header` component, display `dark theme` on the button if the theme is `blue`, `blue theme` otherwise.
- Modify the `RuleList` component to display the `Header` component you juste created and provide the `ThemeContext`.
- Make sur the `theme` value in the `ThemeContext` is correctly updated everytime you click the button in your `Header` component.
- Change `bg-primary` to `bg-dark` and `panel-primary` to `panel-dark` according to the `theme` value.

You can add the following css to you `index.css` file to have a correctly defined dark theme. Feel free to update it according to your preferences.

```
main {
  padding: 1rem;
}

/* Dark theme */
.bg-dark {
  background-color: rgb(63, 61, 61);
}

.panel-dark {
  border-color: rgb(63, 61, 61);
}

.panel-dark .panel-heading {
  background-color: rgb(63, 61, 61);
  color: white;
}
```

Lab 4 - REST

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": "http://localhost:4000"
```

- Restart `create-react-app` .

Dependencies

For this lab, we will use `fetch` (see examples in slides and fetch documentation on MDN.

A bit of refactoring

Most of the time in real world React applications, we have one root component called `App` in which we're going to find the global setup of our app.

Create a `App` component. This component will

- encapsulate the `RuleList` component.

- be used in your `index.js` file as the root component.

Fetch rules

Now that we have a proper root component, we need to get rules from our server. For that, in your `App` component,

- create a `rules` state variable
- use `fetch` to ask our server for the list of rules.
- set your `rules` state variable with the received list of rules
- make sure your `RuleList` component does not use the rules from `data.json`
- verify your app still display the rules

Handle likes & dislikes

Now that we get the rules from our server, time to handle the likes/dislikes feature.

Inside the `LikeBtn` component

- Define an `incrementCounter` function which calls the `/rest/rules/:id/likes` or `/rest/rules/:id/dislikes` endpoint.
- Call `incrementCounter` on click
- Make sure the likes/dislikes functionality is still working.

Tips: you're going to need the rule id in your component.

Another bit of refactoring

You app should now be fully functional again, but using data coming from a server.

However, there is still a problem. You now have two sources of truth for the number of likes/dislikes of a rule. To see that, log the rules when you get them and log the counter value of your `LikeBtn` component when it is updated.

There is two differents numbers.

It may not be a big problem for now, but as the app keeps growing, it will very likely become a source of bug, especialy if this bad pattern keeps repeating.

To handle this, we will create one single source of truth for the rules in our app.

- Create a new context `RulesContext`. It's value is an object with a `rules` property (initialized to an empty array) and a setRules function (initialized to an empty function).
- In your root component, provide this context to the rest of your app, with the rules your already got.

- In your `LikeBtn` component, update the `rules` property in the `RulesContext` everytime you call the endpoint to increment the number of likes/dislikes.
- Clean up your component