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 functional component `RuleList` and export it by default.
- The component returns by default a empty `div` .
- The component receives a props called `rules` which will contains the list of rules.
- The component should iterate over each rule in `rules` and create a JSX block (equivalent to the HTML code you can find in `resources/rule.html`).
- The component should return each JSX block created 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 => <div>{item}</div>);
```

Then bootstrap the application:

- Update the file `src/index.js` to import and render the component `RuleList`
- Drag the file `data.json` from the `resources` directory into `src` and import it's data:

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

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

- Pass the `rules` data to the `RuleList` component as a props named `rules`

Your application should now display the list of rules.

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 function `Rule` and export it by default.
- The component receives a props called `rule` which will contains the the data of the rule to display.

Now, use this component:

- In the file `RuleList.js` , import the `Rule` component.
- Update your JSX to use the Rule component and it's props `rule` .

Your application should now still display the same list of rules.

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

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

The first feature is the ability for the user to fold and unfold a rule (to hide or display its description).

By default, for each rule the description will be displayed. Then, if our user click on the rule title, the description will be hidden. And if our user click again on the title, the description will be displayed again. And so on...

Fold and unfold

- Open the `Rule.js` file:
- Using the `useState` hook, create a `folded` state variable initialized with the value `false` by default.
- Identify the element displaying your rule title.
- Listen on the click event in the title element and when triggered inverse the value of your `folded` state variable.
- Verify your `folded` variable is correctly changing everytime you click on the title of a rule (you can `console.log` your `folded` variable for example)

At this point, you have a variable changing everytime time you click on the title of a rule. Now it's time to actually change what we display in our app according to this variable value.

- Identify in your rule template what element displays the rule description.
- On this element, add the `hidden` class if the `folded` state variable is `true`.

The `hidden` class is a Bootstrap CSS class hidding the element it is placed on. There are many ways in React to add or remove a CSS class according to the value of a variable. One common way to do that is to use the `classnames` library.

```
<div class={classNames('myClass', { 'hidden': myVar })>
```

On this example, `myClass` CSS class is always present on my element, and `hidden` is added only if `myVar` is `true`. If `myVar` is `false`, the `hidden` class is automatically remove. For more informations on how to use this library, go check its documention online (`npmjs.com` is a good starting point for that).

This library is not installed by default in your project. If you want to install this library, you should use `npm` the same way you used it to install `bootstrap` previously.

- Verify your application correctly hide/display the description of a rule everytime you click on its title.

The last thing we will do to finish this feature is to change the icon we display on the top right of the rule title if the description is visible or not.

- Identify which element display your icon.
- If the `folded` state variable is `false`, add the CSS class `glyphicon-chevron-down`.
- If the `folded` state variable is `true`, add the CSS class `glyphicon-chevron-up`.
- Verify your icon is correctly changing when you click on a rule title.

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

Likes and dislikes

The second feature we will add is the "likes" and "dislikes" feature.

- In your `src` folder, create a new file named `LikeBtn.js`.
- Inside this file, create a functional component `LikeBtn` and export it by default.

Both buttons are almost equivalent (visually and semantically), so we will use the same component for the "like" and "dislike" button.

- To indicate which button we should display in our `LikeBtn` component, add a `props` called `type` which can have the value `like` or `dislike`.
- If the props `type` has the `like` value, make sure your component displays the "like" button.
- If the props `type` has the `dislike` value, make sure your component displays the "dislike" button.
- For now, you can just put `0` inside your button (and keep the icon), we will update it later.

Now that your `LikeBtn` component is defined, we should use it.

- Go to your `Rule` component.
- In this component, use your new `LikeBtn` to display the "like" and "dislike" button of your rule.
- Verify your application correctly displays the "likes" and "dislikes" button.

Now that you're using a component for your buttons, it's time to make sure you can actually increment the number of "likes" and "dislikes".

- Go to your `LikeBtn` component.
- Using the `useState` hook, create a `counter` state variable initialized to `0`.
- Instead of always displaying `0` in your template, display the `counter` variable.
- Increment your `counter` value everytime your user click on the button.
- Verify you can now increment the number of likes and dislikes on your rules.

Note: for now, counters values start from 0 after each page refresh: there is no persistence mechanism for the moment.

Props validation

As the application is growing, it is interesting to use the React prop-types validation feature. To do so, in React we use a library called `prop-types`.

```
import { number, oneOf } from 'prop-types'

MyComponent.propTypes = {
  id: number,
  name: oneOf(['Riri', 'Fifi', 'Loulou'])
}
function MyComponent(props) {
  ...
}
```

In this example, I validate that `MyComponent` can receive a props called id and its value is a number. And a props called `name` and its value is either `Riri`, `Fifi` or `Loulou`. If one of this statement is not respected, there will be an error trigerred in the console of my browser.

You can check the document of the library to get more infos.

- Install the `prop-types` library in your project.
- Go to the `LikeBtn` component
- Use `prop-types` to validate the `LikeBtn` component receives in its props `type` only the value `like` or `dislike`.

Bonus: Add props validation for each component you created in your app.

Document title

To finish, we will change the title of our application. To do so, we will use the `useEffect` hook we studied.

To change the title of our app in Javascript, we modify the property `title` of the object `document`:

```
document.title = 'Mon super titre'
```

- Go to the `RuleList` component
- Using the `useEffect` hook, change the title of the document to `{number} rules`. Where `{number}` is actually the number of rules your application display.

Lab 3 - Http requests

The goal of this lab is to use the REST API provided our list of rules and interact with it, like a real world React application.

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
```

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.

- In your `src` folder, create a new file called `App.js`
- In this file, create and export a new component called `App`
- Make sure this new component uses the `RuleList` component in it's template.
- Make sure to pass to your `RuleList` component the list of rules in `data.json` .
- Open your `index.js` file
- Find the `ReactDOM.render` line and render the `App` component instead of the `RuleList` component.

- Verify your application still works properly

Fetch rules from our server

So at this point you have a new root component in your app, called `App`. This new component displays your other component `RuleList`. But `RuleList` still receives its list of rules from our local file `data.json`. We want to get this list from our server now.

- Go to your `App` component
- Remove everything in this file related to `data.json`
- Create a new state variable called `rulesData`
- Use `fetch` to ask our server for the list of rules (remember, you should make sure you only request your rules once, otherwise your application might crash).
- Convert the data your received from your server to json, and put it inside your `rulesData` state variable.
- Pass this `rulesData` state variable to your `RuleList` component.
- Verify your app still display the rules

Likes & dislikes

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

Increment counter

- Go to your `LikeBtn` component
- Inside your component, define an `incrementCounter` function which calls the endpoint POST `/rest/rules/:id/likes`.

To perform a POST request with fetch, we can use the second parameter of `fetch` to specify the HTTP verb we want to use.

```
{
  method: 'POST'
}
```

As you can see in the endpoint, you need the id of the rule to inform our server on which rules the likes should be incremented.

- Define a new props in your `LikeBtn` component called `id`.
- Define a new parameter in your `incrementCounter` function called `id`.
- Make sure to use this parameter in your fetch call.

- Go to your `Rule` component, and make sure to pass the current rule id to your `LikeBtn` components.

With this, you should have your function `incrementCounter` ready to call your server to increment the number of likes on a rule. Time to call it.

- Call `incrementCounter` on click

At this point, if you click on one of your like button, nothing changes on your app. It's because even though you tell your server to increment the likes of your rule, your React application is not aware something changed on your data. In short, after a click you have different data on your server and on your React app.

To fix it, there is may ways. One way is to wait for our call to our server to finish, and then increment a variable to make sure we have the same data both on our server and our React app.

Luckily for you, you already such variable in your `LikeBtn` component, it's your `counter` state variable.

- Update the `incrementCounter` function to increment your `counter` variable when your call is finished.
- Verify that this time, clicking on a like button actually changes the number of likes you see on a rule.

Likes or dislikes ?

You might have noticed there is still something strange with our app. If you click on a dislike button, it's actually the number of likes that changes. It's because in our call, we always call the same endpoint `/rest/rules/:id/likes`. If we want to increment the number of dislikes, we should call `/rest/rules/:id/dislikes`.

- According to the type of your `LikeBtn` component, call the correct endpoint to increment the number of likes or dislikes.

Counter initial value

There is still one problem left in our app.. Increment the number of likes of a rule, and then refresh your page. What do you see ? You lost the number you had a second ago. After all this effort to stock our data in our server, it's too bad !

The problem lies in the `LikeBtn` component and the `counter` variable. When we define this variable with `useState`, we pass an initial value to 0.

So when we create our `LikeBtn` components (when our app appeared for the first time), we have a counter value to 0, which translates to "we have 0 likes and dislikes for every rule we display". Which is now wrong. Because now we stock our data on our server, and so some of the rules we receive in our React appp already have some likes or dislikes.

- Add a `initialCounter` props to your `LikeBtn` component. This props will have the number of likes/dislikes of the rule you display and you will use this as a default value for your `counter` variable.

Congratulations, you now handle your rules exactly like in a real world React applications ! In the futures apps you will be working on, the data might be different, the components too, but the principles of what you just did will always be the same.

Create a new rule

To finish, we will add a new feature in our app: the possibility to add a new rule.

- Go to your `App` component
- In your template, add the following JSX:

```
<h3>Add a new rule</h3>
<label>title<input type="text" name="title"></label>
<label>description<input type="text" name="description"></label>
<button type="submit">Submit</button>
```

- Create two new state variables: `titleData` and `descriptionData`

To get what our user writes in an input, we can listen to the `input` event on it. This event holds data which are automatically passed to our handler function - most of the time, we call this data `event`. Inside this data, you can find what is written in the input in `event.target.value`.

```
<input onInput={event => console.log(event.target.value)}>
```

- Pass what our user writes in our `title` input to the `titleData`
- Pass what our user writes in our `description` input to the `descriptionData`
- Create a function `addRule` which calls the endpoint `POST` `/rest/rules` with and object holding the data in `titleData` and `descriptionData`:

```
{
  title: titleData,
  description: descriptionData
}
```

When performing a POST request we can pass data to it. This data will be received by our server so it can do something with it. We can use the second parameter of the `fetch` function to specify data we want to send.

```
{
  body: myData
}
```

- Call `addRule` when our user click on the submit button.

At this point, if you try to create a new rule, nothing changes. But if you refresh your page, you should see the new rule you just created.

Like ealier with the likes/dislikes buttons, the problem lies in the fact we have different data in our server and in our React app after we create a new rule.

- After the call to create a new rule is finished, perform a new request to get all the rules from our server again and pass the data to your `rules` variable.

Congratulations, you're now capable of creating new rules through your app !

Lab 4: 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.

```css
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;
}
```