Formation React + Redux + Env

# Mise en place du cycle Redux

Maintenant que nous avons la base de notre application ainsi qu’un component fonctionnel, nous allons créer notre cycle Redux.

Voici pour rappel ce que nous disions concernant Redux dans la partie précédente :

Redux est composé en différentes parties, chacune ayant son rôle distinct. Le tout fonctionne ensemble pour faire circuler l’information, de la saisie utilisateur jusqu’au Store, un simple objet au format JSON.

Commençons tout de suite par définir notre arborescence. Les différentes parties de Redux sont :

Actions

Store

Components & Containers

Reducers

Une Action est déclenchée par un utilisateur. Celle-ci est envoyée au Reducer qui va traiter l’action et envoyer le résultat au Store. Ce store est distribué aux Component/Containers afin d’afficher les données à l’écran via le point d’entrée (c.f. step-1).

Commençons tout de suite à mettre en place le cycle Redux.

#### Actions

Puisque nous avons déjà notre arborescence, allez dans le dossier « Action » et créez-y un fichier nommé « ActionsTypes.js » comme suit :

export const SAY\_HELLO = "SAY\_HELLO";

Les ActionsTypes, ou types d’actions, ce sont seulement les noms qui vous permettrons d’identifier vos actions dans votre cycle Redux. Créons une action pour mieux comprendre.

Toujours dans le même dossier « Actions », créez un fichier « HelloActions.js » avec le code suivant :

import \* as types from './ActionsTypes';

const sayHello = hello => ({

type: types.SAY\_HELLO,

hello,

});

export default sayHello;

Si vous avez bien regardé précédemment, nous avons exporté la constante de notre fichier ActionsTypes, nous pouvons donc toutes les récupérer ici grâce à l’écriture « import \* as » suivi d’un nom de variable de votre choix. Nous utilisons ensuite cette variable pour renseigner notre action.

Une action, c’est un objet qui sera envoyé au reducer et qui contient **au** **minimum une clef** nommée conventionnellement « type ».

Notre action « sayHello » contient une autre clef « hello », qui est une variable. Il s’agit d’une syntaxe ES2015. La clef sera créée automatiquement, avec pour valeur le contenu de la variable. Ici, il s’agit du paramètre passé à notre action (Vous trouverez cette clef sous le nom de « payload » lors de vos recherche. J’ai choisi de nommer différemment les clefs afin que ce soit plus clair).

#### Reducers

Allez maintenant dans le dossier « Reducers » et créez-y un fichier « HelloReducer.js » comme suit :

import \* as types from '../Actions/ActionsTypes';

export default function (state = [], action) {

switch (action.type) {

case types.SAY\_HELLO:

return action.hello;

default:

return state;

}

}

Ici aussi nous importons les ActionsTypes. Nous pourrions réécrire le nom de l’action, mais cette méthode permet d’éviter les erreurs de frappe par exemple.

Notre reducer est une fonction. Elle prend en paramètre le state (notre store) ainsi que l’action qui lui a été envoyée. Son but est simple : exécuter un bout de code en fonction du type de l’action. Pour cela nous utilisons un switch…case. Ici, nous retournons simplement la valeur de la clef « hello » de notre action.

**Information :**

En ReactJS, chaque component possède un state propre à lui-même, il est parfois nécessaire de l’utiliser pour gérer des cas particuliers, dans les formulaires par exemple. Le Store, quant à lui, est le state général de l’application, partagé à tous les components (Nous verrons plus tard comment).

Notez que nous exportons directement une fonction anonyme. En effet, un nom n’est pas nécessaire puisqu’il n’y a que cette fonction à l’intérieur (Ce n’est pas toujours le cas, mais ce sont des exceptions, ou des erreurs d’organisation).

Créez en suite un fichier « index.js » dans le dossier « Reducers » comme suit :

import { combineReducers } from 'redux';

import HelloReducer from '../Reducers/HelloReducer';

const allReducers = combineReducers({

hello: HelloReducer,

});

export default allReducers;

Nous importons ici « combineReducer », une méthode du package « redux » permettant de fusionner plusieurs petits reducers en un seul gros afin de l’exporter. Nous créons par la même occasion une clef nommée « hello » que nous réutiliserons plus tard. Retenez là !

#### Store

Créons maintenant notre store. Ajoutez un fichier « index.js » dans le dossier « Store » avec le code suivant :

import { createStore } from 'redux';

import allReducers from '../Reducers';

import sayHello from '../Actions/HelloActions';

const store = createStore(

allReducers,

);

store.dispatch(sayHello("Hey ! Hello buddy !"));

export default store;

Nous importons « createStore », une méthode du package « redux » nous permettant de… créer notre store. Le store, c’est un objet JSON contenant toutes les informations de notre application. createStore va récupérer nos reducers, faire sa tambouille et nous sortir un jolie objet JSON.

Nous avons donc importé notre reducer et l’avons ajouté à notre store. « createStore » prend aussi en compte les middleware à ajouter à votre application, comme « redux-thunk » permettant les requêtes asynchrones. Nous y reviendrons plus tard.

Nous avons aussi importé notre action « sayHello » créée précédemment. Nous l’utilisons en tant que paramètre à la méthode « dispatch » de notre store. **C’est cet appel à « dispatch » qui permet déclencher une action**, l’envoyer aux reducers etc. jusqu’à la vue. (Nous verrons l’évolution du component « Hello » plus bas. Il prend maintenant un paramètre).

Le store est stocké dans une constante « store » que nous exportons à la fin.

Ce store, nous allons le diffuser dans toute l’application. Modifions un peu notre point d’entré « index.jsx » (à la racine du dossier « js » :

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import Store from './Store';

import App from './AppWrapper/App';

ReactDOM.render(

<Provider store={Store}>

<App />

</Provider>,

document.querySelector('#root'),

);

Notez l’arrivée de l’import de « Provider » . « Provider », c’est un simple component qui va encadrer toute notre application et qui prendra notre store en paramètre afin de le diffuser dans tous nos components... et notre application, c’est « App » ! Notre component principal que nous avons créé dans l’étape rpécédente.

Le component « App » est l’enfant de « Provider ». « Provider » s’écrit donc comme une balise ouvrante/fermante, et non comme une auto fermante (c.f. le component « <App /> »).

#### Component principal

Nous allons faire évoluer notre component « Hello ». Il sera lié à un container, il est donc **obligatoire** d’importer le container, et non plus le component :

import React from 'react';

import Hello from '../../Containers/Hello';

const App = () => <Hello />;

export default App;

**C’est une erreur courante** alors faite attention !

#### Component

Le component « Hello » a un peu changé. Il désire maintenant afficher ce qui lui est fourni. Le voici :

import React from 'react';

import PropTypes from 'prop-types';

const Hello = ({ hello }) => <h1>{hello}</h1>;

Hello.propTypes = {

hello: PropTypes.string.isRequired,

};

export default Hello;

Nous constatons un nouvel import : « PropTypes ». Pour faire simple, chaque component possède un objet de propriété nommé « Props ». Les clés de cet objet sont les « prop ». Il s’agit des informations envoyées à notre component. Ici, nous déclarons une **prop** nommée « hello », de type string et obligatoirement renseignée (« isRequired »).

La déclaration des PropTypes n’est pas obligatoire, mais **fortement recommandé**. Même si cela peut paraitre ennuyant, les outils de debug en React ne sont pas encore assez performants pour se permettre d’omettre les PropTypes. Il est nécessaire de savoir ce que votre prop doit contenir, cela vous donnera déjà une indication concernant l’erreur (En cas d’erreur de « type object, required ‘’array’’ » et inversement, pensez à vérifier le retour de vos reducers !).

Concernant le component, il prend maintenant en paramètre notre prop « hello » et celle-ci est affichée à la place du précédent texte.

C’est bien beau tout ça, mais d’où provient-elle cette prop « hello » ? Qui la lui a fournie ? Et bien il s‘agit d’un container !

#### Container

C’est lui qui va fournir à notre component ce que nous voulons lui donner. Créez un fichier « hello.js » dans le dossier « Containers » avec le code suivant :

import { connect } from 'react-redux';

import Hello from '../Components/Hello';

const mapStateToProps = state => ({

hello: state.hello,

});

const matchDispatchToProps = () => {};

export default connect(mapStateToProps, matchDispatchToProps)(Hello);

Un container contient tous se qui attrait à Redux. En général, il ne possèdera que deux méthodes :

* « mapStateToProps » : Cette méthode qui prend en paramètre state (notre store) permet de définir les props de notre component. Ici, nous retrouvons « hello », qui prend comme valeur la clé « hello » de notre store. Les noms n’importent peu, seule la clé de mapStateToProps et la prop définie dans le component doivent être identique.
* « matchDispatchToProps » : Cette méthode permet de connecter nos actions à notre component. Ici, nous n’avons rien puisque nous appelons notre action directement dans le store via « store.dispatch() ».

Nous exportons ensuite la méthode « connect » précédemment importée. Elle prend en paramètre nos méthodes, « mapStateToProps » et « mathDispatchToProps ». Nous précisons à la suite le component auquel nous souhaitons rattacher ces props. Ici, c’est « Hello », nous l’avons aussi importé (Le component, évidemment !).

Le cycle est presque terminé, résumons :

Le store est créé. A ce moment, il dispatch une action. Elle est envoyée au reducer qui modifie le store en conséquence. Ce store contient maintenant une clé « hello » qui est repérée par notre container qui se charge de l’envoyer à notre component qui est affiché à l’écran.

A ce stade, vous pouvez déjà constater que notre titre prend comme valeur ce que nous avons écrit dans notre « store.dispatch() » !

Cette action est dispatchée au démarrage de l’application, donc une seule fois. Voyons comment modifier le titre de notre page grâce à un champ de texte et une action. Pour ce faire, nous allons réutiliser la même action, il n’est pas nécessaire d’en créer une autre puisque celle-ci fait déjà ce que nous souhaitons !

Ajoutons donc cette action aux prop de notre component Hello. Modifiez le container « Hello » comme suit :

import { connect } from 'react-redux';

import { bindActionCreators } from 'redux';

import sayHello from '../Actions/HelloActions';

import Hello from '../Components/Hello';

const mapStateToProps = state => ({

hello: state.hello,

});

const matchDispatchToProps = dispatch => bindActionCreators({ sayHello }, dispatch);

export default connect(mapStateToProps, matchDispatchToProps)(Hello);

Qu’avons-nous ajouté :

* « bindActionCreator » : une méthode importée du package ‘redux’ afin de déclencher le dispatch de notre méthode. De ce fait, nul besoin de le faire nous-même. **Attention : notez que bindActionCreator ne fonctionne pas pour les méthodes asynchrones** (Ou alors je n’ai pas trouvé comment faire). Nous en créerons une plus tard, pas d’inquiétude !
* « sayHello » : Notre action à faire passer au component
* « matchDispatchToProps » : cette méthode renvoi maintenant quelque chose. Nous lui passons dispatch en paramètre et appelons bindActionCreator afin qu’il fasse le nécessaire tout seul pour que notre action soit dispatchée quand elle est appelée. Cette méthode est ajoutée en tant que second paramètre de notre méthode connect() (Si vous n’avez pas de méthode matchDispatchToProps, vous devez écrire « null », tout simplement).

Maintenant, nous devons modifier notre component. Le voici :

import React from 'react';

import PropTypes from 'prop-types';

class Hello extends React.Component {

constructor() {

super();

this.onChange = this.onChange.bind(this);

this.trigger = this.trigger.bind(this);

this.state = {

value: '',

};

}

onChange(e) {

this.setState({ value: e.target.value }, this.trigger);

}

trigger() {

this.props.sayHello(this.state.value);

}

render() {

return (

<div>

<h1>{this.props.hello}</h1>

<input type="text" onChange={this.onChange} value={this.state.value} />

</div>

);

}

}

Hello.propTypes = {

sayHello: PropTypes.func.isRequired,

hello: PropTypes.string.isRequired,

};

export default Hello;

Notre component est passé d’une petite fonction pure, sans logique, à une classe ! Décrivons là :

* Les « imports » ne changent pas.
* « const Hello = () => {…} s’est transformé en « class Hello extends React.Component {…} »
* Nous retrouvons à l’intérieur un constructeur, deux méthodes et la méthode « render », cette dernière servant à l’affichage de notre component.

Le constructeur contient :

* « super » n’est à appeler que lorsque vous avez besoin d’un constructeur comme ici, et vous **DEVEZ** l’appeler s’il s’agit d’une sous-classe. « super() » est nécessaire car c’est lui qui initialise « this ». Si vous regardez bien, notre classe **hérite** de « React.Component ».
* « This.onChange = … » et « this.trigger = … » permettent de connecter nos méthodes et les événements qui leur sont liés au contexte auquel le component appartient. « onchange » est appelée à chaque changement dans le champ de texte, « trigger » est là pour déclencher notre action.
* « this.state » est un objet gérant le statut de notre input. Grâce à ça, nous avons le contrôle dessus.

Notre méthode « onChange » est déclenchée par l’événement « onChange » de l’input. Elle contient un appel à « this.setState » qui permet de modifier la valeur de notre objet « state » définie précédemment dans le constructeur.

La méthode « trigger » est appelée en tant que **callback** de la méthode « setState ». « trigger » contenant l’appel à l’action, celui-ci ne sera exécuté qu’après le changement du statut. C’est important afin de s’assurer que l’on insert les bonnes données dans notre store.

Nous avons également ajouté notre action dans les propTypes, parce-que, voilà, debugging etc.

C’est terminé. En lançant l’application, vous pourrez modifier le titre grâce à l’input.