Formation React + Redux + Env

# Mise en place du cycle Redux

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).

Une interrogation importante subsiste néanmoins :

**Qu’est-ce qu’un component ?**

Il s’agit simplement d’un élément ou un groupe d’élément afficher à l’écran. Par exemple un component « social » contenant les différents boutons de réseaux sociaux (Twitter, Facebook, etc.). L’application React se compose la plupart du temps un component principal, qui remplacera ici notre fameux « Hello World », et c’est dans ce dernier que les autres components seront utilisés.

Concernant les autres parties de Redux, elles seront décrites plus tard.

Maintenant, créons les dossiers correspondants dans « dev/js/ », là où se trouve déjà notre point d’entrée « index.jsx » (Créez bien un dossier « Components » et un autre « Containers », les deux fonctionnent ensembles mais doivent être distinct pour plus de lisibilité).

En matière d’organisation, nous allons créer un dossier « AppWrapper » et un dossier « Api ». « AppWrapper » va contenir notre component principal à afficher à l’écran et « Api » contiendra nos **appels asynchrones vers une api extérieure**.

Bien, maintenant, codons !

### Component React

Nous allons créer notre premier component et l’afficher à l’écran. Pour ce faire, créez un dossier « Hello » dans le dossier « Components » et placez-y un fichier nommé « index.jsx » avec le contenu suivant :

import React from 'react';

const Hello = () => <h1>Hello World From Component !</h1>;

export default Hello;

Il s’agit d’une toute simple fonction écrire en ES2015 et retournant un virtual DOM, ce fameux DOM créé par Facebook pour React. Ce virtual DOM ressemble énormément à du HTML alors vous ne serez pas perdu. Nous exportons en suite notre composant afin de l’inclure… dans le point d’entrée !

Voici donc notre point d’entrée modifié :

import React from 'react';

import ReactDOM from 'react-dom';

import Hello from './Components/Hello';

ReactDOM.render(

<Hello />,

document.querySelector('#root'),

);

Notez l’import du component « Hello » et son utilisation en tant que balise, comme en HTML encore une fois !

Si vous regardez bien, je n’importe pas le fichier « index.jsx » directement, mais seulement le dossier « Hello ». WebPack s’occupe tout seul de m’importer mon fichier. Si le nom du fichier est différent, alors il faudra importer le fichier. Ici aussi, l’extension ne sera pas obligatoire car résolue automatiquement.

« Comment cela ce fesse ? » me direz-vous ? J’ai modifié très rapidement notre fichier de configuration « webpack.config.js » comme suit :

output: {

// ...

},

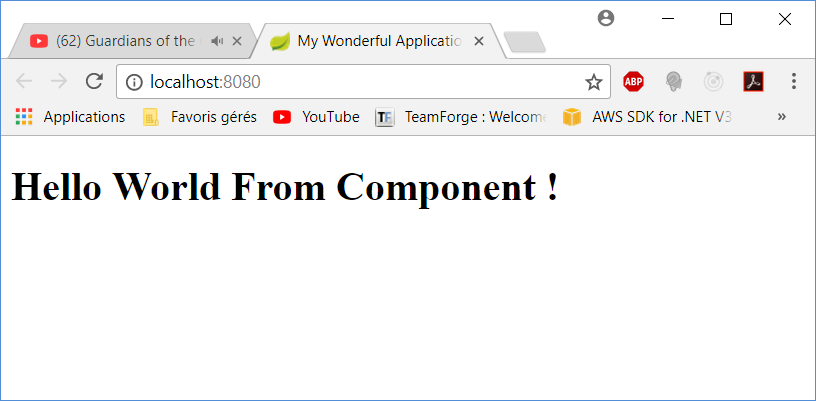
resolve: {

extensions: ['\*', '.js', '.jsx'],

},

« resolve » permet à Webpack de résoudre certaines extensions tout seul.

Lancez votre application et vous devriez constater l’affichage suivant :



Nous avons donc correctement implémenté notre premier component React ! Plutôt simple n’est-ce pas ? Mais insuffisant évidemment… passons à Redux. \*Tin tin tiiiiiiiinnnn !! \*

### Création des fichiers

Nous allons créer notre 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.

#### 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.

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 permettrant 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.

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 les arrivées des imports de « Provider » et de « App ». « Provider » c’est un simple component qui va encadrer toute notre application et qui prendra notre store en paramètre afin de le diffuser dans toute notre application... et notre application, c’est « App » ! Notre component principal que nous allons créer juste après.

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 « <Hello /> » précédemment).

#### Component principal

Voici donc le component « App ». Créez un dossier « App » dans le dossier « AppWrapper » et ajoutez-y un fichier « index.jsx » :

import React from 'react';

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

const App = () => {

return (

<Hello />

);

};

export default App;

C’est un component tout simple qui ne retourne pour le moment que notre component « Hello ». J’ai prévu les accolades et le return pour plus tard, mais dans le cas où vous n’avez qu’une seule chose à retourner, la syntaxe suivante est référable :

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

ESLint vous conseillera d’ailleurs de meilleures écritures si vous êtes sur Visual Studio Code.

Nous affichons donc notre component « App » qui, lui, affiche notre component « Hello ». Si vous avez bien regardé, le chemin d’accès à notre component n’est plus le même ! En effet, il est **associé à un container**. Lorsque c’est le cas, **c’est bien le container qu’il faut importer** ! **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 parraitre 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 indiquation 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 palce 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, seul 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 ratacher ces props. Ici, c’est « Hello », nous l’avons aussi importé.

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 seond paramètre de ntore méthode connect().

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.

### Action asynchrone

Avant de terminer, nous allons créer une action asynchrone car cela est un tout petit peu différent des actions communes. Récupérons une liste de film sur une API extérieure par exemple.

Pour ce faire, créez un fichier « index.js » dans le dossier « api » et insérez-y le code suivant :

import fetch from 'isomorphic-fetch';

export const getMovies = () => fetch('https://facebook.github.io/react-native/movies.json')

.then(

response => response.json(),

error => console.log('An error occured : ', error),

);

Ici, nous importons « isomorphic-fetch », le package nous permettant d’utiliser l’API fetch pour faire des appels API.

Nous avons ensuite une méthode, « getMovies », retournant la liste des films au format json, ou loggant un emssage en console en cas d’erreur.

Installons donc le package isomorphic-fetch et profitons-en pour installer « redux-thunk », un package permettant les appels ajax asynchrones :

npm install --save isomorphic-fetch redux-thunk

Nous avons besoin d’une actionsTypes et d’un ActionCreator afin de faire cet appel dans les règles de l’art.

L’actionsTypes :

export const GET\_MOVIES = "GET\_MOVIES";

L’action va être séparée en deux. Premièrement, l’actionCreator, puis l’appel à l’API ainsi que le dispatch de l’actionCreator :

export const getMoviesSuccess = movies => ({

type: types.GET\_MOVIES,

movies,

});

export const getMovies = () => dispatch =>

api.getMovies()

.then(json => dispatch(getMoviesSuccess(json)),

);

N’oubliez pas d’importer votre api :

import \* as api from '../Api';

La bonne pratique voudrait que l’ont créé un fichier « moviesAction.js » pour cette action, mais dans un souci de flemme, je mets tout dans « helloAction.js ».

Nous avons donc « getMoviesSuccess » qui est notre actionCreator, et « getMovies » notre méthode qui va appeler l’API, récupérer les données et seulement ensuite dispatcher notre objet json.

Cette action, il faut la gérer, et c’est le reducer qui s’en occupe. Nous créons donc un fichier « MoviesReducer.js » et y créons un reducer :

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

export default function (state = {}, action) {

switch (action.type) {

case types.GET\_MOVIES:

return action.movies;

default:

return state;

}

}

(Attention, nous initialisonspar défaut state en objet cette fois, car l’API que nous appelons nous renvoi un objet !)

Nous ajoutons ce reducer dans notre fichier « index.js » du dossier « reducer » avec la clef « movies » :

const allReducers = combineReducers({

hello: HelloReducer,

movies: MoviesReducer,

});

Enfin, et c’est le plus important ici, nous allons ajouter un middleware nommé « redux-thunk » nous permettant d’effectuer des appels ajax asynchrones. Grâce à ça, le retour des appels aux APIs externes seront attendu avant que le code suivant s’exécute. Rendez-vous dans le store et importez-y « redux-thunk » :

import thunk from 'redux-thunk';

Il faut aussi importer la méthode « applyMiddleware » du package « redux » :

import { createStore, applyMiddleware } from 'redux';

Et utilisez ces éléments comme suit :

const store = createStore(

allReducers,

applyMiddleware(thunk),

);

On appelle la méthode « applyMiddleware() » à l’intérieur de « createStore() ». Elle prend en paramètre un ou plusieurs middlewares, selon votre bon vouloir.

Il ne reste plus qu’à appeler notre action dans notre component. Pour cela, rien ne change par rapport aux actions synchrones : nous ajoutons l’action au component via le container et nous l’appelons comme une fonction dans le component. Nous n’oublierons pas les PropTypes ainsi que l’affichage.

Voici le container :

import { connect } from 'react-redux';

import { bindActionCreators } from 'redux';

import { sayHello, getMovies } from '../Actions/HelloActions';

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

const mapStateToProps = state => ({

hello: state.hello,

movies: state.movies,

});

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

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

Constatez l’ajout de « getMovies » dans les imports et dans la méthode binsActionCreators ainsi que la prop « movies » dans « mapStateToProps ».

Pour el component, on va ajouter une méthode « loadMovies() » :

loadMovies() {

this.props.getMovies();

}

Elle se contentera d’appeler notre action « getMovies ». Elle sera déclenchée par un bouton. Comme j’ai la flemme de faire l’affichage, ej vais simplement afficher le résultat de notre requête en console. Voici donc notre méthode « render() » :

render() {

console.log(this.props.movies);

return (

<div>

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

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

<br /><br />

<button type="button" onClick={this.loadMovies}>Load movies</button>

</div>

);

}

Et enfin, le plus important, les PropTypes :

Hello.propTypes = {

sayHello: PropTypes.func.isRequired,

getMovies: PropTypes.func.isRequired,

hello: PropTypes.string.isRequired,

movies: PropTypes.shape({

title: PropTypes.string,

description: PropTypes.string,

movies: PropTypes.arrayOf(

PropTypes.objectOf(

PropTypes.string,

),

),

}).isRequired,

};

Hello.defaultProps = {

movies: {},

};

Remarquez l’ajout de « getMovies » ainsi que « movies ». Ce dernier est décrit assez précisément afin de pouvoir mieux comprendre les messages d’erreurs encore une fois. En effet, il faut jouer avec les « isRequired » et les « defaultProps » (nécessaires lorsque « isRequired » n’est pas définie) afin de bien détailler l’objet que vous devez récupérer. Ça vous permettra de savoir si oui ou non vous connaissez bien ce que l’API doit vous renvoyer. Aussi, ça aide avec l’auto complétion du code lorsque vous faites un affichage correct (pas comme moi donc).

Notez que l’utilisation des objets n’est en rien différente d’un autre langage : vous taperez donc « movies.title » pour afficher le titre par exemple.

Le cycle de notre action asynchrone est donc maintenant terminé ! Si vous exécutez votre application, vous pourrez constater dans la console de développement (F12) que le clic sur le bouton vous permet d’obtenir les informations concernant la requête à l’API :



Aussi, vous pouvez voir que notre application nous retourne un objet vide « {} » lors du chargement du component. Il s’agit là de la valeur par défaut que nous avons donné dans notre reducer ! Puisque nous n’exécutons pas l’action au chargement du component, la prop « movies » prend la valeur par défaut attribuée dans le reducer. S’il n’y avait pas ça, alors vous auriez eu une erreur dans votre console.

Ceci marque la fin de ce chapitre. Voilà.