Formation React + Redux + Env

# Installation de l’environnement

Créer un repository git sur http://github.com (connectez-vous sur votre compte).

Ouvrez une console git à l’emplacement ou vous souhaitez copier votre repository (appelé repo à l’avenir) puis tapez la commande suivante :

git clone https://github.com/[username]/[repository\_name].git

Avant de commencer quoique ce soit, il est nécessaire d’initialiser notre branche master. On va donc créer un fichier readme.md et le push sur le repo distant (sur les serveurs Github/GitLab donc) :

Cd formation-react-redux-env

Copy NUL readme.md

Git add \*

Git commit –am ‘’ init branch master ‘’

Git push –u origin master

Que faisons-nous ici :

* Nous nous déplaçons dans le dossier du repo
* Nous créons un fichier vide nommé « readme.md »
* Nous passons nos modifications de « working directory » à « staging area ». Il s’agit là d’état dans lesquels se trouvent vos fichiers modifiés ([Schema](https://git-scm.com/images/about/index1@2x.png))
* Nous passons nos modifications en « repository » en ajoutant un commentaire sur les modifications effectuées
* Enfin, nous envoyons nos modifications au serveur distant.

Maintenant que ceci est fait, nous allons créer un tutoriel étape par étape via git. Chaque étape sera représentée sous la forme d’une branche.

Allons dans le dossier du repo pour créer notre branche « step-1 » :

Git branch step-1

Git checkout step-1

Initialisons notre projet :

Npm init

Valider toutes les étapes en appuyant sur « entrer ». Vous pouvez renseigner les informations si cela vous chante… Notez la creation d‘un fichier package.json dans votre dossier.

Installons maintenant quelques dépendances nécessaires pour un projet ReactJS/Redux, fonctionnant avec SASS, babel pour l’ES2015, ESLint pour un peu de propreté de code, et enfin Webpack. Nous verrons la config de ce dernier juste après.

Npm install --save react react-dom react-redux redux prop-types babel-core babel-preset-es2015 babel-preset-stage-0 babel-preset-react babel-core node-sass sass-loader css-loader style-loader

Npm install --save-dev eslint@3.19.0 eslint-config-airbnb@15.1.0 eslint-plugin-import eslint-plugin-jsx-a11y@5.1.1 eslint-plugin-react webpack babel-core babel-preset-es2015 babel-preset-stage-0 babel-preset-react babel-core babel-loader node-sass sass-loader css-loader style-loader

Plutôt que d’aller chercher à droite à gauche les packages dont vous avez besoin pour débuter, copier/coller simplement cette ligne dans la console et attendez que l’installation se termine. Vous noterez ensuite la présence de tous les packages dans le fichier « package.json ».

**Information :**

En cas de problème de téléchargement du package node-sass, installez-le package gulp-sass (qui lui contient, entre autres, le package node-sass) via la console :

Npm install –save gulp-sass

Vous pouvez également éditer le fichier package.json et changer le package node-sass par gulp-sass et en ajoutant la version voulue :

"dependencies": {

//…

"gulp-sass": "^3.0.0"

},

"devDependencies": {

// …

}

Nous utilisons l’option « –save » pour installer les dépendances nécessaires à l’application pour se lancer, et l’option « --save-dev » pour les dépendances utilisées par le développeur pour… développer. Proprement.

Prenez le temps de lire le nom des packages, nous les reverrons au fur et à mesure que nous avançons.

Créons maintenant notre arborescence.

* Un dossier « src » et un dossier « dev » à la racine (là où se trouve le fichier « package.json » donc)
* Un dossier « build » à l’intérieur de « src ». Il contiendra nos bundles Javascript et CSS
* Un dossier « js » et un dossier « css » à l’intérieur de « build » pour organiser nos bundles plus tard
* Un dossier « js » et un dossier « scss » à l’intérieur du dossier « dev » pour organiser notre application

Maintenant, on va coder un peu. On reprendra l’arborescence un peu plus tard.

Commençons par le point de départ de notre application. Celui-ci est en deux parties : un fichier « index.html » et un fichier « index.jsx » (« jsx » oui, l’extension créée par Facebook pour le virtual DOM utilisé par ReactJS. Want more ? 🡺 Google is your friend).

Le fichier « index.html » est tout simple. Une page html contenant une balise « div » avec un id nommé « root ». Rendez-vous dans le dossier « src » (oui, c’est bien « src » et non pas « dev ») pour créer le fichier « index.html » et y inclure le code suivant :

<!doctype html>

<html lang="fr">

<head>

<meta charset="utf-8">

<title>My Wonderful Application</title>

</head>

<body>

<div id="root"></div>

<script src="build/js/app-bundle.js"></script>

</body>

</html>

C’est tout. Ce fichier ne sert qu’à l’affichage de toutes les informations visuelles. Notez l’ajout d’une balise « script » à la fin de notre « body ». Nous ajoutons ici le bundle de l’application qui sera généré par Webpack plus tard. Vous pouvez évidemment changer le nom ou la destination, mais penser à modifier cette ligne aussi.

Le fichier « index.jsx » quant à lui, va récupérer toutes les informations de l’application et les envoyer à notre page html, dans la « div » prévue à cet effet.

Rendez-vous dans le dossier « dev » et créez un fichier « index.jsx » et incluez-y le code suivant :

import React from 'react';

import ReactDOM from 'react-dom';

ReactDOM.render(

<h1>Hello World !</h1>,

document.querySelector('#root'),

);

Ici, on importe les packages React et ReactDOM installés précédemment, et on appelle ensuite la méthode « render » afin d’afficher notre virtual DOM (la balise « h1 » ici), dans la balise « div » prévue à cet effet (avec l’id « root »).

Maintenant que le point d’entrée est préparé, nous pouvons nous concentrer sur le cœur de l’environnement : Webpack.

C’est lui qui va permettre de créer les bundles souhaités, gérer les packages, … comme son nom l’indique, il va créer un paquet tout prêt à l’utilisation pour votre navigateur internet (« web », « pack », jusque-là ça devrait aller :D).

Commençons par créer le fichier de configuration.

Rendez-vous dans le dossier racine (là où se trouve « package.json ») et créez un fichier nommé « webpack.config.js » avec le code suivant :

const path = require('path');

module.exports = {

entry: './dev/js/index.jsx',

output: {

path: path.resolve(\_\_dirname, 'src'),

filename: 'build/js/bundle.min.js'

},

devtool: 'source-map',

module: {

loaders: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

loaders: 'babel-loader',

query: {

presets: ['es2015', 'stage-0', 'react']

}

}

]

},

};

Qu’avons-nous ici :

* « entry » : vous reconnaitrez le point d’entrée que nous avons créé tout à l’heure
* « output » : c’est un objet contenant :
  + « path » : le dossier racine de nos bundles
  + « filename » : le nom du bundle. Vous noterez le complément concernant le chemin d’accès. Nous ne renseignons pas complétement le chemin dans « path » afin de pouvoir proprement séparer les points de sorties (bundles). Nous verrons cela plus tard lorsque nous extrairons le CSS.
* « devtool » : Cela permet de créer des fichiers .map afin d‘améliorer le débogage
* « module » : Un objet contenant une clef :
  + « loaders » : un tableau contenant les différents loaders de notre application. Chaque objet représente des conditions différentes. Par exemple, les loaders utilisés pour le SASS seront dans un autre objet, et la clef « test » sera différentes.
    - « test » : représente les extensions des fichiers auxquels doivent être appliqués les loaders. Notez qu’il s’agit d’une expression régulière
    - « exclude » : les dossiers à exclure dans l’analyse du projet par Webpack. Notez qu’il s’agit d’une expression régulière ici aussi
    - « loaders » : le loader appliqué aux fichiers trouvés
    - « query » : un tableau contenant les presets à appliquer. Pour faire simple, « query » sert simplement pour dire à Webpack « charge babel-preset-es2015, babel-preset-stage-0 et babel-preset-react ». Nous pourrions très bien écrire :

« Loaders : [‘babel-preset-es2015’, ‘babel-preset-stage-0’, ‘babel-preset-react ‘] »

Voilà ! Grace à cette configuration, Webpack possède toutes les informations nécessaires pour générer les bundles et faire fonctionner l’application.

Pour tester notre application, nous devons ajouter une ligne dans le package.json :

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

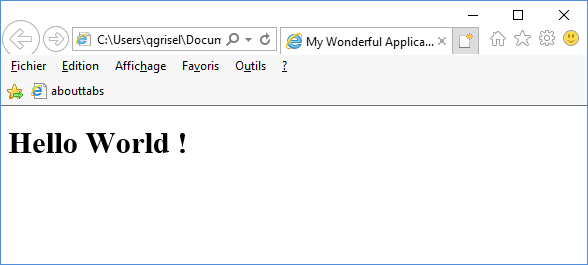
"build": "webpack"

}

Ajoutez le script (il ne s’agit en fait que d’une ligne dans le fichier, rien de plus) contenant la clef « build », sauvegardez et il ne vous reste plus qu’à tester :

Npm run webpack

Webpack va générer vos bundles et il en vous reste plus qu’à double-cliquer sur le fichier « index.html ». Vous devriez avoir le résultat suivant :



Si ce n’est pas le cas, consultez la console, vous avez les messages d’erreur un peu plus haut. La communauté autour de Webpack/React est très grande, si bien que tous les problèmes que vous aurez, des personnes les aurons déjà eus ;)

La prochaine étape, non obligatoire mais conseillée, est d’utiliser « webpack-dev-server ». Il s’agit d’un outil permettant, entre autres, d’utiliser le protocole « http » plutôt que « file » afin de se rapprocher d’un environnement de production.

Pour l’utiliser, installez-le comme suit :

Npm install –save-dev webpack-dev-server

Puis créez un script dans votre package.json :

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack",

"start": "webpack-dev-server"

}

Enfin, dans votre fichier « webpack.config.js », ajoutez un objet permettant de configurer webpack-dev-server :

output: {

//...

},

devServer: {

publicPath: '/',

contentBase: './src',

hot: true

},

Nous avons ici :

* « publicPath » : Le chemin du dossier racine du **projet**
* « contentBase » : Le chemin du dossier contenant notre point d’entrée .html
* « hot » : Cette clef, non présente dans les prochaines étapes de la formation, permet d’ajouter le plugin HotModuleReplacement afin de recharger la page automatiquement.

Ce dernier peut être ajouté via un plugin comme suit :

module: {

loaders: [

//...

]

},

plugins: [

new webpack.HotModuleReplacementPlugin()

]

**Mais attention à n’utiliser qu’une seule de ces deux méthodes !**

Exécutez maintenant votre application via le nouveau script :

Npm run start

Ouvrez votre navigateur et accédez à <http://localhost:8080> et essayez de modifier « Hello world » dans « index.jsx » par autre chose, la page du navigateur devrait se mettre à jour automatiquement ! Vous noterez dans la console la présence d’informations que nous avons renseignées dans notre configuration, à savoir contentBase et publicPath.

Maintenant que cela fonctionne, nous allons continuer par la configuration de l’éditeur. En effet, il est intéressant de parfois spécifier ses propres règles lorsque l’on développe. La taille des tabulations par exemple.

Pour ce faire, puisque j’utilise Visual Studio Code, j’ai besoin de télécharger une extension nommée « EditorConfig for VS Code ».

Ensuite, créez simplement un fichier à la racine du projet nommé « . editorconfig » dans lequel nous spécifierons quelques règles :

root = true

[\*]

indent\_style = space

indent\_size = 2

charset = utf-8

trim\_trailing\_whitespace = true

insert\_final\_newline = true

end\_of\_line = LF

# editorconfig-tools is unable to ignore longs strings or urls

max\_line\_length = null

La configuration est très simple ! Relancez votre IDE et le tour est joué !

Enfin, comme toute dernière partie, nous allons mettre en place un contrôle de code via ESLint. Si vous avez bien lu les packages que nous avons installé au tout début, vous vous rappellerez sûrement de la config airbnb. Il s’agit d’une configuration pré faite que nous allons utiliser pour nous simplifier la vie. J’utilise celle-ci, mais d’autres existent également.

Vous devez d’abord installer l’extension « ESLint » pour Visual Studio Code.

Ensuite, créez un fichier nommé « .eslintrc.js » à la racine du projet et placez-y ce bout de configuration :

module.exports = {

"extends": "airbnb",

  "rules": {

    "react/no-unescaped-entities": 0,

    "max-len": [1, 120, 2, {ignoreComments: true}],

    "react/no-unused-prop-types": [1, {skipShapeProps: true }],

    "no-plusplus": 0,

    "quotes": 0,

  },

};

La configuration est assez claire d’elle-même :

* « extends » : on ajoute une configuration de base, histoire de ne pas tout faire soi-même, ce serait dommage de réinventer la roue
* « rules » : cet objet permet de « surcharger » les règles de la configuration étendue et d’en ajouter si besoin. Sans configuration pré faite, vous devrez écrire toutes vos règles ici.

Toute les règles sont disponibles sur internet et la communauté traitant ESLint est identique à celle de WebPack ou React, ces outils marchants très bien ensembles.

Relancez votre IDE et ESLint devrait vous mettre des erreurs partout, sauf si vous codez très proprement !

Encore une fois, en cas de problème, la communauté aura la réponse adéquate !

Ceci conclu la première partie de la formation, j’ai nommé : mise en place de l’environnement de développement de base.

Le fichier « webpack.config.js » est amené à être un peu modifié, mais rien de bien méchant, l’essentiel pour faire fonctionner une application React se trouve ici.