CRÉER UNE NOUVELLE APPLICATION REACT

Sommaire

I. Approche React

React change complètement d'approche. Il part d'un constat simple : le fait que le DOM ait constamment un état différent, c'est ce qui est difficile à gérer.

Du coup, et si on appelait `.render()` à chaque modification ?

React implémente un **DOM virtuel**, une représentation interne du DOM extrêmement rapide. La méthode `render` retourne des objets correspondant à la représentation interne du DOM virtuel.

Un composant React reçoit des `props`, ses paramètres, et peut avoir un `state`, un état local. Lorsque l'on définit un composant React, on peut y mettre un état par défaut en définissant une propriété `state` dans la `class`.

Après cela, on peut mettre à jour l'état avec la méthode `this.setState`, en y passage les valeurs de l'état à changer afin de mettre à jour le DOM.

Le principal avantage est que l'on est certain, du fait de l'appel systématique à `render`, que notre composant React aura la représentation attendue pour un état donné, et ce à n'importe quel point dans le temps.

Un des autres avantages de React est son algorithme de diff interne. Le DOM virtuel va être comparé avec la version visible, et React effectue à l'aide d'opérations simples les seuls changements nécessaires.

Cela résoud des problématiques comme la position du curseur dans un champ texte qui reçoit sa valeur de JavaScript; puisque l'algorithme n'y voit pas de changement nécessaire, le champ texte n'est pas modifié et l'on garde donc le focus. Du même fait, si vous avez un gif qui boucle, il ne se relancera pas inopinément.

React peut être utilisé avec JSX, un *superset* de js qui permet d'écrire les templates avec une syntaxe XML (voir l'exemple plus bas), ce qui permet de le prendre en main très rapidement.

II. Qu'est-ce que le DOM virtuel?

Le DOM virtuel (VDOM) est un concept de programmation dans lequel une représentation idéale ou «virtuelle» d'une interface utilisateur est conservée en mémoire et synchronisée avec le «vrai» DOM par une bibliothèque telle que ReactDOM. Ce processus s'appelle la réconciliation .

Cette approche active l'API déclarative de React: vous indiquez à React dans quel état vous souhaitez que l'interface utilisateur se trouve, et vous assurez que le DOM correspond à cet état. Ceci résume la manipulation des attributs, la gestion des événements et la mise à jour manuelle du DOM que vous auriez sinon dû utiliser pour créer votre application.

Étant donné que le «DOM virtuel» est plus un modèle qu'une technologie spécifique, les gens le disent parfois pour signifier différentes choses. Dans le monde React, le terme «DOM virtuel» est généralement associé aux éléments React car il s'agit des objets représentant l'interface utilisateur. Cependant, React utilise également des objets internes appelés «fibres» pour stocker des informations supplémentaires sur l'arborescence des composants. Ils peuvent également être considérés comme faisant partie de la mise en œuvre du «DOM virtuel» dans React.

III. Approche ES2015 :

Depuis que EcmaScript6 (ou 2015), le nouveau standard pour l'implémentation de JS, est plus ou moins sur les rails, je regarde les conférences qui y sont dédiés et les différents articles qui parlent de ses nouveautés. Mais pour aller un peu plus loin et s'approprier ces nouvelles syntaxes et techniques, j'ai mis la main à la pâte. Première chose : comment utiliser les éléments et syntaxe ES6 aujourd'hui ? Avant de parler des nouveaux trucs intéressants, parlons du support.

A. Côté front : Babel

C'est d'autant moins valable que comme à chaque fois dans cette situation, des solutions viennent contourner/résoudre le problème. Effectivement, vous pouvez utiliser ES6 dès aujourd'hui dans vos projets front avec Babel, que vous pouvez glisser dans votre outil de build préféré (via un module Gulp/Webpack/Browserify par exemple) pour réécrire le code de façon à le rendre compatible ES5.

Babel peut s'intégrer sous d'autres formes, notamment un appel vers un CDN qui, sauf pour tests maison vite fait, ne sera jamais une bonne solution.

Il faut quand même noter qu'hormis Internet Explorer, les navigateurs, dans leurs versions les plus récentes supportent plutôt bien ces fonctionnalités et qu'il serait dommage de s'en priver.

B. Côté back et mobile

Le support côté back, via des technologies comme Node par exemple s'en sortent pas trop mal. Node 5 et + supporte la moitié des éléments d'ES6 et tend à chaque version vers une prise en charge plus importante.

Pour les plus impatients, il existe io.js, fork de Node qui implémente la norme ES6.

Autre exemple, React Native embarque Babel, donc pour vos apps mobiles natives avec JS, ne vous en priver pas.

IV. Compatibilité entre les navigateurs avec React Native

Bien que l'approche multiplateforme nous aide à créer des applications robustes, le problème de la compatibilité entre navigateurs reste dans une certaine mesure difficile. Nous nous tournons maintenant vers son écosystème parent. Le principal avantage de l'utilisation de React natif est qu'il est similaire à React et même Preact. Vous pouvez donc, si vous le souhaitez, utiliser cet avantage à votre avantage. Lors de la création d'une application Web, vous pouvez exploiter la configuration prédéfinie et vous épargner du travail acharné. Il est livré avec la configuration PWA aussi, utilisez des ouvriers de service sans charge supplémentaire.

Il offre même des fonctionnalités brillantes telles que le rechargement en direct, qui vous permettent de mettre à jour ou de modifier le code source lors de vos déplacements. C'est vraiment impressionnant car cela apporte un certain sentiment de liberté que n'offrent tout simplement pas d'autres frameworks / bibliothèques.

De plus, React implémente un système DOM totalement différent, indépendant du navigateur. La mise en œuvre est effectuée afin de rendre l'application compatible avec tous les navigateurs.
Les pertes de mémoire sont moindres lorsque vous utilisez React en natif, ce qui améliore directement les performances.

V. Présentation de ReactJS :

ReactJs est un Framework JavaScript qui a été conçu par les équipes de Facebook. Il permet de développer des applications web modernes en utilisant la notion de composants. Plus précisément, il correspond à la partie vue du modèle MVC.

Comme souvent en JavaScript, le premier obstacle est le manque de documentation des différents outils et le second est de s'orienter parmi les possibilités qui nous sont offertes. En effet, bien que ReactJs puisse fonctionner en « stand alone », il existe beaucoup d'outils pour vous simplifier la vie.

En plus de ReactJs nous allons donc utiliser quelques outils et technologies qui simplifieront le développement. Si comme moi vous ne maîtrisez pas tous ces outils, ne vous inquiétez pas, nous allons les passer en revue et identifier ce qu'il est nécessaire de savoir !

Nous utiliserons la configuration suivante pour créer le projet :

- es6
- jsx
- babel
- webpack
- webpack-dev-server

VI. Contexte

A. ES6

ECMAScript 6 (ou ES6) est un standard d'implémentation pour JavaScript qui ajoute de nouvelles syntaxes au JavaScript actuel.

Nous utiliserons les éléments d'ES6 suivants :

- les modules, avec `import` et `export` (voir un exemple);
- les classes (voir un exemple);

Il faut savoir que la syntaxe ES6 n'est pas supportée par les navigateurs actuels. Heureusement pour nous, il existe des outils qui transpilent (transforment) automatiquement le code ES6 en code JavaScript standard qui lui est supporté. Dans notre cas, c'est Babel qui s'occupera du sale boulot.

Plus de détails sur ES6

B. JSX

JSX est une extension de la syntaxe JavaScript qui permet d'écrire du code sous forme de balises directement dans un JavaScript. Par exemple avec JSX nous pouvons écrire :

```
var element = <div id="main-element" class="title"
>Hello, world!</div>;
```

Il s'agit simplement d'un sucre syntaxique qui doit être transformé en code JavaScript pour pouvoir être compréhensible par un navigateur. Dans notre cas, c'est Babel qui va transformer ce code JSX en composant JS utilisant ReactJs. L'exemple précédent serait donc transformé comme suit :

```
var element = React.createElement(
  "div",
  {id: 'main-element', className: "title"},
  "Hello, world!"
)
```

Il est tout à fait possible d'écrire directement ce code, pour éviter une phase de compilation. Mais ce genre de code est moins lisible et donc moins maintenable quand on commence à imbriquer les composants.

Notez que l'attribut HTML class s'écrit className en JSX, cela vient du fait que le mot class est protégé en JS.

Plus de détails sur JSX

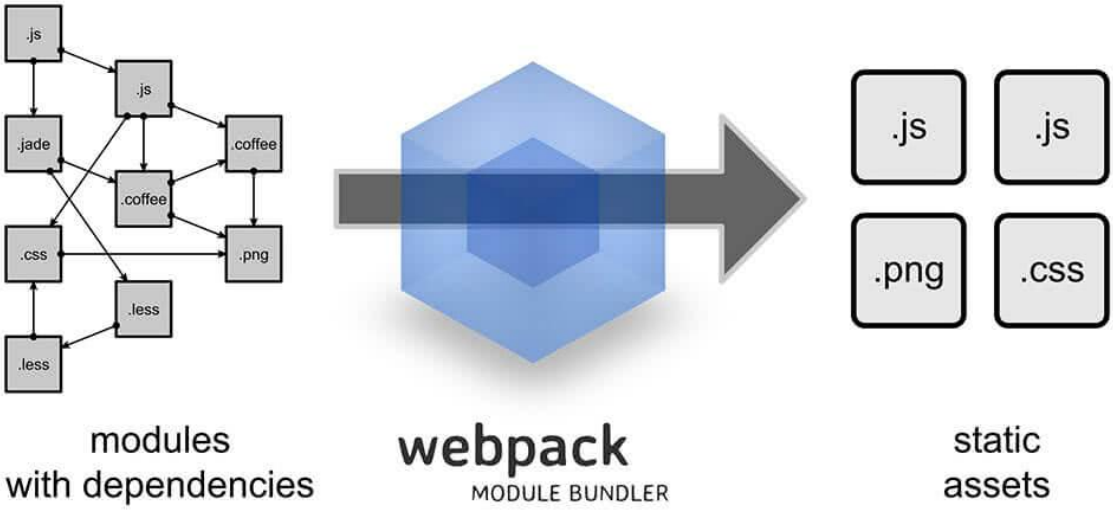
C. Babel

Babel est un compilateur JavaScript qui a pour rôle de transformer du code JavaScript « moderne » (ES6, ReactJs ou autre) en une version compréhensible par tous les navigateurs.

Plus de détails sur Babel

D. Webpack

Webpack est un constructeur de modules que nous utiliserons pour notre application, il se charge de découvrir vos modules et toutes leurs dépendances, de les transformer en utilisant des loaders, puis de les packager dans des fichiers statiques contenant la totalité du code prêt à être utilisé.



Il délègue la transformation du code JavaScript à Babel. La configuration de webpack est définie dans webpack.config.js. Pour en savoir plus sur la configuration de Webpack, je vous invite à lire Configuration de développement versus configuration de production avec Webpack.

Plus de détails sur Webpack

E. webpack-dev-server

Enfin, nous utiliserons webpack-dev-server pour héberger notre application. C'est un serveur de développement en NodeJs dont le rôle est de servir les fichiers statiques et de gérer le live reload.

Le live reload fonctionne de la manière suivante :

1. Le serveur écoute les modifications dans le code source JavaScript
2. Il fait appel à Webpack pour recompiler le code modifié
3. Il notifie le client de la modification pour mettre à jour uniquement ce qui a changé, pas besoin de recharger complètement la page.

Plus de détails sur webpack-dev-server

F. Mise en place de l'environnement

Nous considèrerons que NodeJs et npm sont déjà installés, si ce n'est pas le cas, voici la marche à suivre.

La première chose à faire est d'installer les outils que nous allons utiliser pendant le projet. Pour rappel, nous avons besoin de babel, webpack et webpack-dev-server.

Etant donné qu'ils peuvent vous être utiles dans d'autres projets, vous pouvez les installer globalement (-g) :

```
$ npm install babel webpack webpack-dev-server -g
```

VII. Initialisation du projet (avant pas à pas)

Nous pouvons maintenant rentrer dans le vif du sujet : créer une application utilisant ReactJs « from scratch ». Nous partons ici d'un répertoire vide, mais vous pouvez également ajouter ces éléments dans un projet existant.

Dans votre terminal, mettez-vous à la racine de votre projet. Nous allons commencer par initialiser un projet NodeJs :

```
$ npm init
```

Le terminal va alors vous poser un certain nombre de questions, vous pouvez y passez du temps pour bien faire les choses… ou appuyer sans réfléchir une bonne dizaine de fois sur la touche « Entrée » pour vous débarrasser de cela rapidement (vous pourrez modifier ces informations après) !

```
name: (myproject)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to
/Users/gbe/Development/myproject/package.json:

{
  "name": "myproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" &&
exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this ok? (yes) yes
```

La commande **npm init** vient de créer un fichier package.json dans notre répertoire. Ce dernier va contenir entre autre tous les modules NodeJs dont nous aurons besoin pour le projet.

Nous pouvons maintenant importer les modules, ces derniers seront alors automatiquement ajoutés dans le package.json.

Ainsi, si une autre personne récupère le projet il aurra simplement à exécuter la commande **npm install** pour importer automatiquement tous les éléments nécessaires :

Les modules nécessaires pour utiliser ReactJs sont react et react-dom :

```
$ npm install react react-dom --save
```

Il faut également importer les modules qui seront nécessaires à Babel pour compiler le code JavaScript :

```
$ npm install babel-loader babel-core babel-preset-
es2015 babel-preset-react webpack --save-dev
```

A. Création de la structure du projet

Notre HelloWorld est un projet simple, nous avons donc besoin de très peu de choses pour le faire fonctionner. Les fichiers nécessaires sont les suivants :

- webpack.config.js : le fichier de configuration de webpack
- index.html : la page web qui va acceuillir notre application.
- main.js : le fichier JavaScript qui sert à injecter l'application dans la page.
- App.js : l'application à proprement parler

```
$ mkdir app
$ touch index.html app/App.js app/main.js
webpack.config.js
```

B. webpack.config.js

Il s'agit du fichier de configuration de webpack, c'est grâce à lui que nous allors pouvoir transformer notre code en Javascript classique. Nous allons donc lui spécifier les actions et les modules à mettre en place.

Il doit contenir les éléments suivants :

```
module.exports = {
    entry: "./app/main.js",
    output: {
        path: "./",
        filename: "index.js"
    },
    devServer: { // configuration du server
permettant le live-reload
        inline: true,
        port: 8080
    },
    module: {
        loaders: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                loader: 'babel-loader',
                query: {
                    presets: ['es2015','react']
                }
            }
        ]
    }
}
```

Voici le détail de la configuration :

- nous spécifions le point d'entrée de l'application, dans notre cas main.js

```
    entry: "./app/main.js",
```

- nous indiquons le chemin où le bundle final sera généré par webpack, c'est ce fichier qui devra être importé dans la page web

```
    output: {
        path: "./",
        filename: "index.js"
    },
```

- nous définissons les loaders qui seront lancés pendant la compilation et leur ordre d'application

```
module: {
        loaders: [
            {
                test: /\.js$/, // On ne considère
que les fichiers *.js
                exclude: /node_modules/, // On ne
s'occupe pas des fichiers présents dans
node_modules
                loader: 'babel', // On lance Babel
pour gérer es2015 et react
                query: {
                    presets: ['es2015','react']
//On utilise les presets de gestion de la syntaxe
ES6 et de ReactJs
                }
            }
        ]
    }
```

- nous configurons le serveur sur le port 8080 en mode inline. C'est ici que la magie opère, il n'y a rien de plus à faire, le code qui sert au live reload est automatiquement incorporé par webpack dans le bundle

```
devServer: { // configuration du server permettant le
live-reload
        inline: true,
        port: 8080
    },
```

C. index.html

Le fichier html est ici volontairement léger, mais une application ReactJs peut être ajoutée dans n'importe quelle page. Il suffit de créer une balise pour contenir l'application et d'ajouter le script qui va s'occuper du reste.

```
<!DOCTYPE html>
  <html>
  <head>
      <meta charset="UTF-8">
      <title>React 101</title>
  </head>
  <body>
  <div id="app"></div>
  <script type="text/javascript"
src="index.js"></script>
  </body>
  </html>
```

Le conteneur dans lequel nous allons injecter l'application a pour id app, et seul le bundle généré par webpack (index.js) est nécessaire.

D. App.js

Nous allons maintenant réaliser notre application HelloWorld avec ReactJs en utilisant les syntaxes ES6 et JSX.

Comme dans beaucoup de lanquages, il existe une convention qui consiste à commencer le nom des composants React par une majuscule.

Voici donc l'implémentation de notre HelloWorld :

```
import React from "react";

    class App extends React.Component {
      render() {
        return <div id="main-element"
className="title">Hello World !</div>
      }
    }

    export default App;
```

La première chose qui est faite est l'import de la librairie react en utilisant la syntaxe ES6.

```
    import React from "react";
```

Ensuite nous déclarons notre composant React, il s'agit ni plus ni moins d'une classe étendant la classe Component de ReactJs. Encore une fois, nous utilisons la syntaxe ES6 :

```
    class App extends React.Component {
    }
```

Pour que notre composant soit fonctionnel, il est obligatoire d'implémenter la méthode `render()` qui doit retourner un composant. Avec JSX, nous pouvons facilement décrire la div qui contiendra « HelloWorld »

```
    render() {
        return <div id="main-element"
className="title">Hello World !</div>
    }
```

Comprenez bien que nous n'avons pas écrit du code HTML dans notre fichier JavaScript, nous utilisons simplement une syntaxe (JSX) proche du HTML qui sera transformé en code JavaScript standard. D'ailleurs, si nous n'avions pas utilisé JSX, nous aurions dû écrire :

```
    render() {
        return React.createElement(
                  "div",
                  {id: 'main-element', className:
"title"},
                  "Hello, world!"
                )
    }
```

Enfin, nous devons exporter notre composant pour qu'il puisse être utilisé à l'extérieur du module App, tout comme l'import, nous utilisons la syntaxe ES6

```
export default App;
```

Si vous trouvez qu'un composant aussi simple ne devrait pas prendre 7 lignes à définir, sachez que vous pouvez également l'écrire de la manière suivante :

```
  import React from "react";

  export default const App = () => <div id="main-element" className="title">Hello World !</div>;
```

Par défaut, si un composant ne contient qu'une méthode, il s'agit de la méthode **render().** Cette écriture est très pratique pour tous les composants simples qui ne contiennent pas d'état interne.

E. main.js

C'est le point d'entrée de l'application, c'est à dire le code qui va injecter notre composant App dans la page HTML. Comme il ne s'agit pas d'une classe, son nom commence par une minuscule.

```
  import React from "react";
  import ReactDom from "react-dom";

  import App from "./App";

  ReactDom.render(<App />, document.getElementById('app'));
```

Comme précédemment, nous avons besoin du module **react**. Nous aurons également besoin de **react-dom** qui contient les méthodes permettant d'injecter notre application. Finalement, nous importons notre **App**.

```
  import React from "react"; // qui se trouve dans
node_module
  import ReactDom from "react-dom"; // qui se trouve
dans node_module
  import App from "./App"; // qui ne se trouve pas
dans node_module, nous devons donc spécifier le
chemin
```

Nous allons alors utiliser react-dom pour injecter App dans la balise HTML dont l'id est « app » :

```
  ReactDom.render(<App />,
document.getElementById('app'));
```

F. Vue d'ensemble

Si tout s'est bien passé, voici à quoi ressemble le répertoire du projet:

```
.
├── app
│   ├── App.js
│   └── main.js
├── index.html
├── index.js
├── node_modules
│   └── ...
│
├── package.json
└── webpack.config.js
```

G. Compiler le projet

Nous pouvons maintenant transpiler notre projet en un code exécutable par un navigateur :

```
$ webpack
```

H. Lancer le server avec hot-reload

Pour démarrer webpack-dev-server, il faut lancer :

```
$ webpack-dev-server
```

Nous pouvons aussi utiliser les scripts de npm pour démarrer le serveur en ajoutant dans package.json :

```
  "scripts": {
    "server": "webpack-dev-server"
  },
```

Nous lancerons alors le serveur avec :

```
$ npm run server
```

L'application est maintenant accessible sur le port 8080 de notre locahost.

VIII. Nouvelle commande NPM pour créer un projet Reactjs

A. Création du projet avec assistant :

- Installer la création de projet reactJS :
  - npm install -g create-react-app
- create-react-app cge-app
- cd cge-app
- Après installation :

```
cge-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── registerServiceWorker.js
```

B. Commandes à dispo :

- yarn start : lance l'application avec un serveur dans le navigateur par défaut sur : http://localhost:3000
- yarn test : lance les tests spécifiés dans les fichiers *.spec.js
- yarn build : créer notre artefact/bundle/build pour déployer « l'application ReactJS ».