Framework front-end ReactJS

Membre 1 : Pierre Bastien

Membre 2 : Ruell Arnaud

# Signalétique

|  |  |
| --- | --- |
| Version | 16.5.0 |
| Langage (Javascript, typescript, etc.) | Javascript |
| Divers | Utilisation de Redux et de React-router |
| Site officiel | https://reactjs.org/ |

# Installation

Responsable membre 1

## Prérequis techniques

## - Il faut que node packet manager (npm) soit installé dans sa dernière version

- Il faut que le paquet « *create-react-app* » soit installé en global (*npm install -g create-react-app*)

## Installation

- Lancer la commande create-react-app nomduprojet

### Environnement de développement

Windows 10 (64bits)

### Environnement de production

Ubuntu 18.04

## Organisation des fichiers

## 

- package.json va regrouper les dépendances de notre application ainsi que des informations comme la version, le nom ou les scripts qui nous permettront d’exécuter l’application.

C’est dans le dossier src que nous mettrons nos components et après à vous de voir comment les organiser avec des dossiers.

Dans le dossier *src*  
- *index.js* va être le point d’entrer de l’application, il va juste rendre *App.js* dans la page *index.html*.  
- *App.js* est le component de départ, c’est donc là que nous allons commencer à ajouter des components.  
- *index.css* sera le css qui sera partagé à l’application entière  
- *App.test.js* va nous permettre de faire des tests sur le component *App*  
- *App.css* est le css qui sera appliqué uniquement au component *App* par convention  
- *serviceWoker.js* nous permet d’enregistrer un service worker mais nous ne l’utiliseront pas ici

## Modules supplémentaires

- Déplacez-vous dans le dans le dossier du projet et lancer le terminal dans celui-ci  
 - Lancer la commande npm install redux react-redux redux-thunk react-router-dom  
 (Si vous avez une erreur dans l’installation avec redux-thunk relancer l’installation en tant qu’administrateur)

- Redux sera la librairie qui permet d’avoir une achitecture flux, ce qui facilitera le partage de données et leur cohérence entre les vues  
 - React-redux va permettre d’avoir des outils pour nous faciliter l’utilisation de Redux avec un projet React  
 - Redux-thunk va nous permettre de faire des fonctions asynchrones dans la libraire Redux  
 - React-router-dom va nous permettre d’avoir un système de routage

Nous utiliserons la commande npm start pour démarrer l’application et nous nous rendrons sur le lien mentionné dans la console

# Création de composants web

Responsable membre 2

## Description d’un composant web

React permet de créer des composants web héritant de la classe « Component » de React. Ces composants sont des classes Javascript qui possèdent la méthode « render » qui est appelée pour effectuer l’affichage de la page. Il suffit ensuite de retourner depuis cette fonction l’affichage qui est décrit comme une page HTML. Il y a ensuite les « reducers » et les « actions » de Redux qui vont permettre d’interagir avec le back-end. Il suffit d’appeler les méthodes dans les « actions » qui vont-elles même appeler le « reducer » qui mettra à jour les informations de l’application web et qui actualisera la vue. LA communication avec le back end se fait à l’aide de fetch. Les informations de l’application étant récupérées depuis le back-end sont des objets Javascript qui peuvent donc être directement utilisés dans la vue.

Le composants web sont rangés dans le dossier « pages ».

Plus de détail sur React : <https://reactjs.org/docs/getting-started.html>

Doc Redux pour comprendre comment l’implémenter : <https://redux.js.org/>

## Exemple de composant de top niveau ou destiné au routing

Provider : Permet de donner le « store » qui contient les données à tous les composants de l’application.

BrowserRouter : Permet de faire le routage de l’application.

Header : Représente le header qui va apparaitre sur chaque page.

Navbar : Représente la barre de navigation qui va apparaitre sur chaque page.

Route : Permet d’afficher le composant correspondant à la route utilisée.

## Exemple de composants de haut niveau

La méthode « render » permet d’afficher la page. On y retrouve des balises avec des « className » qui permettent de styliser l’affichage (elles correspondent à l’attribut *class* d’une balise HTML) et on trouve aussi des appels aux méthodes « renderEvents » et « renderPartners » qui permettent de générer l’affichage de chaque objet évènement et partenaire. L’affichage de ces objets est géré par les sous-composants « Event » et « Partner ». Après avoir importé ces composants (qui agissent de la même manière que le composant « Dashboard ») on peut simplement les appeler là où on souhaite les afficher. Chaque objet est passé en propriété via « data={item} » au sous-composant correspondant ce qui permet à ce dernier d’utiliser les données de l’objet.

## Exemple de composant avec formulaire

On créé le formulaire avec la balise <form> et on lui donne dans « onSubmit » une fonction qui va empêcher son évènement par défaut puis qui va aller effectuer un appelle vers la fonction « fetchTown » qui permet de récupérer la commune si son nom et son mot de passe (passés dans un objet comme argument) correspondent à une entrée dans la base de données.

# Routage frontend

Responsable membre 1

## Introduction

Le fichier concerné est celui où le component « BrowserRouter » est présent.

Doc : <https://reacttraining.com/react-router/web/guides/quick-start>

Voici un exemple de mise en place d’un système de routage :

<BrowserRouter>

<div className="h-100">

<Route exact path="/" component={Home}/>

<Route path="/dashboard" component={Dashboard}/>

<Route path="/citizen" component={Citizen}/>

<Route path="/event" component={Events}/>

<Route path="/partner" component={Partners}/>

</div>

</BrowserRouter>

Pour voyager entre les route on utilisera la balise <Link to= "/"> au lieu de <a href= "/">.

## Routage simple

<Route path="/dashboard" component={Dashboard}/>

## Routage avec paramètres explicites

Nous n’en utilisons pas dans le projet mais voici un exemple :

<Route path="/event/:id" component={Event}/>

Et on récupère le paramètre comme cela:

function Event(props) {

return <h1>Evènement n° {props.match.params.id}!</h1>;

}

Framework backend : ExpressJS

# Signalétique

|  |  |
| --- | --- |
| Version | 4.16.3 |
| Langage (Javascript, typescript, etc.) | Javascript |
| Divers | Utilisation de mongoose |
| Site officiel | https://expressjs.com/ |

# Installation

Responsable membre 2

## Prérequis techniques

Avoir *node package manager* installé sur l’ordinateur.

## Installation

Lancer la commande : *npm install express* dans le dossier du projet.

### Environnement de développement

Windows 10 (64bit)

### Environnement de production

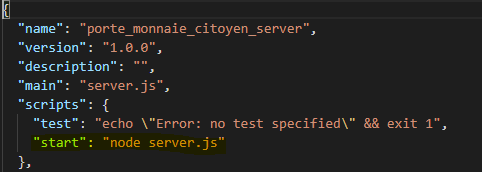
Ubuntu 18.04

## Organisation des fichiers

ExpressJS ne demande pas d’organisation particulière de ses fichiers.

## Configuration

Il faut spécifier dans le « package.json » quel est le fichier du serveur (qui doit se trouver à la racine du projet) et pour cela il faut ajouter dans « scripts » l’entrée « **"start" : "node {nom du fichier serveur}"** ».



## Modules supplémentaires

body-parser : Permet d’utiliser les json dans les réponses.

app.use(bodyParser.json());

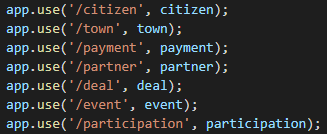
Nous démarrerons le serveur avec un *npm start* et nous y accèderons depuis le lien mentionné.

# Routage backend

Responsable membre 2

## Introduction

Il faut créer un fichier qui va servir en tant que serveur dans lequel on va définir toutes les routes utilisables par le serveur.



On crée le serveur avec la ligne suivante à la fin du fichier serveur :

app.listen(50001, () => console.log('Server started on port 50001'));

Il faut ensuite créer les fichiers qui vont contenir les différentes routes et utiliser au début des fichiers :

const express = require('express');

const router = express.Router();

## Routage simple

router.get("/", (req, res) => {

Citizen.find()

.populate('town')

.then(citizens => res.json(citizens))

.catch(error => res.json({error : "Impossible de récupérer les citoyens"}))

});

## Routage avec paramètres explicites

router.get("/:id", (req, res) => {

Citizen.findOne({\_id : req.params.id})

.populate('town')

.then(citizen => res.json(citizen))

.catch(error => res.json({error : "Impossible de récupérer le citoyen"}))

});

# Base de données

Responsable membre 1

## ORM

On utilise « Mongoose » qui est un ORM pour « MongoDB » (NoSql) et qui utilise « CodeFirst ».

Définition d’une entité/table/schéma :

const mongoose = require('mongoose');

const DealSchema = new mongoose.Schema({

name: {

type: String,

required: true

},

price: {

type: Number,

required: true

},

partner: { Cet objet permet de créer une relation

type: mongoose.Schema.Types.ObjectId, étrangère entre « Deal » et « Partner »

ref: 'partner', Il suffirait de mettre l’objet entre

required: true crochet pour une relation many

}

});

module.exports = Deal = mongoose.model('deal', DealSchema); Permet de créer une table « deal » suivant le schéma ci-dessus

Les tables seront créées au moment où on ajoutera le premier tuple dedans (exemple : ci-dessous avec la partie « create »).

## Gestion de données

Read :

router.get('/', (req, res) => {

Deal.find()

.populate('partner') populate permet de faire un jointure avec

.then(deals => res.json(deals)) l’attribut mentionné

.catch(err => res.json(err));

});

Create :

router.post('/', (req, res) => {

newDeal = new Deal({

name: req.body.name,

price: req.body.price,

partner: req.body.partner

});

newDeal.save()

.then(deal => res.json(deal))

.catch(err => res.json(err));

});

Update :

router.put('/', (req, res) => {

Deal.updateOne({\_id: req.body.deal.\_id}, req.body.deal)

.then(deal => res.json(deal))

.catch(err => res.json(err));

});

Delete :

router.delete('/:id', (req, res) => {

Deal.deleteOne({\_id: req.params.id})

.then(d => res.json(d))

.catch(err => res.json(err));

});