

React Workshop

Ce workshop permet de découvrir `react` et son écosystème par la pratique, étape par étape !



react-workshop de [Mathieu ANCELIN et Sébastien PRUNIER](#) est mis à disposition selon les termes de la [licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International](#).

Le sujet

Lors de ce workshop, nous allons développer une application web permettant de gérer ses vins préférés* !

Les principales fonctionnalités de l'application sont :

- Lister les vins par région viticole,
- Afficher la fiche détaillée d'un vin,
- Aimer un vin,
- Ajouter un commentaire sur un vin.

* *L'abus d'alcool est dangereux pour la santé, à consommer avec modération ;-)*

Attention

Durant votre voyage initiatique au coeur de `react` vous allez vous rendre compte, avec épouvante, que malgré de nombreuses descriptions des choses à faire dans les étapes, certains morceaux de code sont déjà écrit !!! Ceci est bien évidemment volontaire de notre part pour afin de vous laisser un maximum de temps pour vous consacrer aux choses importantes et de ne pas vous ennuyer outre mesure avec des tâches fastidieuses.

Si cette philosophie ne vous convient pas, vous pouvez également démarrer votre application de gestion des vins de zéro dans un dossier à part et suivre les indications dans les diverses étapes.

Les pré-requis techniques

Les pré-requis techniques sont les suivants :

- Node (version 4.x ou 5.x) et NPM (version 3.x)
- Git
- Atom
- React Developer Tools

La dernière étape de ce workshop implique l'utilisation de `react-native`. En plus des outils précédent vous aurez besoin de :

- Toutes plateformes
 - `watchman`
 - `react-native-cli`

- iOS
 - Xcode
- Android
 - Java (JDK8)
 - Gradle
 - Android Studio

Node.js

Téléchargez et installez la version de Node.js correspondant à votre système d'exploitation, en suivant les indications disponibles sur le site officiel :
<https://nodejs.org/en/download/>

Vérifiez l'installation en lançant les commandes suivantes dans un terminal :

```
$ node -v  
v4.4.0  
  
$ npm -v  
3.8.1
```

Si npm n'est pas en version 3.x, vous pouvez effectuer la mise à jour grâce à la commande suivante : `npm upgrade -g npm`

Attention: il semblerait que le workshop fonctionne bien avec Node 0.12.x et NPM 2.x, vous pouvez donc conserver votre setup si besoin. Mais sérieusement, Node 0.12 ???

Git

Téléchargez et installez la version de Git correspondante à votre système d'exploitation, en suivant les indications disponibles sur le site officiel : <https://git-scm.com/downloads>

Vérifiez l'installation en lançant la commande suivante dans un terminal :

```
$ git --version  
git version 2.7.3
```

Atom

L'éditeur préconisé pour le workshop est [Atom](#).

Téléchargez et installez Atom, puis installez les packages suivants :

- language-javascript-jsx
- linter-eslint

Pour savoir comment gérer les packages d'Atom : <https://atom.io/docs/latest/using-atom-atom-packages>

React Developer Tools

Afin de disposer d'outils spécifiques à `react` dans votre navigateur web, installez **React Developer Tools** :

- [React Developer Tools pour Google Chrome](#)
- [React Developer Tools pour Mozilla Firefox](#)

Xcode

Xcode est installable depuis le Mac App Store présent sur votre Mac

Vous pouvez l'installer directement depuis [ici](#)

Java Development Kit

Si vous n'avez pas de JDK 8 installé, rendez-vous sur le [site d'Oracle](#) pour télécharger et installer la dernière version du JDK.

Attention, ce n'est pas parce que vous avez Java sur votre machine que vous avez le JDK. Testez avec la commande `javac`

Gradle

Gradle est un outil de build utilisé par les projets Android. Vous pouvez procurer la dernière version de Gradle [ici](#)

Android Studio

Android Studio est l'IDE officiel pour développer des applications Android. Vous pouvez télécharger une version du studio [ici](#).

Il vous faudra ensuite configurer un émulateur Android.

Pour cela, vous pouvez suivre [cette documentation](#) sur le site de `react-native`.

En ce qui concerne l'émulateur, si vous n'avez rien d'existant pour le moment, vous devez suivre [cette partie de la documentation](#) en utilisant le nom d'émulateur `reactnative` et en utilisant `1Go` de mémoire interne et `1Go` de mémoire externe et `2Go` de RAM, cf. [cette capture d'écran](#)

watchman

Sous Linux et Mac OS, vous aurez sûrement besoin d'installer `watchman`, suivez les [instructions d'installation](#).

react-native-cli

```
npm install -g react-native-cli@0.2.0
```

Pré-installer les dépendances

Pour pré-installer les dépendances NPM à l'avance, vous pouvez lancer le script `install.sh` qui lancera les commandes `npm install` dans les différentes étapes du workshop.

Nous vous recommandons également de télécharger un exemplaire des dépendances pour éviter les problèmes de réseau potentiels

- [node modules pour Mac OS](#)
- [node modules pour Linux](#)
- [node modules pour Windows](#)

Pour la partie `react-native`

- [node modules pour Mac OS](#)
- [node modules pour Linux](#)
- [node modules pour Windows](#)

Si les dépendances posent des problèmes, notamment au niveau des paquets natifs, n'hésitez pas à lancer un `npm rebuild` pour recompiler ces paquets sur votre machine.

API

L'application de gestion des vins utilise une API REST comme source de données lui permettant d'afficher des vins par régions, avec leur détail, une photo de la bouteille, etc ... Cette API est en fait un petit serveur NodeJS/Express avec des données en mémoire disponible dans le dossier [api](#). Pour toutes les étapes du workshop vous allez avoir besoin de cette API pour alimenter votre application. Vous pouvez donc déjà démarrer le serveur d'API dans un onglet à part de votre terminal afin qu'il soit toujours disponible.

Pour démarrer le serveur exposant l'API, lancez les commandes suivantes :

```
$ cd api  
$ npm install  
$ npm start
```

Rendez-vous ensuite sur <http://localhost:3000> pour parcourir la documentation des différentes routes disponibles. Nous vous recommandons de passer au moins une fois sur cette documentation afin d'avoir une idée globale des données fournies par l'API pour alimenter votre application.

Tests

De manière générale, chaque étape du workshop (hormis les premières) possède des tests pour valider le fonctionnement de l'application. Ces tests vont vous permettre de savoir où vous en êtes dans l'étape courante. N'hésitez pas à lire les tests pour voir de quelle façon l'application doit se comporter, ainsi qu'à lancer la commande magique (`npm test`) à tout moment.

Les étapes du workshop

- [Etape 0](#)
 - Mise en place des outils de build
 - webpack
 - webpack-dev-server
 - babel
 - eslint
 - Création du premier composant `react`
- [Etape 1](#)
 - Ajout de tests unitaires sur le composant créé à l'étape 0
 - react-test-utils
 - mocha
 - chai
 - jsdom
- [Etape 2](#)
 - Création d'une application `react` basique

- Découpage en composants
- Interactions entre composants
- Utilisation du `state`
- Utilisation de l'API (Appels AJAX)
 - Liste des régions viticoles,
 - Liste des vins d'une région
 - Détail d'un vin
- Bonnes pratiques `react`
 - PropTypes
 - "Dumb components" vs "Smart components"
- [Etape 3](#)
 - Single Page Applications
 - Introduction à `react-router`
 - Refactoring de l'étape 2 pour obtenir une SPA
- [Etape 4](#)
 - Ajout de fonctionnalités
 - Aimer un vin
 - Ajouter un commentaire
- [Etape 5](#)
 - Introduction à redux
 - Ajout d'une fonctionnalité avec redux
- [Etape 6](#)
 - Refactoring complet de l'application avec redux
- [Etape 7](#)
 - Nouveaux patterns `react`
 - Classes
 - HOC
 - Stateless components
 - Contexts
- [Etape 8](#)
 - `react-native`
 - Implémentation de l'application en `react-native`

A vous de jouer !

C'est le moment de commencer, si vous souhaitez en savoir plus sur la façon de monter un projet `react`, installer le tooling, etc ... vous pouvez partir de [l'étape 0](#) sinon vous pouvez directement vous rendre à [l'étape 1](#). Bon courage !!!

Etape 0 - Installation et premier composant

Package.json

Créez un nouveau dossier pour votre application. Une fois dans ce dossier lancez la commande `npm init`. Répondez aux diverses questions de l'assistant afin d'initialiser votre fichier `package.json`.

Une fois le fichier créé, installez les dépendances de la manière suivante :

```
npm install --save react@0.14.7 react-dom@0.14.7
```

avec cette commande, vous spécifiez à `npm` d'aller chercher la dernière version des paquets `react` et `react-dom` sur `npmjs.com`, de les installer en local dans le dossier `node_modules` local et de les déclarer comme dépendance dans le fichier `package.json` (via l'argument `--save`).

Une autre possibilité est de créer un fichier `package.json` et de déclarer manuellement les dépendances `react` :

```
{
  "name": "react-workshop",
  "description": "React Workshop",
  "version": "0.1.0",
  "dependencies": {
    "react": "0.14.7",
    "react-dom": "0.14.7"
  }
}
```

puis de lancer la commande `npm install` afin de télécharger localement les dépendances (elles se trouvent dans le répertoire `node_modules`)

Premier composant

wine.js

Dans le répertoire `src/components`, créez le fichier `wine.js` qui contient le code de notre premier composant, nommé `Wine` :

```
import React from 'react';

const WineStyle = {
  padding: 8,
  boxShadow: '0 1px 6px rgba(0,0,0,0.12), 0 1px 4px rgba(0,0,0,0.12)'
};

const Wine = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired
  }
});
```

```

} ,

render() {
  return (
    <div style={WineStyle}>
      {this.props.name}
    </div>
  );
}

);

export default Wine;

```

Ce premier composant est volontairement très simple (de type "hello world"), il retourne simplement le nom du vin dans un élément HTML `<div>`. Le nom du vin est passé au composant grâce à une propriété `name`. Cette propriété est définie comme étant de type `string` et obligatoire (partie `propTypes` du composant).

Un style est également appliqué au composant via l'attribut `style`.

Vous pouvez également utiliser directement la fonction `createElement` de l'API `react`:

```

render() {
  return React.createElement(
    'div',
    {style: WineStyle},
    this.props.name
  );
}

```

index.html

Dans le dossier `public`, créez une page HTML basique et ajoutez-y une `<div>` (possédant l'identifiant `main`) dans laquelle nous effectuerons le rendu de notre composant.

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>React Workshop</title>
</head>
<body>
<h1>Wines</h1>

<div id="main"></div>

</body>
</html>

```

app.js

A la racine du répertoire `src`, créez le fichier `app.js` qui contient le code nécessaire au rendu du composant `Wine` dans la `<div>` créée précédemment :

```

import React from 'react';
import ReactDOM from 'react-dom';

```

```
import Wine from './components/wine';

ReactDOM.render(
  <Wine name="Château Chevrol Bel Air"/>,
  document.getElementById('main')
);
```

Vous pouvez également utiliser directement la fonction `createElement` de l'API `react` :

```
ReactDOM.render(
  React.createElement(
    Wine,
    {name: 'Château Chevrol Bel Air'}
  ),
  document.getElementById('main')
);
```

Build avec Webpack

Nous utilisons l'outil [Webpack](#) afin de construire notre application.

En complément de Webpack, nous utilisons [Babel](#), un compilateur Javascript qui permet d'utiliser les dernières nouveautés du langage. Dans notre cas, nous utilisons les plugins `react` et `es2015`.

Dans le fichier `package.json`, ajoutez les dépendances de développement nécessaires au build Webpack :

```
"devDependencies": {
  "webpack": "1.12.14",
  "babel-loader": "6.2.4",
  "babel-preset-es2015": "6.6.0",
  "babel-preset-react": "6.5.0"
}
```

vous pouvez évidemment les ajouter via la ligne de commande :

```
npm install --save-dev webpack@1.12.14 babel-loader@6.2.4 babel-preset-es2015
```

ici l'argument `--save-dev` indique que la dépendance doit être inscrite dans les dépendances du build et non du projet lui-même

Créez le fichier `webpack.config.js` permettant de configurer Webpack et Babel :

```
var webpack = require('webpack');

module.exports = {
  output: {
    path: './public/js/',
    publicPath: '/js/',
    filename: 'bundle.js'
  },
  entry: {
    app: ['./src/app.js']
```

```
        },
        resolve: {
            extensions: [ '', '.js', '.jsx' ]
        },
        module: {
            loaders: [
                {
                    test: /\.js/,
                    exclude: /node_modules/,
                    loader: 'babel',
                    query: {
                        presets: [ 'react', 'es2015' ]
                    }
                }
            ]
        }
    };
};
```

La configuration de Webpack est simple :

- Le point d'entrée est le fichier `src/app.js`
- Le fichier `bundle.js` est généré dans le répertoire `public/js`
- Le build exécute le lanceur `babel` avec les plugins `react` et `es2015`
 - *Remarque : les plugins babel peuvent également être définis dans le fichier de configuration `.babelrc`*

Vous pouvez ajouter les commandes Webpack sous forme de scripts dans le fichier `package.json`. Par exemple :

```
"scripts": {
    "bundle": "webpack -p --colors --progress"
}
```

Ainsi, la commande `npm run bundle` permet de construire le fichier `bundle.js`

Enfin, pensez à référencer le script `bundle.js` dans le fichier `index.html` :

```
<body>
<h1>Wines</h1>

<div id="main"></div>

<script src="/js/bundle.js"></script>
</body>
```

Exécution avec Webpack Dev Server

Afin de rendre la page `index.html` dans un navigateur, nous utilisons [Webpack Dev Server](#).

Ajoutez la dépendance à `webpack-dev-server` dans le fichier `package.json` :

```
"devDependencies": {
    "webpack-dev-server": "1.14.1"
}
```

ou via la commande `npm install --save-dev webpack-dev-server@1.14.1`

il faudra également rajouter un peu de configuration pour plus tard dans le fichier `webpack.config.js`

```
devServer: {  
    historyApiFallback: true,  
    proxy: {  
        '/api/*': {  
            target: 'http://localhost:3000'  
        }  
    }  
}
```

cette ajout ne nous est pas tout de suite utile, mais le deviendra dans les prochaines étapes. Il permet à notre serveur de développement de proxyfier tous les appels à `http://localhost:8080/api/*` vers `http://localhost:3000/api/*` qui se trouve être notre serveur de données sur les vins (d'ailleurs n'oubliez pas d'aller lancer ce serveur dans le dossier `api` via la commande `npm start`). Cette astuce permet d'éviter à avoir à gérer des appels `CORS` en développement. La configuration `historyApiFallback` permet au serveur de renvoyer la page d'index en cas de page non trouvée. Cette astuce nous permettra d'utiliser facilement l'API `history` du navigateur.

Ajoutez un nouveau script permettant de lancer le serveur Webpack :

```
"scripts": {  
    "start": "webpack-dev-server -d --colors --inline --content-base public"  
}
```

Lancez enfin la commande `npm start` et ouvrez la page `http://localhost:8080`.

Modifiez le code du composant `Wine` et observez les modifications en live dans votre navigateur !

Attention

Si vous souhaitez travailler sur un environnement déporté, il sera nécessaire de pouvoir accéder au `webpack-dev-server` depuis l'extérieur. Par défaut, `webpack-dev-server` n'écoute que `localhost`. Afin d'éviter ce comportement, vous pouvez spécifier sur quel host `webpack-dev-server` doit écouter. Changez le script `start` par :

```
"scripts": {  
    "start": "webpack-dev-server -d --colors --inline --content-base public -  
}
```

ESLint

[ESLint](#) est un outil qui permet d'analyser votre code Javascript selon un certains nombre de règles.

Dans notre cas, nous allons l'utiliser avec le plugin `eslint-plugin-react` qui propose des règles spécifiques au développement de composants `react`.

Pour commencer, ajoutez les dépendances nécessaires dans le fichier `package.json` :

```
"devDependencies": {
  "eslint": "2.4.0",
  "eslint-plugin-react": "4.2.3"
}
```

ou via la commande `npm install --save-dev eslint@2.4.0 eslint-plugin-react@4.2.3`

Créez ensuite le fichier `.eslintrc` qui permet de configurer ESLint :

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "plugins": [
    "react"
  ],
  "parserOptions": {
    "ecmaVersion": 6,
    "sourceType": "module",
    "ecmaFeatures": {
      "jsx": true,
      "experimentalObjectRestSpread": true
    }
  },
  "rules": {
    "react/display-name": 0,
    "react/forbid-prop-types": 1,
    "react/jsx-boolean-value": 1,
    "react/jsx-closing-bracket-location": 1,
    "react/jsx-curly-spacing": 1,
    "react/jsx-handler-names": 1,
    "react/jsx-indent-props": 1,
    "react/jsx-key": 1,
    "react/jsx-max-props-per-line": 1,
    "react/jsx-no-bind": 1,
    "react/jsx-no-duplicate-props": 1,
    "react/jsx-no-literals": 1,
    "react/jsx-no-undef": 1,
    "react/jsx-pascal-case": 1,
    "jsx-quotes": 1,
    "react/jsx-sort-prop-types": 1,
    "react/jsx-sort-props": 1,
    "react/jsx-uses-react": 1,
    "react/jsx-uses-vars": 1,
    "react/no-danger": 1,
    "react/no-did-mount-set-state": 1,
    "react/no-did-update-set-state": 1,
    "react/no-direct-mutation-state": 1,
    "react/no-multi-comp": 1,
    "react/no-set-state": 1,
    "react/no-unknown-property": 1,
    "react/prefer-es6-class": 0,
    "react/prop-types": 1,
    "react/react-in-jsx-scope": 1,
    "react/require-extension": 1,
    "react/self-closing-comp": 1,
    "react/sort-comp": 1,
    "react/wrap-multilines": 1
  }
}
```

- L'attribut `extends` permet d'hériter d'une configuration existante. `eslint:recommended` contient les règles recommandée par ESLint.
- La partie `env` permet de définir quelles variables globales sont potentiellement utilisées dans le code. Ici nous ajoutons celles du navigateur et celle de node.
- La partie `plugins` permet d'ajouter des plugins ESLint. Ici nous ajoutons le plugin `react`.
- La partie `ecmaFeatures` permet de définir les options du langage Javascript supportées lors de l'analyse. Ici nous activons la syntaxe JSX ainsi que les modules ES6.
- La partie `rules` permet de définir les règles à appliquer lors de l'analyse du code. Pour plus de détails sur les règles disponibles :
<https://www.npmjs.com/package/eslint-plugin-react>

Il est possible d'exclure certains fichiers ou dossiers de l'analyse, grâce au fichier `.eslintignore`. Exemple :

```
node_modules
webpack.config.js
public
```

Enfin, ajoutez un script dans le fichier `package.json` permettant d'exécuter ESLint grâce à la commande `npm run lint` :

```
"scripts": {
  "lint": "eslint src"
}
```

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)

Etape 1 - Testing

Pré-requis

L'étape 0 du workshop a permis de développer un composant `react` basique `Wine` :

```
const Wine = React.createClass({
  propTypes: {
    name: React.PropTypes.string
  },
  render() {
    return (
      <div className="wine" style={WineStyle}>
        {this.props.name}
      </div>
    );
  }
});
```

Le code disponible ici correspond au résultat attendu de l'étape 0. L'objectif de l'étape 1 est de le compléter pour ajouter le test unitaire du composant `Wine`.

Pour lancer l'application de l'étape 0, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur.

Test unitaire

Les tests unitaires sont primordiaux dans le développement. Ils ne doivent en aucun cas être négligés, c'est pourquoi nous les introduisons dès le début du workshop.

L'objectif est de mettre en place le test unitaire du composant `Wine`. Pour cela, nous allons nous appuyer sur les librairies suivantes :

- [react-addons-test-utils](#) : addon `react` facilitant les tests de composants `react`.
- [jsdom](#) : librairie implémentant les standards DOM et HTML, qui permettra de créer un document HTML dans lequel faire le rendu des composants à tester.
- [Chai](#) : librairie d'assertions, orientée BDD/TDD.
- [Mocha](#) : framework Javascript de tests unitaires.
- [babel-register](#) : permet d'utiliser Babel lors de l'exécution des tests avec Mocha.

Commencez par ajouter ces librairies au fichier `package.json` :

```
"devDependencies": {
  "babel-register": "6.7.2",
  "jsdom": "8.1.0",
  "mocha": "2.4.5",
  "chai": "3.4.1",
  "react-addons-test-utils": "0.14.7"
}
```

vous pouvez évidemment les ajouter via la ligne de commande :

Etape 2 - Développer une application react

Pré-requis

Vous devez maîtriser les étapes 0 et 1 du workshop afin de pouvoir réaliser l'étape 2.

Le code disponible dans cette étape correspond au résultat attendu des étapes 0 et 1. Vous pouvez partir de cette base pour développer l'étape 2.

Pour lancer l'application de l'étape 1, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur.

Dans cette étape, vous allez avoir besoin de l'API. Pour l'exécuter, lancez la commande `npm start` dans le dossier `api`.

Vous avez également la possibilité de lancer les tests de cette étape (que nous avons rédigé pour vous) en utilisant la commande `npm test` afin de voir quelles parties de l'étape fonctionnent et quelles parties ne fonctionnent pas du tout. N'hésitez pas à lire le code des tests afin d'avoir quelques indications en plus sur la façon d'écrire votre application.

Objectif

L'objectif de cette étape est de développer une application permettant d'afficher la fiche descriptive d'un vin, chaque vin étant catégorisé dans une région viticole.

Voici une maquette de l'application :

- Une première colonne listant les régions viticoles. Chaque région est sélectionnable par un clic.
- Une deuxième colonne listant les vins de la région viticole sélectionnée. Chaque vin est sélectionnable par un clic.
- Une troisième colonne affichant la fiche descriptive du vin sélectionné, contenant :
 - Le nom du vin
 - Le type de vin (rouge, blanc, rosé, etc ...)
 - L'appellation du vin (nom de l'appellation et région)
 - La liste des cépages du vin
 - Une image du vin

Wines

Regions Wine List Wine Description

Region #1

Wine #A

Region #2

Wine #B

Region #3

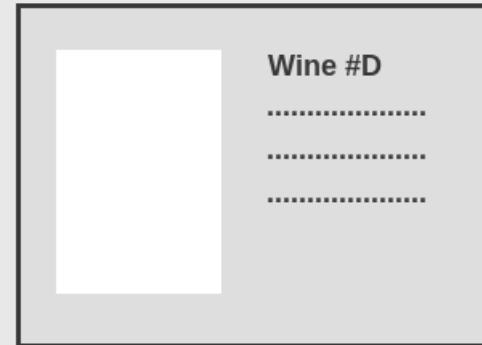
Wine #C

Region #4

Wine #D

Wine #E

Wine #F



Avant de partir la tête baissée dans le développement, il est nécessaire de se poser quelques questions :

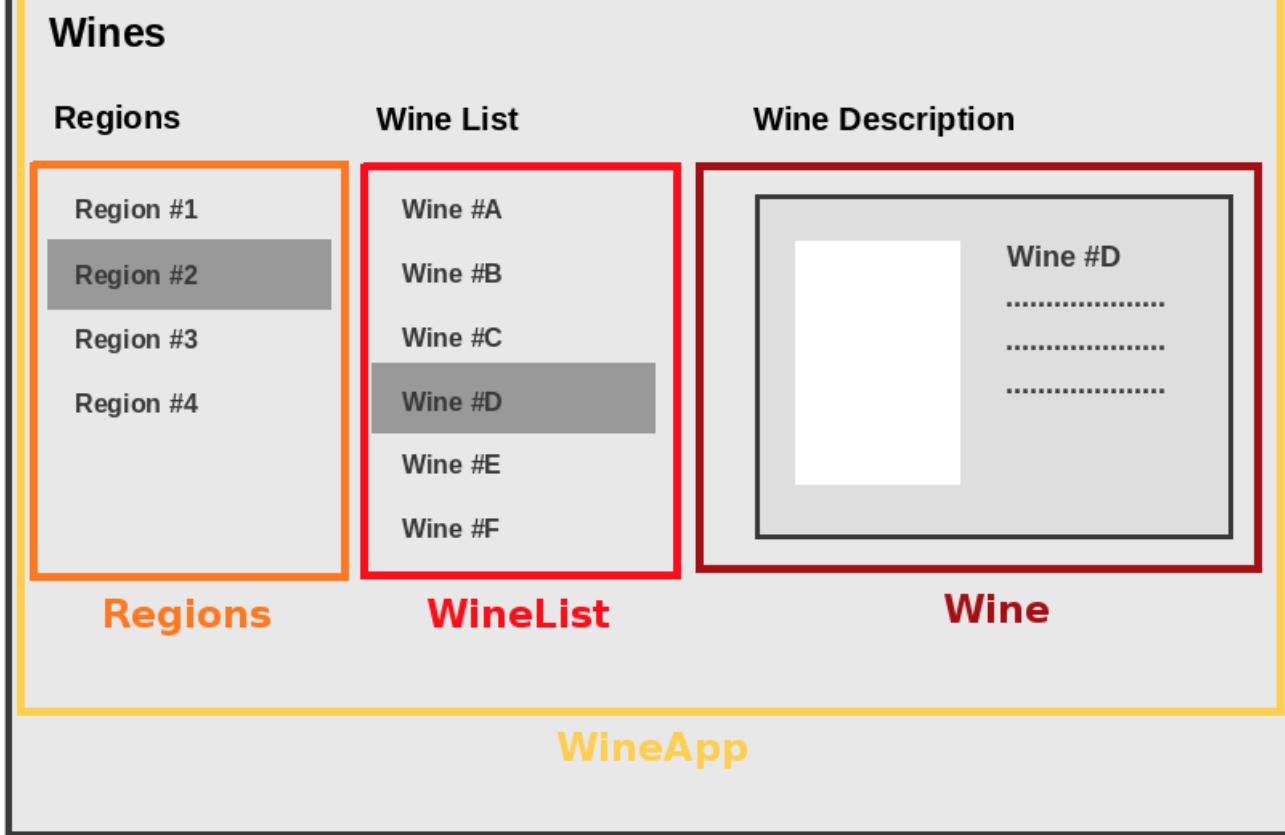
- Quels sont les composants de mon application et comment sont-ils liés ?
- Quelles sont les données gérées par mes composants ? Ces données sont-elles gérées via les `props` ou via le `state` ?
- Quelles sont les interactions entre mes composants ? Quels événements doivent-être gérés ?
- Ok, je suis prêt pour développer, mais par quoi je commence ?

Hiérarchie de composants

La première chose à faire est de mettre son cerveau en **mode React** et de **penser composants** :-)

A partir de la maquette fournie, nous pouvons identifier les principaux composants de notre application :

- `Regions` : le composant gérant la liste des régions,
- `WineList` : le composant gérant la liste des vins,
- `Wine` : le composant gérant la fiche descriptive d'un vin,
- `WineApp` : le composant parent permettant d'assembler les autres composants, qui représente au final notre application.



Props et state

Les données gérées par l'application sont les suivantes :

- la liste des régions viticoles,
- la référence à la région sélectionnée dans la liste des régions,
- la liste des vins de la région viticole sélectionnée,
- la description du vin sélectionné dans la liste des vins.

Il reste à savoir de quelle manière ces données doivent être gérées au niveau de chaque composant : via les `props` ou via le `state` ?

Les `props` d'un composant sont immutables contrairement au `state` qui lui est mutable. Il est donc généralement conseillé de définir deux types de composants dans une application `react` :

- Les composants dédiés purement à la présentation, qui s'appuieront uniquement sur leur `props`. Nous les surnommerons `Dumb components`.
- Les composants plutôt orientés "conteneurs", qui sont responsables du fonctionnement de l'application, qui s'appuient fortement sur leur `state`. Nous les surnommerons `Smart components`.

Vous pouvez lire un article très intéressant sur le sujet : [Smart and Dumb Components](#)

Dans notre cas, `Regions`, `WineList` et `Wine` sont des composants orientés présentation et utiliseront uniquement leurs `props`. `WineApp` est le conteneur et utilisera donc son `state`.

Interactions entre les composants

Les interactions entre les composants sont les suivantes :

- La sélection d'une région provoque la mise à jour de la liste des vins (affichage de la liste des vins de la régions sélectionnée) et de la description du vin (affichage de la description du premier vin de la liste)
- La sélection d'un vin provoque la mise à jour de la description du vin.

Par où commencer le développement ?

Par expérience, une bonne façon de démarrer le développement d'une application react est la suivante :

- Construire une version statique de l'application en créant l'ensemble des composants définis lors des précédentes étapes de conception.
- Définir l'état initial des composants de type conteneur et utiliser cet état dans le rendu des composants.
- Charger dynamiquement les données depuis l'API (appels Ajax) et mettre à jour l'état des composants.
- Gérer les événements pour rendre l'application interactive.

Version statique de l'application

Dumb

Créez l'ensemble des composants de type présentation: Regions , WineList , et Wine et implémentez la méthode render() de chacun d'eux, en vous appuyant sur les props .

Par exemple pour le composant Regions :

```
const Regions = React.createClass({
  render () {
    return (
      <div>
        { this.props.regions.map(region => <div key={region}>{region}</div>)
      )
    }
}

export default Regions
```

Lors du rendu d'une liste de composants, react a besoin d'une propriété unique key sur chaque composant de la liste. Pour plus de détails techniques, [lisez la documentation](#)

PropTypes

Il est possible de définir plus précisément le format attendu dans les props d'un composant, grâce aux [PropTypes](#).

Par exemple sur notre composant Wine qui gère l'affichage de la description d'un vin :

```
import React, { PropTypes } from 'react';

const Wine = React.createClass({
```

```

propTypes: {
  wine: PropTypes.shape({
    id: PropTypes.string,
    name: PropTypes.string,
    type: PropTypes.oneOf(['Rouge', 'Blanc', 'Rosé', 'Effervescent', 'Moell']),
    appellation: PropTypes.shape({
      name: PropTypes.string,
      region: PropTypes.string
    }),
    grapes: PropTypes.arrayOf(PropTypes.string)
  })
},
// ...
}

```

Définissez les `propTypes` de chaque composant, en vous appuyant sur le format des données remontées par l'API.

Attention : les `propTypes` ne sont vérifiées qu'en mode développement pour aider le développeur à utiliser correctement les composants.

Smarts

Créez le composant conteneur `WineApp` qui permet d'assembler l'ensemble des composants.

C'est ce composant qui sera utilisé pour effectuer le rendu de l'application dans le DOM, dans le fichier `app.js` :

```

import React from 'react';
import ReactDOM from 'react-dom';

import WineApp from './components/wine-app';

ReactDOM.render(
  <WineApp />,
  document.getElementById('main')
);

```

Afin de pouvoir tester rapidement de rendu de l'application, vous pouvez utiliser des valeurs "en dur" pour alimenter les `props` des composants. Par exemple :

```

const WineApp = React.createClass({
  render() {
    return (
      ...
      <Regions regions={[ "Bordeaux", "Bourgogne" ]} />
      ...
    )
  }
})

```

Etat initial des composants

L'étape suivante consiste à définir l'état initial des composants de type conteneur, donc `WineApp` dans notre cas.

Cela se fait via la méthode `getInitialState()` du composant :

```

const WineApp = React.createClass({
  getInitialState() {
    return {
      regions: [],
      selectedRegion: null,
      wines: [],
      selectedWine: null
    };
  },
  // ...
})

```

Utilisez ensuite le `state` dans le rendu. Par exemple pour alimenter la liste des régions :

```

const WineApp = React.createClass({
  // ...
  render() {
    return (
      ...
      <Regions regions={this.state.regions} />
      ...
    )
  }
})

```

Récupération des données via l'API

Les données doivent être récupérées en Ajax au travers de l'API mise à disposition.

Dans un composant `react`, les appels Ajax se font généralement dans la méthode `componentDidMount()` du [cycle de vie du composant](#).

Nous utilisons `fetch` pour effectuer les appels Ajax. Par exemple pour charger la liste des régions :

```

const WineApp = React.createClass({
  // ...
  componentDidMount() {
    fetch('/api/regions')
      .then(r => r.json())
      .then(data => {
        this.setState({
          regions: data,
          selectedRegion: data[0]
        });
        // TODO Charger les vins de la région : this.loadWinesByRegion(data)
      })
      .catch(response => {
        console.error(response);
      });
  },
  // ...
})

```

La méthode `setState()` permet de mettre à jour l'état du composant, ce qui provoque son re-rendu.

Complétez le code du composant `WineApp` afin de gérer l'ensemble des appels Ajax permettant d'alimenter correctement le `state`.

Gestion des événements

La dernière étape consiste à gérer les événements afin de rendre l'application interactive.

Ce sont les composants de type présentation qui vont capter les événements. Mais ce ne sont pas eux qui effectueront le traitement lié aux événements : ils se contenteront de déléguer ce traitement au composant parent, via une fonction de type "handler" passée via les `props`.

Par exemple pour le composant `Regions`, il faut déterminer ce que l'on fait lorsque l'utilisateur sélectionne une région (événement `onClick`).

Au niveau du composant `Regions` cela donne simplement :

```
const Regions = React.createClass({
  handleRegionClick(event) {
    this.props.onRegionChange(event.target.textContent);
  },

  render () {
    return (
      <div>
        {
          this.props.regions.map(region =>
            <div key={region} onClick={this.handleRegionClick}>
              {region}
            </div>
          )
        }
      </div>
    )
  }
})
```

Et au niveau du composant `WineApp` :

```
const WineApp = React.createClass({
  // ...
  handleRegionChange(region) {
    this.setState({
      selectedRegion: region
    });
    // TODO Recharger les vins de la région sélectionnée : this.loadWinesByRe
  },

  render() {
    return (
      ...
      <Regions regions={this.state.regions} onRegionChange={this.handleRegionC...
      ...
    )
  }
})
```

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Pour le style de l'application, ne vous prenez pas la tête, l'enjeu n'est pas là :-) Pour

L'organisation en colonne, vous pouvez utiliser le framework CSS [avalanche](#). Le CSS est disponible dans le dossier `public/css/avalanche.css`.

Exemple de grille avec Avalanche :

```
<div className="grid">
  <div className="1/4 grid__cell">
    <h2>Regions</h2>
    <!-- ... -->
  </div>
  <div className="1/3 grid__cell">
    <h2>Wine List</h2>
    <!-- ... -->
  </div>
  <div className="5/12 grid__cell">
    <h2>Wine Description</h2>
    <!-- ... -->
  </div>
</div>
```

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)

Etape 3 : SPA et react-router

Pré-requis

Vous devez maîtriser les étapes 0, 1 et 2 du workshop afin de pouvoir réaliser l'étape 3.

Le code disponible dans cette étape correspond au résultat attendu des étapes 0, 1 et 2. Vous pouvez partir de cette base pour développer l'étape 3.

Pour lancer l'application de l'étape 2, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur.

Dans cette étape, vous allez avoir besoin de l'API. Pour l'exécuter, lancez la commande `npm start` dans le dossier `api`. La documentation de l'API est disponible à l'adresse <http://localhost:3000>

Vous avez également la possibilité de lancer les tests de cette étape (que nous avons rédigé pour vous) en utilisant la commande `npm test` afin de voir quelles parties de l'étape fonctionnent et quelles parties ne fonctionnent pas du tout. N'hésitez pas à lire le code des tests afin d'avoir quelques indications en plus sur la façon d'écrire votre application.

Objectif

Maintenant que notre application possède les fonctionnalités de base, nous allons commencer à nous attaquer à la navigation.

En effet pour le moment notre application regroupe toutes les fonctionnalités dans un seul et même écran, ce qui s'avère très pratique d'un point de vue technique mais qui n'est pas génial d'un point de vue utilisateur.

Nous allons donc découper notre application en 3 écrans successifs, permettant de choisir la région d'un vin,

puis le vin à consulter

et enfin la fiche de consultation du vin.

Ici chaque sélection entraînera la navigation jusqu'à l'écran suivant, etc ...

Mais comme nous sommes dans un contexte frontend et que nous ne sommes pas là pour coder du backend, nous allons développer tout ça sous forme d'une SPA (Single Page Application).

Une SPA est une application web (front) accessible via une page unique. Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée et donc de fluidifier l'expérience utilisateur.

Il va donc nous falloir un moyen de router l'utilisateur à travers divers écrans.

Un moyen simple pourrait être d'avoir un composant technique au plus haut niveau de notre application pour gérer la navigation. Ce composant gérerait la pile d'appel et la vue courante dans son `state` et proposerait une API pour être piloté depuis les diverses vues.

Par exemple, nous pourrions définir un composant comme suivant :

```
const Navigator = React.createClass({
  propTypes: {
    initialRoute: React.PropTypes.shape({
      component: React.PropTypes.func.isRequired,
      title: React.PropTypes.string,
      props: React.PropTypes.object,
    }).isRequired,
  },
  getInitialState() {
    return {
      component: null,
      title: null,
      props: null,
    };
  },
  componentDidMount() {
    this.setState({
      component: this.props.initialRoute.component,
      title: this.props.initialRoute.title,
      props: this.props.initialRoute.props,
    });
  },
  navigateTo({ component, title, props }) {
    this.setState({ component, title, props });
  },
  render() {
    const Component = this.state.component;
    const { title, props } = this.state;
    return (
      <Component
        {...props}
        navTitle={title}
        navigator={{ navigateTo: this.navigateTo }} />
    );
  }
});

const Page2 = React.createClass({
  ...
});

const Page1 = React.createClass({
  gotoNext() {
    this.props.navigator.navigateTo({
      title: 'Page 2',
      component: Page2,
      props: {
        foo: 'bar',
      },
    });
  },
  render() {
    return (
      <div>
        <h2>Hello World!</h2>
        <button onClick={this.gotoNext}>Next</button>
      </div>
    );
  }
});
```

```

        }
    });

ReactDOM.render(
    <Navigator initialRoute= {{ title: 'Page 1', component: Page1 }} />,
    document.getElementById('main')
);

```

Cependant, ce genre d'approche a l'inconvénient de perdre la navigation courante lorsque l'on recharge la page. Du coup il existe de meilleures solutions, notamment, [`react-router`](#) que nous allons utiliser pour gérer la navigation de notre application.

Commençons par ajouter une dépendance pour `react-router` dans l'application.

Dans le fichier `package.json` ajoutez la dépendance suivante :

```

"dependencies": {
    ...
    "react-router": "2.0.1",
    ...
}

```

vous pouvez évidemment l'ajouter via la ligne de commande :

```
npm install --save react-router@2.0.1
```

Maintenant nous pouvons commencer l'intégration du router (l'intégration de base est présente dans le projet mais vous pouvez tout de même lire les paragraphes suivant).

Pour ce faire, commençons par lire [l'introduction](#) à `react-router` puis importons les APIs dans `src/app.js`. Pour des raisons de testabilité, nous allons faire en sorte d'encapsuler toute l'application et son système de routage dans un composant dédié dans `src/app.js` et faire en sorte que ce composant puisse recevoir une API `history` dédiée (différente en environnement de test). La montage de ce composant dans le DOM sera effectué dans le fichier `src/index.js`

```
import { Router, Route, browserHistory, IndexRoute } from 'react-router';
```

l'initialisation du routeur se fera de la façon suivante :

```

import { Router, Route, browserHistory, IndexRoute } from 'react-router';
import NotFound from './components/not-found';

export const App = React.createClass({
  propTypes: {
    history: PropTypes.object,
  },
  render() {
    const history = this.props.history || browserHistory;
    return (
      <Router history={history}>
        <Route path="/" component={???}>
          <IndexRoute component={???} />
        ...
        <Route path="*" component={NotFound} />
      </Route>
    );
  }
});

```

```
    }  
});
```

Ici nous configurons le routeur pour utiliser l'API `history`, tirée de HTML5, du navigateur comme URL de routage côté client. Cette API permet de faire varier l'URL du navigateur sans pour autant déclencher un rechargement de la page.

```
const history = this.props.history || browserHistory;  
<Router history={history}> ... </Router>
```

puis nous spécifions un container qui aura le rôle d'afficher la vue courante du router et qui sera le point d'entrée de l'application.

```
<Route path="/" component={???}> ... </Route>
```

D'après le tutorial de `react-router`, ce genre de composant peut s'écrire de la façon suivante :

```
const MyApp = React.createClass({  
  render() {  
    return (  
      <div>  
        <h1>App</h1>  
        {this.props.children}  
      </div>  
    )  
  }  
});
```

ensuite nous spécifions la vue à afficher pour une navigation vers `/`

```
<IndexRoute component={???} />
```

et enfin nous spécifions une route permettant d'attraper tous les appels n'ayant pu être routés

```
<Route path="*" component={NotFound} />
```

Pour notre application, nous vous proposons de respecter les schémas d'url suivants :

- `/` => vue des régions
- `/regions/:regionId` => vue des vins de la région
- `/regions/:regionId/wines/:wineId` => vue du vin sélectionné

Ce routage est défini dans le routeur via l'utilisation du composant

```
<Route path="/mon/path" component={MonComponent} />
```

Pour passer des paramètres aux routes et les récupérer, vous pouvez déclarer vos routes comme ceci

```

<Route path="/mon/path/:monId" component={MonComponent} />

const MonComponent = React.createClass({
  render() {
    return (
      <div>Valeur de monId: {this.props.params.monId}</div>
    );
  }
})

```

enfin vous pouvez créer des liens en utilisant l'API

```

import { Link } from 'react-router';

...
<Link to="/mon/path/1234">Chose 1234</Link>

```

de `react-router` ou en utilisant directement l'API `history` disponible à travers le contexte `react`.

Pour cela, il est nécessaire de spécifier sur le composant routé, un `contextType` afin de valider le contenu du contexte

```

export const Page = React.createClass({

  contextTypes: {
    router: React.PropTypes.object
  },

  handleNavigationTo(path) {
    // ici on déclenche la navigation vers l'url /view/${path}/details
    this.context.router.push({
      pathname: `/view/${path}/details`
    });
  },

  render() {
    ...
  }
}

```

Un dernier petit conseil, vos composants existent déjà et sont idiots (Dumb Components). Ce qui veut dire qu'ils n'ont pas d'état propre, et fonctionnent uniquement via les propriétés qui leur sont passés. Autrement dit, ce sont des composants stateless.

Ce genre d'approche est plutôt intéressante car elle permet de bien séparer ce genre de composants des composants intelligents (Smart Components) qui eux sont souvent stateful et technique sans forcément produire des éléments graphiques. (voir cet [article](#) sur le sujet)

Dans le cadre de notre application, il serait intéressant de garder nos composants graphiques simple tel qu'ils sont, et les wrapper dans des composants intelligents qui se chargeront des appels HTTP et de la gestion de l'état

- `RegionsPage => Regions`
- `WinelistPage => Winelist`
- `WinePage => Wine`

Les composants `RegionsPage`, `WinelistPage` et `WinePage` sont mis à votre disposition dans les fichiers `src/components/regions.js`, `src/components/wine-list.js`, `src/components/wine.js`

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Bonus

A priori vous avez créé un composant type `container` lié à l'URL `/` qui a pour seul rôle de contenir les différentes pages de l'application. Il pourrait être intéressant que ce composant affiche le titre courant de la vue (`Regions` -> `Wines from Bordeaux` -> `Cheval Noir`). Pour cela une petite astuce, le meilleur moyen à votre disposition est de rajouter une fonction de mise à jour du titre courant dans les props de la vue rendue à l'intérieur du container principal. Il vous est donc possible de cloner l'élément à render et de lui rajouter des propriétés de la façon suivante :

```
const WineApp = React.createClass({
  render () {
    return (
      <div className="grid">
        <div className="1/2 grid__cell">
          {this.props.children && React.cloneElement(this.props.children, {
            uneNouvelleProps: 'Hello World!'
          })}
        </div>
      </div>
    );
  }
});
```

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)

Etape 4 - Ajout de fonctionnalités

Pré-requis

Vous devez maîtriser l'étape 3 du workshop afin de pouvoir réaliser l'étape 4.

Le code disponible dans cette étape correspond au résultat attendu de l'étape 3. Vous pouvez partir de cette base pour développer l'étape 4.

Pour lancer l'application de l'étape 3, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur.

Dans cette étape, vous allez avoir besoin de l'API. Pour l'exécuter, lancez la commande `npm start` dans le dossier `api`. La documentation de l'API est disponible à l'adresse <http://localhost:3000>

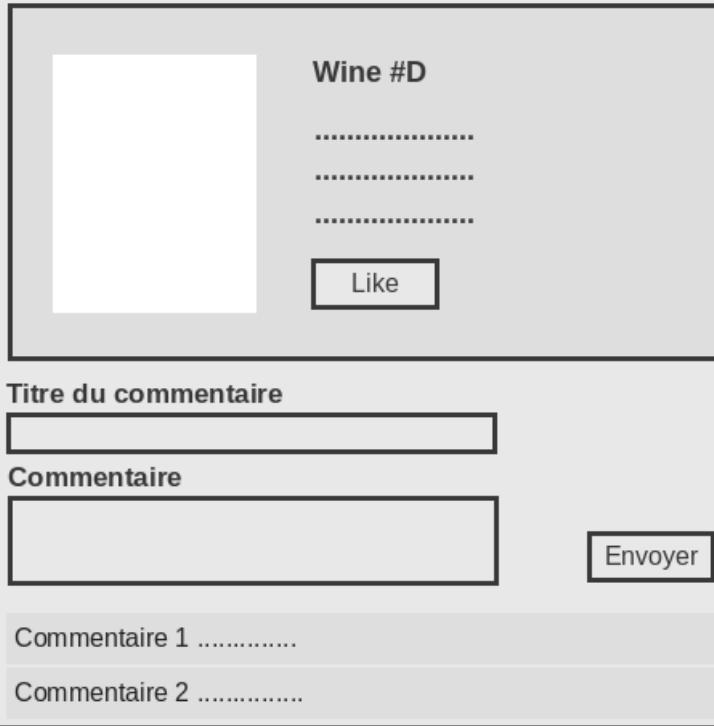
Vous avez également la possibilité de lancer les tests de cette étape (que nous avons rédigé pour vous) en utilisant la commande `npm test` afin de voir quelles parties de l'étape fonctionnent et quelles parties ne fonctionnent pas du tout. N'hésitez pas à lire le code des tests afin d'avoir quelques indications en plus sur la façon d'écrire votre application.

Objectif

L'objectif de cette étape est d'ajouter des fonctionnalités sur la fiche de description d'un vin, à savoir :

- Aimer / ne plus aimer le vin
- Visualiser les commentaires sur le vin
- Ajouter un nouveau commentaire

Wine Description



Like

Les composants impactés par la fonctionnalité "Aimer / ne plus aimer" un vin sont `Wine` et `WinePage`.

Faites évoluer le composant `Wine` :

- Ajout d'une nouvelle propriété : `liked` .
- Ajout d'un bouton "like / unlike" dans le rendu du composant se basant sur la valeur de la propriété `liked` .
- Gestion de l'événement `onClick` sur le bouton.

Faites évoluer le composant `WinePage` :

- Ajout dans le `state` d'un attribut `liked` .
- Evolution du code de la méthode `componentDidMount()` pour aller chercher la valeur de `liked` sur le serveur.
- Gestion du click sur le bouton "like / unlike" du composant `Wine` dans une nouvelle méthode `handleToggleLike()` .

Comment

Créez un nouveau composant `Comments` permettant :

- d'afficher la liste des commentaires d'un vin,
- d'ajouter un nouveau commentaire sur le vin.

Ce composant sera directement utilisé dans le rendu du composant `WinePage`, de la manière suivante :

```
render() {
  // ...
  return (
    <div>
      <Wine ... />
      <Comments wineId={this.state.wine.id} />
    </div>
  );
}
```

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)

Step 5 : redux

Pré-requis

Vous devez maîtriser les étapes 0, 1, 2, 3 et 4 du workshop afin de pouvoir réaliser l'étape 5.

Le code disponible dans cette étape correspond au résultat attendu des étapes 0, 1, 2, 3 et 4. Vous pouvez partir de cette base pour développer l'étape 5.

Pour lancer l'application de l'étape 4, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur.

Dans cette étape, vous allez avoir besoin de l'API. Pour l'exécuter, lancez la commande `npm start` dans le dossier `api`. La documentation de l'API est disponible à l'adresse <http://localhost:3000>

Vous avez également la possibilité de lancer les tests de cette étape (que nous avons rédigé pour vous) en utilisant la commande `npm test` afin de voir quelles parties de l'étape fonctionnent et quelles parties ne fonctionnent pas du tout. N'hésitez pas à lire le code des tests afin d'avoir quelques indications en plus sur la façon d'écrire votre application.

Objectif

Dans cette étape, nous allons intégrer [Redux](#) à notre magnifique application.

[Redux](#) est un container d'état global pour des applications JavaScript. Redux peut être utilisé dans n'importe quel environnement et ne dépend pas de `react`. Redux possède quelques propriétés très intéressantes en matière de cohérence, de prévisibilité et d'expérience du développeur.

Redux peut être vu comme une implémentation du [pattern Flux](#) avec cependant quelques variations, notamment au niveau de la complexité de mise en place bien moindre que Flux. Redux s'inspire également de patterns propre au langage Elm.

Redux fournit un concept de `store` de données unique pour l'application (singleton), auquel nos composants vont s'abonner. Il est ensuite possible de dispatcher des `actions` sur ce `store` qui déclencheront une mutation de l'état contenu dans le `store`. Une fois la mutation effectuée, le `store` notifiera tous ses abonnés du changement d'état. L'intérêt d'un tel pattern devient évident lorsqu'une application grossit et que plusieurs composants `react` ont besoin d'une même source de données. Il est alors plus simple de gérer l'état *fonctionnel* de l'application en dehors des composants et de s'y abonner.

Pour fonctionner Redux utilise une notion de `reducer` qui fonctionne exactement de la même façon qu'une fonction de réduction sur une collection. Si on visualise l'état de l'application comme une collection de mutations, le `reducer` est simplement la fonction qui prend en paramètre l'état précédent et retourne le prochain état via un second paramètre qui dans notre cas est une `action`. Un `reducer` est donc une fonction pure avec la signature suivante `(state, action) => state` qui décrit comment une action transforme l'état courant en un nouvel état.

Regardons un exemple simple

```
import { createStore } from 'redux'

/**
 * Ici nous avons un unique reducer qui va gérer un état de type number.
 * On note que l'état initial est fourni via les paramètres par défaut
 * On utilise un switch pour gérer les différentes actions,
 * mais ce n'est absolument pas obligatoire
 * A noter qu'il est impératif lors d'une mutation d'état de
 * renvoyer un nouvel état et non l'ancien état muté.
 */
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

// Ici nous créons un store pour notre application
// se basant sur le reducer `counter`
// l'API du store est la suivante { subscribe, dispatch, getState }.
let store = createStore(counter)

// Maintenant nous pouvons nous abonner aux modifications
// de l'état contenu dans le store.
// Un cas d'utilisation pourrait être un composant react qui
// veut afficher l'état courant du compteur.
// On note que le nouvel état n'est pas véhiculer dans le
// callback mais qu'il faut aller le chercher
// directement sur le store.
store.subscribe(() =>
  console.log(store.getState())
)

// Maintenant, le seul moyen de muter l'état interne
// du store est de lui envoyer une action
store.dispatch({ type: 'INCREMENT' })
// afficher 1 dans la console
store.dispatch({ type: 'INCREMENT' })
// afficher 2 dans la console
store.dispatch({ type: 'DECREMENT' })
// afficher 1 dans la console
```

Il est bien évidemment possible d'avoir plusieurs `reducers`, ou de faire en sorte qu'un `reducer` ne soit en charge que d'une partie de l'état global, etc ...

Pour plus de détails et explications sur Redux, vous pouvez consulter la [documentation de la librairie](#) qui est très bien faite.

Mise en pratique

Commencez par installer `redux` et `react-redux`. Dans le fichier `package.json` ajoutez les dépendances suivantes :

```
"dependencies": {
  ...
  "react-redux": "4.4.1",
  "redux": "3.3.1",
```

```
}
```

ou via la ligne de commande

```
npm install --save redux@3.3.1 react-redux@4.4.1
```

Dans le cadre de notre application, nous allons afficher des informations statistiques globales concernant notre application. A savoir le nombre global de likes et le nombre global de commentaires. Ces données seront mises à jour en temps réel.

L'état de notre application contenu dans un store `redux` sera le suivant

```
{
  "comments": 42,
  "likes": 42
}
```

Nous allons donc avoir besoin de deux reducers, chacun gérant respectivement le compteur de commentaires et le compteur de likes.

Nous allons également avoir besoin d'actions permettant de muter notre état applicatif.

Globalement, pour chacun des compteurs nous avons besoin d'une action initiale qui sera lancée après un appel HTTP et qui changera l'état pour lui donner une valeur initiale provenant du serveur. Ensuite nous aurons besoin d'une action permettant d'incrémenter le compteur et d'une action permettant de décrémenter le compteur.

Vous pouvez maintenant implémenter toutes vos actions dans un fichier `src/actions/index.js` tel que suivant (ici, pour un exemple de compteur classique)

```
export const incrementCounter = () => {
  return {
    type: 'INCREMENT_COUNTER',
    incrementValue: 2
  };
}

export const decrementCounter = () => {
  return {
    type: 'DECREMENT_COUNTER',
    decrementValue: 1
  };
}
```

puis vous pouvez créer vos reducers dans des fichiers respectivement nommés `src/reducers/comments.js` et `src/reducers/likes.js`. Chaque `reducer` générera un compteur avec une valeur initiale à 0.

Il s'agit maintenant de créer un `reducer` global, pour cela commencez par créer un fichier `src/reducers/index.js` avec le contenu suivant :

```
import { combineReducers } from 'redux';
import comments from './comments';
```

```

import likes from './likes';

const reducer = combineReducers({
  comments: comments,
  likes: likes
});

export default reducer;

```

ici la fonction `combineReducers` permet de créer un reducer global à partir de différents reducers responsables de différentes parties de l'état global, dans notre cas `comments` et `likes`. N'oubliez pas de créer vos fichiers `src/reducers/likes` et `src/reducers/comments` contenant respectivement des reducers publiques (exportés) nommés `likes` et `comments`.

Il ne nous reste plus qu'à créer le store de notre application, par exemple dans le fichier `src/app.js`.

```

import { createStore } from 'redux';
import app from './reducers';

const store = createStore(app);

```

il ne reste plus qu'à faire deux appels HTTP aux apis

- `/api/likes`
- `/api/comments`

afin d'initialiser le store avec les bonnes valeurs.

```

import { createStore } from 'redux';
import app from './reducers';
import { setCounterValue } from './actions';

const store = createStore(app);

fetch(`/api/count`)
  .then(r => r.json())
  .then(r => store.dispatch(setCounterValue(r.count)));

```

Nous avons maintenant un état global correctement alimenté. Cependant lorsqu'il est nécessaire de faire beaucoup d'asynchrone avec `redux`, il devient utile d'utiliser [redux-thunk](#) permettant de faire de l'inversion de contrôle au niveau des actions dispatchées au `store` et donc de dispatcher plusieurs fois ou de manière conditionnelle pour une même action.

Maintenant, il ne nous reste qu'à le connecter à l'UI

react-redux

Connecter un composant react à notre store est finalement très simple. Il suffit simplement d'abonner le composant au store une fois monté dans le DOM, le désabonner lors de sa disparition du DOM et mettre à jour son état à chaque notification du store. Par exemple, pour un composant lambda :

```

const Component = React.createClass({
  propTypes: {

```

```

    store: PropTypes.shape({
      subscribe: PropTypes.func,
      getState: PropTypes.func
    })
  },
  getInitialState() {
    return {
      counter: 0
    };
  },
  componentDidMount() {
    // lorsque l'on monte le composant dans le DOM, on souscrit aux notifications
    // afin de savoir lorsque celui-ci est mis à jour
    this.unsubscribe = this.props.store.subscribe(this.updateViewFromRedux);
  },
  componentWillUnmount() {
    // lorsqu'on démonte le composant du DOM, on annule la souscription aux notifications
    // pour ne pas mettre à jour un composant qui n'existe plus
    this.unsubscribe();
  },
  updateViewFromRedux() {
    const { counter } = this.props.store.getState(); // on récupère l'état du store
    this.setState({ counter });
  },
  render() {
    return (
      <div>
        <span>counter : {this.state.counter}</span>
      </div>
    );
  }
});

```

Cependant tout ce boilerplate peut se révéler fastidieux et rébarbatif à écrire à la longue. Pour éviter tout ce code inutile, la librairie `react-redux` permet de fournir un store à un arbre de composants `react` et de connecter un composant `react` à ce store, voire même de mapper automatiquement des propriétés de l'état du `store` sur des propriétés du composant.

La première chose à faire est de fournir le store à notre arbre de composants. Nous allons utiliser un composant `<Provider store={...} />` fourni par `react-redux` que nous allons placer à la racine de l'application dans le fichier `src/app.js`

```

import React from 'react';
import ReactDOM from 'react-dom';

import WineApp from './components/wine-app';
import { RegionsPage } from './components/regions';
import { WineListPage } from './components/wine-list';
import { WinePage } from './components/wine';
import { NotFound } from './components/not-found';

import { Router, Route, browserHistory, IndexRoute } from 'react-router';

import { Provider } from 'react-redux';
import { createStore } from 'redux';
import app from './reducers';

const store = createStore(app);

...

export const App = React.createClass({
  propTypes: {
    history: PropTypes.object
  }
});

```

```

},
render() {
  const history = this.props.history || browserHistory;
  return (
    <Provider store={store}>
      <Router history={history}>
        <Route path="/" component={WineApp}>
          <IndexRoute component={RegionsPage} />
          <Route path="regions/:regionId" component={WineListPage} />
          <Route path="regions/:regionId/wines/:wineId" component={WinePage}>
            <Route path="*" component={NotFound} />
          </Route>
        </Router>
      </Provider>
    );
}
);
}
);

```

Provider va simplement alimenter le context react avec un champ store . (pour plus de détails sur le contexte react , voir <https://facebook.github.io/react/docs/context.html>)

Il faut maintenant être capable de récupérer le store dans un composant afin de pouvoir

- dispatcher des actions
- récupérer l'état du store

Pour cela, redux propose une fonction connect permettant de créer un composant wrapper (Higher Order Component) qui fera le lien entre le store présent dans le contexte react et le composant wrappé. Par exemple, si dans un composant quelconque (le composant ci-dessous est fourni à titre d'exemple) nous voulons dispatcher une action react :

```

import { incrementCounter } from '../actions';
import { connect } from 'react-redux';

const SimpleComponent = React.createClass({
  propTypes: {
    dispatch: PropTypes.func
  },
  handleButtonClick() {
    dispatch(incrementCounter(42));
  },
  render() {
    ...
  }
});

export const ConnectedToStoreComponent = connect()(SimpleComponent);

```

ici le fait d'appeler connect avec le composant original en paramètre créé une nouvelle classe de composant comportant la logique d'abonnement et mise à jour du composant nécessaire à redux . Lorsque connect est appelé sans premier argument, le composant wrappé se voit ajouter une propriété dispatch permettant de dispatcher une action sur le store. Dans le cas de notre application, les deux composants qui auraient besoin d'accéder seulement au dispatch du store sont bien évidemment, les composants WinePage (pour gérer le like) et Comments (pour gérer l'ajout de commentaires).

Cependant en utilisant seulement connect()(Composant) , il n'est pas possible de récupérer l'état courant du store. Pour ce faire, il est nécessaire de spécifier une fonction

de mapping (`connect(mapStateToProps)(Composant)`) permettant d'extraire un ensemble de propriétés depuis l'état du store `redux` pour qu'elles soient ajoutées aux propriétés du composant wrappé (ici encore, le composant suivant est fourni à titre d'exemple) :

```
import { incrementCounter } from '../actions';
import { connect } from 'react-redux';

const SimpleComponent = React.createClass({
  propTypes: {
    dispatch: PropTypes.func,
    simpleCounter: PropTypes.number
  },
  handleButtonClick() {
    this.props.dispatch(incrementCounter(42));
  },
  render() {
    return (
      <div>
        <span>Clicked : {this.props.counter} </span>
        <button type="button" onClick={this.handleButtonClick}>+1</button>
      </div>
    );
  }
});

const mapStateToProps = (state, ownProps) => {
  return {
    simpleCounter: state.counter
  }
};

export const ConnectedToStoreComponent = connect(mapStateToProps)(SimpleComp
```

Il ne nous reste donc plus qu'à émettre les différentes actions au bon moment pour muter l'état de notre `store` (ajout de commentaire, ajout de like, retrait de like) et de rajouter un composant global (`<GlobalStats>` dans `<WineApp>` par exemple) affichant le nombre global de likes et de commentaires dans l'application. C'est ce composant qui aura besoin d'une fonction type `mapStateToProps` afin de récupérer les données nécessaire depuis le `store`.

Au final, l'arbre de composants effectifs pour ce genre d'application sera le suivant :

```
const store = ...
const mapStateToProps = (state) => ({ simpleCounter: state.counter });
...
<Provider store={store}>
  // Connector est créé via l'appel à connect() sur le composant wrappe (SimpleComponent)
  <Connector mapStateToProps={mapStateToProps}>
    <SimpleComponent
      dispatch={Provider.store.dispatch} // dispatch est récupéré depuis le Provider
      // simpleCounter est récupéré la fonction mapStateToProps du Connector
      // appelé sur le store récupéré depuis le Provider
      simpleCounter={Connector mapStateToProps(Provider.store.getState()).simpleCounter}
    </SimpleComponent>
  </Connector>
</Provider>
```

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)

Step 6 : redux, redux-thunk et les devtools

Pré-requis

Vous devez maîtriser les étapes 0, 1, 2, 3, 4 et 5 du workshop afin de pouvoir réaliser l'étape 6.

Le code disponible dans cette étape correspond au résultat attendu des étapes 0, 1, 2, 3, 4, et 5.

Pour lancer l'application de l'étape 5, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur.

Dans cette étape, vous allez avoir besoin de l'API. Pour l'exécuter, lancez la commande `npm start` dans le dossier `api`. La documentation de l'API est disponible à l'adresse <http://localhost:3000>.

Vous avez également la possibilité de lancer les tests de cette étape (que nous avons rédigé pour vous) en utilisant la commande `npm test` afin de voir quelles parties de l'étape fonctionnent et quelles parties ne fonctionnent pas du tout. N'hésitez pas à lire le code des tests afin d'avoir quelques indications en plus sur la façon d'écrire votre application.

Objectif

Maintenant que vous maîtrisez `redux`, vous pouvez vous lancer dans un gros refactoring pour que tout l'état applicatif soit contenu dans le store `redux`.

L'état global de l'application

durant cette étape votre état global va énormément évoluer et devrait finalement ressembler à ceci

```
{  
  comments: {  
    count: 42          // the global count for comments  
  },  
  likes: {  
    count: 42         // the global count for likes  
  },  
  regions: {  
    data: [...]        // the list of regions  
  },  
  wines: {           // a big object with list of wines by region  
    bordeaux: {  
      data: [...]      // the list of wines for bordeaux  
    },  
    ...               // here goes the other regions  
  },  
  currentWine: {  
    wine: {...},       // the actual current wine object  
    liked: false,       // do you like the current wine  
  }  
}
```

```

    comments: [...], // comments for the current wine
  },
  title: 'Bordeaux', // The current title of the app
  http: {
    state: 'LOADING', // values can be : LOADING, LOADED, ERROR
    error: '...' // the last HTTP error
  }
}

```

afin d'arriver à un model tel que celui-ci vous devrez créer un certain nombre de reducers dans le dossier `src/reducers` et avoir un fichier `src/reducers/index.js` comme celui-ci

```

import { combineReducers } from 'redux';
import { comments } from './comments';
import { likes } from './likes';
import { regions } from './regions';
import { wines, currentWine } from './wines';
import { title } from './title';
import { http } from './http';

export const app = combineReducers({
  comments,
  likes,
  regions,
  wines,
  currentWine,
  title,
  http
})

```

pour information voici une liste des actions nécessaires pour faire fonctionner l'application

function addLike()	// +1 sur le compteur de likes
function removeLike()	// -1 sur le compteur de likes
function setLikes(likes)	// définit la valeur du compteur de likes
function addComment()	// +1 sur le compteur de comments
function setComments(comments)	// définit la valeur du compteur de comments
function setCurrentWine(wine)	// définit la valeur du vin courant
function setCurrentComments(comments)	// définit la valeur des commentaires
function setCurrentLiked(liked)	// définit la valeur du like du vin courant
function setRegions(regions)	// définit la valeur des régions
function setTitle(title)	// définit la valeur du titre de la page
function setWines(region, wines)	// définit la valeur des vins pour une région
function loading()	// définit le fait que la page est en train de charger
function loaded()	// définit le fait que la page est chargée
function errorLoading(error)	// définit le fait que le chargement a échoué
function fetchLikesCount()	// lance le chargement du compteur de likes
function fetchCommentsCount()	// lance le chargement du compteur de comments
function fetchRegions()	// lance le chargement des régions
function fetchWinesForRegion(region)	// lance le chargement des vins pour une région
function fetchWine(wine)	// lance le chargement d'un vin
function toggleWineLiked()	// lance le changement de la valeur du like du vin
function fetchWineLiked()	// lance le chargement du like du vin
function fetchComments(wine)	// lance le chargement des commentaires
function postComment(wine, comment)	// poste un nouveau commentaire pour le vin

appels asynchrones et redux-thunk

Il est évident que certaines actions vont avoir besoin de déclencher des appels HTTP et

de dispatcher un résultat en conséquence. Il va donc falloir être capable de dispatcher plusieurs messages dans une même action afin que la logique métier soit localisée dans les actions. Ce genre de fonctionnement n'est pas supporté par défaut dans `redux`. Il va falloir ajouter un [middleware redux](#) afin de gérer ce cas.

Nous allons utiliser [redux-thunk](#) qui permet de faire de l'inversion de contrôle au niveau des actions `redux` et d'avoir connaissance sur state courant et du dispatcher dans l'action.

par exemple une action classique se défini de la façon suivante

```
export function setTitle(title) {
  return {
    type: 'SET_TITLE',
    title
  };
}
```

si on souhaite utiliser cette action de manière asynchrone, nous devons enchaîner les dispatchs de façon externe à l'action

```
this.props.dispatch(setTitle('Loading ...'));
fetch('/title.json')
  .then(r => r.json())
  .then(data => this.props.dispatch(setTitle(data.title)));
```

Le problème ici est qu'il peut être compliqué de mettre en commun les actions asynchrones. Avec `redux-thunk`, l'action s'écrira de la façon suivante

```
export function setTitle(title) {
  return {
    type: 'SET_TITLE',
    title
  };
}

export function fetchTitle() {
  return (dispatch, state) => {
    // ici on peut accéder au state global avec la fonction state
    dispatch(setTitle(`Loading ${state().nextTarget} ...`));
    fetch('/title.json')
      .then(r => r.json())
      .then(data => dispatch(setTitle(data.title)));
  };
}

...

this.props.dispatch(fetchTitle());
```

Avec `redux-thunk` il est donc facile d'écrire des actions qui dispatcheront d'autre actions, et donc il est simple de mettre en commun toute la logique de chargement de données via HTTP dans les actions.

Il pourrait également être opportun d'afficher en haut de l'application l'activité réseau (`Loading ... , Erreur : ...`)

Pour installer les `redux-thunk`, ajoutez les lignes suivantes dans votre fichier `package.json` puis lancez `npm install`

```
"dependencies": {  
    ...  
    "redux-thunk": "2.0.1",  
    ...  
}
```

ou alors via la ligne de commande

```
npm install --save redux-thunk@2.0.1
```

il vous faudra ensuite modifier la création de votre `store` `redux` pour prendre en compte ce `middleware`

```
import { applyMiddleware, createStore } from 'redux';  
import thunk from 'redux-thunk';  
import app from './reducers';  
  
const store = createStore(app, applyMiddleware(thunk));  
  
store.dispatch(fetchLikesCount());  
store.dispatch(fetchCommentsCount());
```

redux-devtools

Une fois que vous aurez réussi à passer votre application en `redux` du sol au plafond, vous pourrez installer [redux-devtools](#) et jouer avec le `time travelling` rendu possible par `redux` et sa gestion d'état immuable.

Pour installer les `devtools`, ajoutez les lignes suivantes dans votre fichier `package.json` puis lancez `npm install`

```
"dependencies": {  
    ...  
    "redux-devtools": "3.1.1",  
    "redux-devtools-log-monitor": "1.0.5",  
    "redux-devtools-dock-monitor": "1.1.0",  
    ...  
}
```

ou alors via la ligne de commande

```
npm install --save redux-devtools@3.1.1 redux-devtools-log-monitor@1.0.5 redu
```

il vous faudra ensuite modifier la création de votre `store` `redux` et ajouter un composant `DevTools` pour prendre en compte le nouveau plugin.

Commencez par créer un nouveau composant dans `src/components/devtools.js` et ajoutez y le contenu suivant

```
import React from 'react';  
import { createDevTools } from 'redux-devtools';  
import LogMonitor from 'redux-devtools-log-monitor';  
import DockMonitor from 'redux-devtools-dock-monitor';
```

```

export const DevTools = createDevTools(
  <DockMonitor toggleVisibilityKey="ctrl-h" changePositionKey="ctrl-q">
    <LogMonitor theme="solarized" />
  </DockMonitor>
);

```

Ce composant va être responsable de l'affichage des devtools dans l'application (Attention ce genre d'outil ne doit être utilisé que dans un environnement de développement).

Une fois le composant fini, il va falloir l'inclure dans l'application (`wine-app.js`). Cependant, comme notre application peut-être lancée dans un environnement de test, nous aimerions que les Devtools ne soient pas utilisés lors des phases de test. En effet, leur utilisation dans l'environnement de test ralentit énormément ces derniers. Nous allons donc tester si nous sommes en environnement de test et si c'est le cas, rendre un composant `null`.

```

import React, { PropTypes } from 'react';
import { GlobalStats } from './stats';
import { connect } from 'react-redux';
import { DevTools } from './devtools';

const NoDevToolsCauseInTestEnv = React.createClass({
  render() {
    return null;
  }
});

const mapStateToProps = (state) => {
  ...
};

export const WineApp = connect(mapStateToProps)(React.createClass({
  propTypes: {
    ...
  },
  contextTypes: {
    ...
  },
  render () {
    const Tools = window.TEST ? NoDevToolsCauseInTestEnv : DevTools;
    return (
      <div>
        <div className="grid">
          ...
        </div>
        <Tools />
      </div>
    );
  }
}));

```

Il ne reste plus qu'à customiser le store pour l'instrumenter avec les devtools

dans votre fichier `src/app.js` changez la déclaration du store pour la suivante

```

import { createStore, applyMiddleware, compose } from 'redux';
import thunk from 'redux-thunk';

```

```

import { app } from './reducers';
import { DevTools } from './components/devtools';

// ici on compose différents middleware que l'on application a la fonction ci
const store = compose applyMiddleware(thunk), DevTools.instrument()(createSt

```

Et vous pouvez maintenant jouer avec les devtools et jouer avec le temps dans votre application. Il y a cependant un problème. Même si l'état de votre application change, ce dernier n'est pas synchronisé avec le routeur et l'expérience des `devtools` n'est pas optimale. Pour éviter cela nous allons utiliser `react-router-redux`.

Commençons par installer la dépendance. Ajoutez les lignes suivantes dans votre fichier `package.json` puis lancez `npm install`

```

"dependencies": {
  ...
  "react-router-redux": "4.0.0",
  ...
}

```

ou alors via la ligne de commande

```
npm install --save react-router-redux@4.0.0
```

Ajoutez ensuite le `reducer` dédié de `react-router-redux` dans votre `reducer global` (`src/reducers/index.js`)

```

import { combineReducers } from 'redux';
import { comments } from './comments';
import { likes } from './likes';
import { regions } from './regions';
import { wines, currentWine } from './wines';
import { title } from './title';
import { http } from './http';
import { routerReducer } from 'react-router-redux';

export const app = combineReducers({
  comments,
  likes,
  regions,
  wines,
  currentWine,
  title,
  http,
  // ici on rajoute un reducer capable de mémoriser la route courante
  routing: routerReducer
})

```

il ne reste plus qu'à synchroniser les changements d'état du routeur avec le store, ce qui revient à dispatcher des actions pour chaque changement de route.

Editez votre fichier `src/app.js` comme suivant

```

import React from 'react';
import ReactDOM from 'react-dom';

import { Router, Route, browserHistory, IndexRoute } from 'react-router';

```

```

import { Provider } from 'react-redux';
import { createStore, applyMiddleware, compose } from 'redux';
import { syncHistoryWithStore } from 'react-router-redux';
import thunk from 'redux-thunk';
import { app } from './reducers';
import { DevTools } from './components/devtools';

const store = compose(applyMiddleware(thunk), DevTools.instrument())(createSt
...

export const App = React.createClass({
  propTypes: {
    history: PropTypes.object
  },
  componentDidMount() {
    store.dispatch(fetchLikesCount());
    store.dispatch(fetchCommentsCount());
  },
  render() {
    // ici on synchronise le routeur avec le store
    const history = syncHistoryWithStore(this.props.history || browserHistory
    return (
      <Provider store={store}>
        <Router history={history}> // et on utilise notre historique synchronis
          <Route path="/" component={WineApp}>
            <IndexRoute component={RegionsPage} />
            <Route path="regions/:regionId" component={WineListPage} />
            <Route path="regions/:regionId/wines/:wineId" component={WinePage}>
              <Route path="*" component={NotFound} />
            </Route>
          </Router>
        </Provider>
    );
  }
});

```

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Bonus

Vous pouvez profiter du passage au mode full redux pour faire de la mise en cache d'appels de services.

Vous pouvez facilement arriver à ce résultat en gérant le cache au niveau des actions permettant de récupérer les régions ainsi que les vins d'une région.

Voici à quoi pourrait ressembler votre état global

```
{
  comments: {
    count: 42
  },
  likes: {
    count: 42
  },
  regions: {
    lastUpdated: 0, // timestamp du dernier fetch réel pour les régions
    data: [ ... ]
}
```

```
},
wines: {
  bordeaux: {
    lastUpdated: 0, // timestamp du dernier fetch réel pour les vins d'une région
    data: [...]
  },
  ...
},
currentWine: {
  wine: {...}
},
liked: true,
comments: [...],
},
title: 'Bordeaux',
http: {
  state: 'LOADING', // values are : LOADING, LOADED, ERROR
  error: '...'
}
}
```

pour les régions par exemple, il est possible d'avoir un timestamp pour situer le dernier `fetch` dans le temps. Si ce dernier n'est pas assez récent, il est possible de retourner la valeur déjà présente dans l'état au lieu d'aller la chercher sur le serveur.

n'oubliez pas de mettre à jour le timestamp lorsque la valeur en cache n'est plus valide ;-)

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)

Step 7 : aller plus loin ...

Pré-requis

Le code disponible dans cette étape correspond au résultat attendu des étapes 0, 1, 2, 3, 4, 5 et 6.

Pour lancer l'application de l'étape 6, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL <http://localhost:8080> dans votre navigateur. Vous avez également besoin de l'API (lancez la commande `npm start` dans le dossier `api`).

Why react is awesome

<http://jlongster.com/Removing-User-Interface-Complexity,-or-Why-React-is-Awesome>

Presentational and Container components

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.ys0wrphn6

react components as Classes

<https://facebook.github.io/react/docs/reusable-components.html#es6-classes>

Higher Order components

<https://medium.com/@franleplant/react-higher-order-components-in-depth-cf9032ee6c3e>

Stateless components

<https://facebook.github.io/react/docs/reusable-components.html#stateless-functions>

react context

<https://facebook.github.io/react/docs/context.html>

Prochaine étape

Une fois cette étape terminée, vous pouvez aller jusqu'à [l'étape suivante](#) et vous frotter à `react-native`

Step 8 : react-native

Pré-requis

Le code disponible dans cette étape à une application [react-native](#) avec un minimum de code pour commencer dans de bonnes conditions.

Pour lancer l'application, exécutez la commande `npm start` (après avoir fait un `npm install`), n'oubliez pas de lancer l'API (lancez la commande `npm start` dans le dossier `api`).

Objectif

Ecrire l'application de gestion de vin pour mobile en [react-native](#)

Mise en place de l'environnement

Pour cette partie, vous aurez besoin d'installer quelques petites choses en plus par rapport à la version web.

Commencez par installer le client `react-native` de manière globale (`-g`) sur votre système.

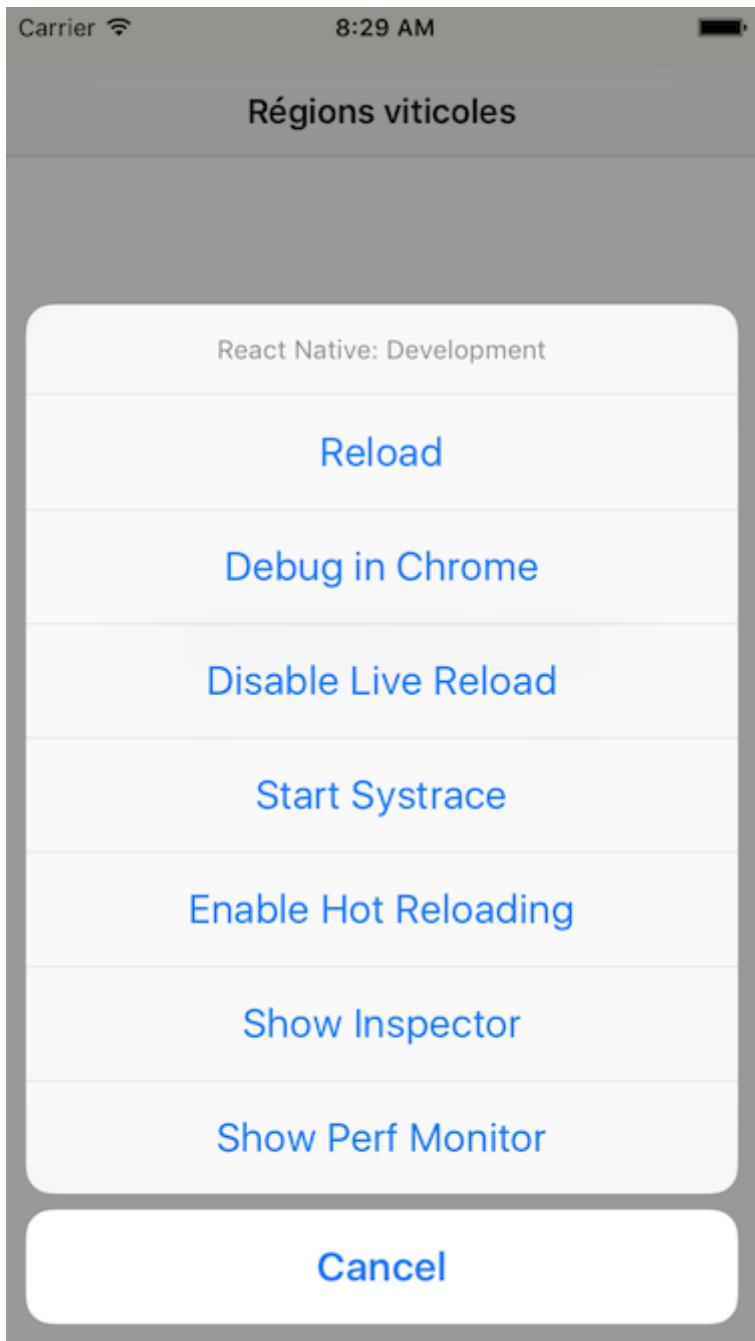
```
npm install -g react-native-cli
```

Sous Linux et Mac OS, vous aurez sûrement besoin d'installer `watchman`, suivez les [instructions d'installation](#).

iOS

Commencez par vous rendre [sur le Mac App Store](#) et installez Xcode.

Une fois Xcode installé, ouvrez le projet `step-8/ios/wines.xcodeproj` dans `xcode` et lancez le projet dans un émulateur (iPhone 6). Faites `Cmd+D` et activez le `Hot reload` pour recharger l'application au fur et à mesure des changements dans le code.

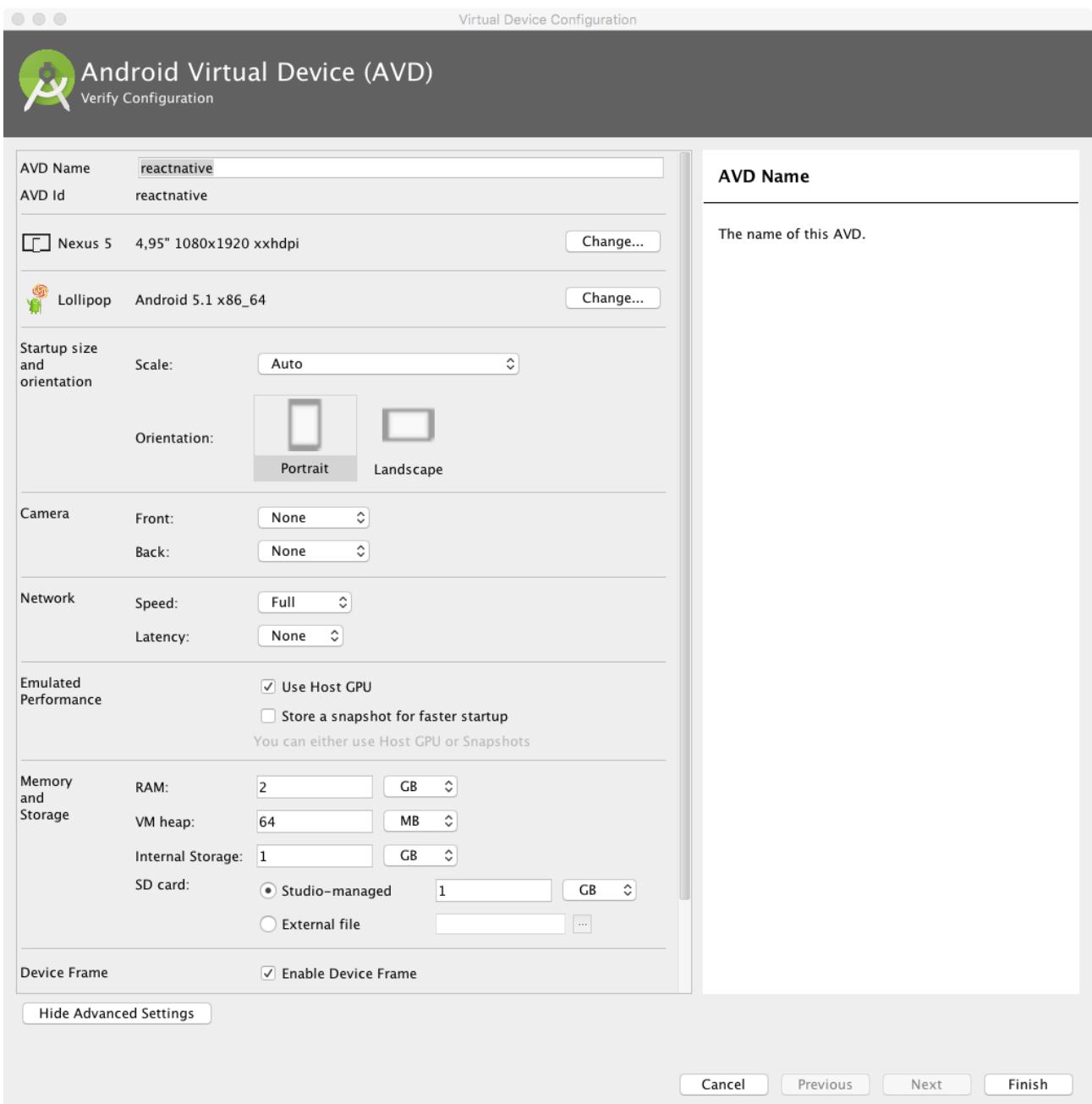


Android

Commencez par installer un [JDK8](#), [Gradle](#), et [Android Studio](#)

Vous pouvez ensuite suivre [cette documentation](#) sur le site de `react-native`.

En ce qui concerne l'émulateur, si vous n'avez rien d'existant pour le moment (émulateur, Genymotion), vous pouvez suivre [cette partie de la documentation](#) en utilisant le nom d'émulateur `reactnative` et en utilisant 1Go de mémoire interne et 1Go de mémoire externe et 2Go de RAM (Android Lollipop, API 22, 1Go de stockage interne, 1Go de stockage SD, 2Go de RAM, Nexus 5, nommé `reactnative`)



Ouvrez le projet `step-8/android` dans `Android Studio`, lancez l'émulateur (manuellement ou avec la commande `npm start run-adv`), déployez l'application (avec la commande `npm start install-android`) et lancez la depuis le menu application du device Android. Faites `Ctrl+F2` et activez le `Hot reload` pour recharger l'application au fur et à mesure des changements dans le code.

Passage de l'application en `react-native`

Un des avantages de `react-native` est de pouvoir faire de la réutilisation de code, nous allons donc pouvoir récupérer une bonne partie du code de notre application.

Voici ce que devrait donner l'application une fois finie

iOS

Régions viticoles

Bordeaux

Bourgogne

Champagne

Languedoc

Loire

La page de sélection de la région

[!\[\]\(54e2ac1b5c94f4b72d3cac879dcd2ffb_img.jpg\) Régions viticoles](#) Vins de Bordeaux

- Château Chevrol Bel Air
- Clarendelle "Inspiré par Haut Brion"
- Cheval Noir
- Les Hauts de Tour Prignac
- Château Bel-Air
- Château Lestage
- Château Tour des Termes
- L'Oratoire de Chasse Spleen
- Château Laniote
- Château Patache d'Aux
- Château Chantalouette
- Château Latour Camblanes
- Chateau Cormeil Figeac

~~Leillane Cuvée de l'Abbaye Brut~~

La page de sélection du vin

[Vins de Bordeaux Château Chevrol Bel Air](#)

Château Chevrol Bel Air



Type

Rouge

Région

Bordeaux

Appellation

Lalande-de-Pomerol

Cépages

Cabernet Sauvignon, Merlot,
Cabernet Franc

Like

Comments

Un bon bordeaux ! (le 01/04/2016 à 17:55:33)

J'ai bu le millésime 2009, parfait après une heure en carafe !

Titre du commentaire

Commentaire

Commenter

La page de description du vin avec un commentaire

[Vins de Bordeaux Château Chevrol Bel Air](#)

Château Chevrol Bel Air



Type

Rouge

Région

Bordeaux

Appellation

Lalande-de-Pomerol

Cépages

Cabernet Sauvignon, Merlot,
Cabernet Franc

Unlike

Comments

Un bon bordeaux ! (le 01/04/2016 à 17:55:33)

J'ai bu le millésime 2009, parfait après une heure en carafe !

Titre du commentaire

Commentaire

Commenter

*La page de description du vin avec un favori***Android**

6:10

Régions viticole

Bordeaux

Bourgogne

Champagne

Languedoc

Loire

La page de sélection de la région

6:10

Vins de Bordeaux

Château Chevrol Bel Air

Clarendelle "Inspiré par Haut Brion"

Cheval Noir

Les Hauts de Tour Prignac

Château Bel-Air

Château Lestage

Château Tour des Termes

L'Oratoire de Chasse Spleen

Château Laniote

Château Patache d'Aux

Château Chantalequette



La page de sélection du vin

Château Chevrol Bel Air



Type

Rouge

Région

Bordeaux

Appellation

Lalande-de-Pomerol

Cépages

Cabernet Sauvignon, Merlot,
Cabernet Franc

Like

Comments

Un bon bordeaux ! (le 01/04/2016 à 17:55:33)

J'ai bu le millésime 2009, parfait après une heure en carafe !

Titre du commentaire

Commentaire



La page de description du vin avec un commentaire

Château Chevrol Bel Air



Type

Rouge

Région

Bordeaux

Appellation

Lalande-de-Pomerol

Cépages

Cabernet Sauvignon, Merlot,
Cabernet Franc

Unlike

Comments

Un bon bordeaux ! (le 01/04/2016 à 17:55:33)

J'ai bu le millésime 2009, parfait après une heure en carafe !

Titre du commentaire

Commentaire



Mise en place de l'environnement

Cette partie décrit comment créer le projet `react-native` à titre informatif car ce dernier a déjà été créé dans l'étape 8 avec quelques additions pour pouvoir se lancer plus rapidement. Pour pouvez cependant dérouler le scénario dans un dossier à part pour maîtriser le processus.

Commencez par installer `watchman` et `react-native-cli`

puis créez votre projet à l'endroit désiré

```
react-native init wines
```

maintenant il ne reste plus qu'à lancer l'application dans un émulateur.

iOS

Ouvrez le projet `wines/ios/wines.xcodeproj` dans `xcode` et lancez le projet dans un émulateur (iPhone 6). Faites `Cmd+D` et activez le `Hot reload` pour recharger l'application au fur et à mesure des changements dans le code. Essayez de modifier le contenu de `wines/index.ios.js` pour voir vos modifications s'appliquer directement.

Android

Ouvrez le projet `wines/android` dans `Android Studio`, lancez l'émulateur (manuellement ou avec la commande `npm start run-adv`), déployez l'application (avec la commande `npm start install-android`) et lancez la depuis le menu application du device Android. Faites `Ctrl+F2` et activez le `Hot reload` pour recharger l'application au fur et à mesure des changements dans le code.

Let's code !!!

L'application mobile proposée a pour point d'entrée `index.ios.js` ou `index.android.js`, suivant la plateforme d'exécution. Ce fichier est un simple lanceur vers votre véritable application se trouvant dans `src/app.js`.

Dans ce fichier vous allez pouvoir instancier votre composant WineApp tout en lui fournissant un store `redux` (via le composant `<Provider />`). On repart sur les même base que l'application Web, la partie Redux reste sensiblement la même et est donc déjà fournie dans le dossier de l'étape.

```
import React from 'react-native';

import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import { app } from './reducers';

import { WineApp } from './components/wine-app';

const store = createStore(app, applyMiddleware(thunk));

export const App = React.createClass({
  render() {
```

```

        return (
          <Provider store={store}>
            <WineApp />
          </Provider>
        );
      }
    )];

```

N'oubliez pas pour le reste de l'application, le fichier `src/components/style.js` contient tous les styles dont vous pouvez avoir besoin pour votre application.

Attention : si vous utilisez développez une application Android, allez modifier le fichier `src/actions/index.js` et changez `export const apiHost = '127.0.0.1';` par votre adresse IP afin que l'émulateur Android puisse atteindre le serveur de l'API

Routage

Votre application mobile, à l'instar la version web, va avoir besoin de router son utilisateur à travers divers écrans, cependant ici, pas besoin de se soucier de l'URL, des boutons du navigateur, etc ...

Cependant, le routage entre écran sous iOS et Android est complètement différent, nous allons donc utiliser deux API différentes suivant l'OS désiré, c'est pour celà que nous allons avoir deux fichiers `src/components/wine-app.ios.js` et `src/components/wine-app.android.js`.

iOS

`react-native` fournit une API de navigation liée à iOS, qui utilise directement des APIs Objective-C. Le composant en question s'appelle `<NavigatorIOS />` et permet de spécifier un style, ainsi qu'une route de départ, votre écran d'accueil. Chaque composant rendu par `<NavigatorIOS />` se verra assigné une propriété `navigator` possédant une méthode `push` pour naviguer vers l'écran suivant.

Chaque `route` passée au `navigator` est en fait un object constitué d'un titre, d'un composant et éventuellement d'une liste de propriétés

```

this.props.push({
  title: 'Ma page 1',
  component: Page1,
  passProps: {
    itemsToDisplay: [ ... ]
  }
});

```

Par exemple dans notre cas, linstanciation de `<NavigatorIOS />` s'écrit de la façon suivante :

```

import React, { NavigatorIOS } from 'react-native';
import { Regions } from './regions';
import { styles } from './style';

export const WineApp = React.createClass({
  render() {
    return (
      <NavigatorIOS
        style={styles.navigatorios}
        initialRoute={{
          title: 'Régions viticoles',

```

```

        component: Regions
    } } />
);
}
});

```

Android

Le cas d'Android est un peu plus compliqué, puisqu'il n'y a pas d'API dédiée à la navigation. Nous allons donc nous appuyer sur une API purement Javascript `<Navigator />` fournie par `react-native`. De plus, pour ne pas avoir trop de code spécifique à écrire, nous allons faire en sorte que l'API de navigation fournie pour chaque `page` soit la même que pour iOS, à savoir une propriété `navigator` possédant une méthode `push` pour naviguer vers l'écran suivant. Chaque `route` passée au `navigator` est en fait un object constitué d'un titre, d'un composant et éventuellement d'une liste de propriétés

```

this.props.push({
  title: 'Ma page 1',
  component: Page1,
  passProps: {
    itemsToDisplay: [ ... ]
  }
});

```

Nous utiliserons donc l'API `<Navigator />` de la façon suivante :

```

import React, { Navigator, BackAndroid, Text, View } from 'react-native';
import { Regions } from './regions';
import { styles } from './style';

export const WineApp = React.createClass({
  componentWillMount() {
    // ici on décable le bouton back android
    BackAndroid.removeEventListener('hardwareBackPress', this.handleBackButton);
  },

  handleBackButton() {
    // si on ne se trouve pas sur l'écran d'accueil
    if (!this.isOnMainScreen) {
      // on demande au navigator de faire un retour arrière
      this.navigator.pop();
      return true;
    }
    return false;
  },

  renderScene(route, nav) {
    if (!this.navigator) {
      this.navigator = nav;
      // ici on cable le bouton back android pour faire un retour arrière
      BackAndroid.addEventListener('hardwareBackPress', this.handleBackButton);
    }
    // on récupère le composant sur la route
    const Component = route.component;
    // on récupère les propriétés sur la route
    const props = route.passProps || {};
    // on récupère le titre de la route
    const title = route.title;
    // on regarde si on est sur l'écran d'accueil
    if (Component === Regions) {
      this.isOnMainScreen = true;
    } else {

```

```

        this.isOnMainScreen = false;
    }
    // on retourne notre vue qui est composée d'un titre et du composant avec
    // le title et l'API de navigation
    return (
      <View>
        <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
          <Text style={{ flex: 1, textAlign: 'center', fontSize: 20, font
        </View>
        <Component {...props}>
          navigator={nav}
          title={title} />
      </View>
    );
  },

  // configuration de la scène d'un point de vue animation
  // ici chaque nouvel écran arrivera par le bas
  configureScene(route) {
    if (route.sceneConfig) {
      return route.sceneConfig;
    }
    return Navigator.SceneConfigs.FloatFromBottom;
  },

  // ici instantiation du navigateur avec son style, sa configuration de scèn
  // sa route initiale
  render() {
    return (
      <Navigator
        style={styles.navigatorandroid}
        initialRoute={{ title: 'Régions viticoles', component: Regions }}
        renderScene={this.renderScene}
        configureScene={this.configureScene} />
    );
  }
);

```

Listes

Maintenant, vous allez avoir à créer deux listes permettant d'afficher les régions et les vins pour chaque région. Pour cela nous allons utiliser l'API `<ListView />` de `react-native`.

Cette API fournit un modèle de programmation pour afficher des listes cliquables. Pour implémenter une liste d'items, nous aurons besoin de deux composants, la liste et les cellules à afficher par la liste. Une cellule est un composant très simple comme par exemple :

```

import React, { PropTypes, Text, TouchableHighlight, View } from 'react-native';
import { styles } from './style';

export const ItemCell = React.createClass({
  propTypes: {
    onSelect: PropTypes.func,
    item: PropTypes.object,
  },
  render() {
    return (
      <TouchableHighlight onPress={this.props.onSelect}>
        <View style={styles.container}>
          <Text style={styles.cellTitle}>
            {this.props.item.name}
          </Text>
        </View>
    );
  }
});

```

```

        </TouchableHighlight>
    );
}
});

```

La liste quant à elle doit utiliser notre `store redux` et peut se coder de la façon suivante. Vous noterez l'utilisation de `componentDidUpdate` pour détecter les changements de valeur du store et mettre à jour la liste.

```

import React, { PropTypes, ListView } from 'react-native';

import { connect } from 'react-redux';
import { fetchItems } from '../actions';
import { ItemCell } from './region-cell';
import { MyComponent } from './wine-list';
import { styles } from './style';

const mapStateToProps = (state) => {
  return {
    items: state.items
  };
}

export const ItemList = connect(mapStateToProps)(React.createClass({
  propTypes: {
    dispatch: PropTypes.func.isRequired,
    navigator: PropTypes.object,
    items: PropTypes.arrayOf(PropTypes.object)
  },
  getInitialState() {
    return {
      // ici on utilise une datasource spécifique pour la ListView.
      dataSource: new ListView.DataSource({ rowHasChanged: (row1, row2) => rc
    };
  },
  // ici on lance l'appel HTTP pour récupérer les données à afficher
  componentDidMount() {
    this.props.dispatch(fetchItems());
  },
  // logique permettant de détecter les changements de valeurs de items au ni
  // du store redux et de mettre à jour la datasource le cas échéant.
  componentDidUpdate(prevProps) {
    if (prevProps.items !== this.props.items) {
      this.setState({
        dataSource: this.state.dataSource.cloneWithRows(this.props.items)
      });
    }
  },
  // cette fonction gère les évènements de type `selection` sur une cellule de
  // la liste et déclenche la navigation vers la prochaine vue.
  selectItem(item) {
    this.props.navigator.push({
      title: `Showing ${item.name}`,
      component: MyComponent,
      passProps: {
        item
      }
    });
  },
  // cette fonction retourne une vue pour chaque cellule de la list

```

```

renderItem(item) {
  return(
    <ItemCell onSelect={() => this.selectItem(item)} item={item} />
  );
},
// ici on rend une ListView avec une source de données et une fonction pour
// rendre chaque cellule de la liste
render() {
  return (
    <ListView style={styles.listView}
      dataSource={this.state.dataSource}
      renderRow={this.renderItem} />
  );
}
);
}
));

```

Dans notre application nous avons donc besoin de deux listes. Il serait bien que chaque liste affiche une page de chargement durant l'appel HTTP (voir le composant `<Loading />`).

La liste de sélection des vins pourrait également afficher une miniature de la photo de la bouteille de vin en plus du nom du vin.

Fiche de détail d'un vin

La fiche de détail d'un vin est laissée à votre imagination, n'hésitez pas à consulter [la documentation react-native](#) pour voir les possibilités offertes par le framework.

La structure technique du composant `src/components/wine.js` est cependant la suivante :

```

import React, { PropTypes, Image, View, Text, ScrollView, TouchableWithoutFee
import { connect } from 'react-redux';
import { fetchWine, fetchWineLiked, setTitle, toggleWineLiked } from '../acti
import { styles } from './style';
import { apiHost } from '../actions';

const mapStateToProps = (state) => {
  return {
    currentWine: state.currentWine.wine,
    liked: state.currentWine.liked
  };
}

export const Wine = connect(mapStateToProps)(React.createClass({
  componentDidMount() {
    this.props.dispatch(fetchWine(this.props.wine.id)).then(() => {
      this.props.dispatch(fetchWineLiked(this.props.wine.id));
    });
  },
  handleToggleLike() {
    this.props.dispatch(toggleWineLiked(this.props.wine.id));
  },
  render() {
    const { wine, liked } = this.props;
    return (
      ...
    );
  }
}));

```

Commentaires

Le composant de commentaire est extrêmement similaire à la version web, vous avez à votre disposition le composant `<TextInput />` offert par `react-native` et un composant `<Button />` dans le dossier `src/components`

```
import React, { PropTypes, View, Text, TextInput } from 'react-native';
import moment from 'moment';
import { connect } from 'react-redux';
import { addComment, fetchComments, postComment } from '../actions';
import { styles } from './style';
import { Button } from './button';

const mapStateToProps = (state) => {
  return {
    comments: state.currentWine.comments
  };
}

export const Comments = connect(mapStateToProps)(React.createClass({
  getInitialState() {
    return {
      commentTitle: '',
      commentBody: ''
    };
  },
  handlePostComment() {
    const payload = { title: this.state.commentTitle, content: this.state.com
    this.props.dispatch(postComment(this.props.wineId, payload)).then(() => {
      ...
      this.setState({ commentTitle: '', commentBody: '' });
    });
  },
  render() {
    return (
      ...
    );
  }
}));
```

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Bonus

Vous pouvez rajouter le compteur de stats globales présent dans la version web (qui a volontairement été oublié) à la version mobile. A vous de fouiller [la documentation react-native](#) afin de trouver une belle façon d'intégrer cette nouvelle fonctionnalité.

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#)

```
npm install --save-dev babel-register@6.7.2 jsdom@8.1.0 mocha@2.4.5 chai@3.4.
```

Ajoutez la configuration nécessaire à `babel-register` dans le fichier `.babelrc` :

```
{  
  "presets": ["es2015", "react"]  
}
```

Cette configuration est globale pour babel et sera utilisée si aucune autre configuration n'est passée aux outils babel. Ainsi vous pouvez enlever la partie configuration du `babel-loader` dans `webpack.config.js` afin que tous vos outils utilisent la même configuration babel.

Créez un dossier `tests` dédiés aux tests unitaires de vos composants. Ecrivez ensuite le test du composant `Wine` dans un fichier `tests/components/wine.spec.js`, en utilisant :

- la syntaxe Mocha pour décrire le test,
- ReactTestUtils pour effectuer le rendu et parcourir l'arbre DOM du composant `Wine`,
- Chai pour vérifier le texte affiché.

```
import React from 'react';  
import { expect } from 'chai';  
import ReactTestUtils from 'react-addons-test-utils';  
  
import Wine from '../../src/components/wine';  
  
describe('Wine', () => {  
  it('affiche le nom du vin', () => {  
    const wine = ReactTestUtils.renderIntoDocument(<Wine name="Un bon Bourgogne" />);  
    const div = ReactTestUtils.findRenderedDOMComponentWithTag(wine, 'div');  
    expect(div.textContent).to.be.equal('Un bon Bourgogne');  
  });  
});
```

Pour s'exécuter correctement, le test précédent nécessite de disposer globalement des objets `window` et `window.document`, ainsi que de la fonction `window.documentElement.createElement`. Pour cela, créez un fichier `bootstrap.js` qui se base sur la librairie `jsdom` pour créer l'environnement DOM nécessaire au bon fonctionnement de `ReactTestUtils`.

```
import jsdom from 'jsdom';  
  
export function bootstrapEnv(body = '') {  
  const doc = jsdom.jsdom(`<!doctype html><html><body>${body}</body></html>`)  
  const win = doc.defaultView;  
  function propagateToGlobal(window) {  
    for (const key in window) {  
      if (!window.hasOwnProperty(key)) continue;  
      if (key in global) continue;  
      global[key] = window[key];  
    }  
  }  
  global.document = doc;  
  global.window = win;  
  propagateToGlobal(win);  
  console.log('\nENV setup is done !!!');
```

```
}
```

Créez enfin un fichier `index.js`, point d'entrée permettant d'exécuter l'ensemble des tests unitaires :

```
import { bootstrapEnv } from './bootstrap';
bootstrapEnv();
const tests = [
  require('./components/wine.spec.js')
];
```

Exécution des tests via NPM

Ajoutez un nouveau script dans le fichier `package.json` permettant de lancer les tests à l'aide de la commande `npm test` :

```
"scripts": {
  "test": "mocha --compilers js:babel-register tests/index.js"
}
```

Vous pouvez également préciser à ESLint qu'il doit désormais également traiter le dossier `tests` :

```
"scripts": {
  "lint": "eslint src tests"
}
```

Pour plus tard ...

`ReactTestUtils` permet également de simuler des clics sur des éléments du DOM :

```
ReactTestUtils.Simulate.click(button);
```

Si vous voulez aller plus loin dans les tests, il pourrait être intéressant de regarder du côté [d'enzyme](#), un utilitaire de test pour `react` créé par Airbnb et qui permet de manipuler plus facilement votre arbre de composant pour le tester. En effet `enzyme` s'inspire de l'API de `jquery` pour pouvoir [requêter l'arbre de composants](#).

```
import React from 'react';
import { expect } from 'chai';
import { shallow } from 'enzyme';

import Wine from '../../../../../src/components/wine';

describe('Wine', () => {
  it('affiche le nom du vin', () => {
    const wine = shallow(<Wine name="Un bon Bourgogne" />);
    expect(wine.contains(<div>Un bon Bourgogne</div>)).to.be.true;
    expect(wine.find('div.wine')).to.have.length(1);
  });
});
```

});

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la [version corrigée](#) ;-)

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la [version corrigée](#) puis aller jusqu'à [l'étape suivante](#)