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 a 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 facon 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 précédant 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 parametres par defaut
 * 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'usage 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 a 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 prevenant 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` gèrera 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 différent 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'a 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 lors qu'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 meme 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 a jour son état a 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 notificat
    // 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 n
    // 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 co
    this.setState({ counter });
  },
  render() {
    return (
      <div>
        <span>counter : {this.state.counter}</span>
      </div>
    );
  }
});
```

Cependant tout ce boilerplate peut se réveler fastidieux et rébarbatif à écrire à la longue. Pour éviter tout ce code inutile, la librairie `react-redux` permet de fournir un store a un arbre de composants `react` et de connecter un composant `react` a 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 a 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 celà, `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 componsant 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 a 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 composant 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)(SimpleCompo
```

Il ne nous reste donc plus qu'a 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 (Simp
  <Connector mapStateToProps={mapStateToProps}>
    <SimpleComponent
      dispatch={Provider.store.dispatch} // dispatch est récupéré depuis le P
      // simpleCounter est récupéré la fonction mapStateToProps du Connector
      // appelé sur le store récupéré depuis le Provider
      simpleCounter={Connector.mapStateToProps(Provider.store.getState()).sim
  </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 jetter 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