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 maitrisez `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)              // défini la valeur du compteur de lik
function addComment()                 // +1 sur le compteur de comments
function setComments(comments)        // défini la valeur du compteur de com
function setCurrentWine(wine)         // défini la valeur du vin courant
function setCurrentComments(comments) // défini la valeur des commentaires c
function setCurrentLiked(liked)       // défini la valeur du like du vin cou
function setRegions(regions)          // défini la valeur du des régions
function setTitle(title)              // défini la valeur du titre de la pag
function setWines(region, wines)      // défini la valeur des vins pour une
function loading()                    // défini le fait que la page est en t
function loaded()                     // défini le fait que la page est char
function errorLoading(error)          // défini le fait que le chargement co
function fetchLikesCount()            // lance le chargement du compteur de
function fetchCommentsCount()         // lance le chargement du compteur de
function fetchRegions()               // lance le chargement des régions
function fetchWinesForRegion(region)  // lance le chargement des vins pour u
function fetchWine(wine)              // lance le chargement d'un vin
function toggleWineLiked()            // lance le changement de la valeur du
function fetchWineLiked()             // lance le chargement du like du vin
function fetchComments(wine)          // lance le chargement des commentaire
function postComment(wine, comment)   // poste un nouveau commentaire pour l
```

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 enchainer 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'écrirai 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 immutable.

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 effets, leur utilisation dans l'environnement de test ralentit énormément ces dernier. 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) => {
  return {
    ...
  };
}

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'a 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 cr
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`.

Commencons 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'a synchroniser les changements d'état du routeur avec le store, ce qui revient a 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 synchron
          <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 jetter 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 région
   data: [...]
```

```
  },
  wines: {
    bordeaux: {
      lastUpdated: 0, // timestamp du dernier fetch réel pour les vins d'une r
      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 est 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 a 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