Un tutoriel d'introduction à React + Redux

17 octobre 2017

Dans ce tutoriel, nous allons créer une liste de contacts en utilisant React et Redux. Nous aurons une liste de contacts et une fois que nous aurons cliqué sur l'un d'eux, les détails de ce contact seront affichés.

Cela nous servira de prétexte pour comprendre les composants clés et fondamentaux d'une application *React + Redux.* Cet exemple est basé sur l'exemple de livre de Stephen Grider, un cours de Modern React with Redux, sur Udemy . J'encourage quiconque à l'acheter.

Le code commence également à partir de son projet React Simple Starter

Remarque : aimez-vous la coloration syntaxique? Essayez l' essentiel ici ou vérifiez le dépôt ici

Commençons:

Les **composants clés** d'une application React + Redux sont les suivants:

Réducteurs

Les conteneurs

Actions (et ActionCreators)

1. Réducteurs

Un réducteur est une fonction qui retourne une partie de l'état de l'application

Disons que l'état de l'application est comme suit

```
state = {
  contacts: [{
    "name": "Miguel",
    "phone": "123456789",
  },{
    "name": "Peter",
    "phone": "883292300348",
  },{
    "name": "Jessica",
    "phone": "8743847638473",
  },{
```

```
      "name": "Michael",
      "phone": "0988765553",
    }],
    activeContact: {
      "name": "Miguel",
      "phone": "123456789",
    }
}
```

Nous aurions besoin de **deux réducteurs** , on pour les *contacts* et un autre pour *activeContact*

Par exemple, notre liste de contacts peut être un élément d'état dans notre application et un réducteur totalement valide (mais statique) pour contenir cette liste serait:

```
export default function () {
  return [{
    "name": "Miguel",
    "phone": "123456789",
  },{
    "name": "Peter",
    "phone": "883292300348",
  },{
    "name": "Jessica",
    "phone": "8743847638473",
```

```
 },{
   "name": "Michael",
   "phone": "0988765553",
 }];
}
```

Nous devons avoir un *"RootReducer"* qui est la combinaison de tous les réducteurs de notre application (notre état d'application / *Redux Store*). Pour cela, *Redux* propose la fonction *combineReducers* :

```
// reducers/index.js
import { combineReducers } from 'redux';
import ContactsReducer from './reducer_contacts'
import ActiveContactReducer from './reducer_active_contact'
const rootReducer = combineReducers({
 contacts: ContactsReducer,
 activeContact: ActiveContactReducer
});
export default rootReducer;
```

Comment pouvons-nous câbler cet état dans un composant *React* ? Nous avons besoin du paquet ReactRedux, qui nous aide à câbler React et Redux. Comment? **Les conteneurs**

2. Conteneurs

Un conteneur enveloppe un composant *React* et lui fournit l'état qu'il doit lui transmettre en tant *qu'accessoires* à l'intérieur du composant.

Quels composants sont supposés être des conteneurs / composants intelligents et lesquels doivent être des composants muets? **Lorsqu'un composant a besoin de connaître un élément d'état particulier dans notre *magasin Redux* , il doit alors s'agir d'un conteneur.**

Pour un connecteur au Composant *Redux magasin* , nous importateur de la Devons de connexion à fonction partir de la bibliothèque react-Redux et lui à paramêtres Transmettre DÉCRIRE CERTAINS.

Supposons que nous avons une liste de contacts dans un composant appelé ContactList et que nous devons le connecter au *Redux Store* .

```
import React, {Component} from 'react'
import {connect} from 'react-redux'
import selectContact from '../actions/action_select_contact'
import {bindActionCreators} from 'redux'
class ContactList extends Component {
 renderList() {
  return this.props.contacts.map((contact) => {
   return (
    <li
     key={contact.phone}
     onClick={() => this.props.selectContact(contact)}
     className='list-group-item'>{contact.name}</li>
```

```
      );

    });

  }

  render() {

    return (

      <ul className = 'list-group col-sm-4'>

        {this.renderList()}

      </ul>

    );

  }

}

function mapStateToProps(state) {

  return {

    contacts: state.contacts

  };

}

export default connect(mapStateToProps[,...later...])(ContactList)
```

Remarquez comment la fonction mapStateToProps recevra l’ état **complet** du *Redux Store* et nous devons en sélectionner la pièce qui **nous intéresse** , en renvoyant un objet qui sera utilisé comme accessoire de ContactList. Si nous avons retourné l'objet suivant

```
function mapStateToProps(state) {

  return {
```

```
    amazingContacts: state.contacts

  }
}
```

Une fois connecté à ContactList, il serait disponible via this.props.amazingContacts

3. Actions

Si vous réfléchissez attentivement à ces composants, vous constaterez quelque chose de très important: **nous lisons simplement** , mais nous n'avons pas le moyen de modifier l'état de notre composant. Pour cela, nous avons besoin de **créateurs d'action** qui vont générer des actions. Ces actions seront ensuite transmises à tous les réducteurs combinés dans notre rootReducer et chacun d'eux retournera une nouvelle copie de l'état modifié (ou non) en fonction de l'action demandée. Notre réducteur ressemblerait maintenant à quelque chose comme ceci (remarquez le cas par défaut, car nous devons **toujours** retourner un état

```
// reducer_active_contact.js

export default function(state = null, action) {

 switch (action.type) {

  case 'CONTACT_SELECTED':

   return action.payload

 }

 // i dont care about the action because it is not inside my

 // list of actions I am interested int (case statements inside the switch)

 return state
```

```
}
```

Nous avons encore du travail à faire, nous devons créer les créateurs d'action, qui est un nom de *fantaisie* pour une fonction qui retourne un objet avec 2 éléments: *type* et *charge utile* . De ces 2 éléments, le seul obligatoire (y compris le type de nom) est le type.

```
// actions/action_select_contact.js

function selectContact(contact) {

  return {

    type: 'CONTACT_SELECTED',

    payload: contact

  }

}

export default selectContact;
```

Nous pouvons maintenant avoir une fonction qui crée des actions, mais ces actions doivent passer par *Redux* . Comment pouvons-nous faire en sorte que les actions se *déroulent via Redux Reducers* ? Via la fonction de **connexion** ! C'est le paramètre de fonction secret

mapDispatchToProps:

```
// ... some other imports...

import selectContact from '../actions/action_select_contact'

import {bindActionCreators} from 'redux'

// The ContactList component goes here

// mapStateToProps is here
```

```
function mapDispatchToProps(dispatch) {

  return bindActionCreators({selectContact: selectContact}, dispatch);

}

export default connect(mapStateToProps, mapDispatchToProps)(ContactList)
```

Cette fonction est celle qui garantit que chaque fois que l'action selectContact est déclenchée, tout ce qu'elle retourne passera par tous les réducteurs. Si nous avions plus d'actions, cela ferait la même chose pour chaque action. Notez que le nom peut différer de l'objet:

```
function mapDispatchToProps(dispatch) {

  return bindActionCreators({

    myAction1: action1,

    myAction2: action2

  }, dispatch);

}
```

Maintenant, nos accessoires à l'intérieur du composant ContactList contiendront également les actions que nous avons câblées (selectContact, myAction1, myAction2...)

Le résultat du composant ContactList est maintenant le suivant:

```
import React, {Component} from 'react'

import {connect} from 'react-redux'

import selectContact from '../actions/action_select_contact'

import {bindActionCreators} from 'redux'

class ContactList extends Component {
```

```
  renderList() {
    return this.props.contacts.map((contact) => {
      return (
        <li
          key={contact.phone}
          onClick={() => this.props.selectContact(contact)}
          className='list-group-item'>{contact.name}</li>
      );
    });
  }
  render() {
    return (
      <ul className = 'list-group col-sm-4'>
        {this.renderList()}
      </ul>
    );
  }
}
function mapStateToProps(state) {
  return {
    contacts: state.contacts
  };
}
```

```
function mapDispatchToProps(dispatch) {

  return bindActionCreators({
    selectContact: selectContact
  }, dispatch);

}

export default connect(mapStateToProps, mapDispatchToProps)(ContactList)
```

Les prochaines étapes seraient maintenant:

> Dans les réducteurs (notez le pluriel) agissez en fonction du type d'action reçu
>
> Etat de la mise à jour (cela déclenche un nouveau rendu des composants affectés)

Dans notre exemple, le seul réducteur qui se soucie du contact sélectionné est le ActiveContactReducer que nous n'avons pas encore créé mais que nous allons créer maintenant.

Les réducteurs reçoivent deux arguments: la **pièce d'État** qu'ils respectent (et **non** l'ensemble de l'État) et l' **action actuelle** qui traverse les réducteurs. N'oubliez pas qu'ils renvoient une nouvelle **copie** de cet état modifiée en conséquence de l'action reçue.

Dans ce cas, l'élément d'état activeContact sera mis à jour pour le contact contenu dans le fichier action.payload. Quelle que soit la fonction retournée, elle sera affectée telle quelle à l'état que manipule le réducteur. Un réducteur doit toujours retourner quelque chose:

```
//reducer_active_contact

export default function (state = null, action) {
```

```
 switch (action.type) {
  case 'CONTACT_SELECTED':
    return action.payload
 }
 return state;
}
```

Maintenant, pour afficher un contact ou un autre, nous avons besoin d'un conteneur <ContactDetail> (rappelez-vous qu'il s'agit d'un conteneur car il concerne un élément de l'état du *magasin Redux).*

Nous devons donc maintenant:

Créer un composant <ContactDetail>

Connecter le composant au *Redux Store*

Ajouter le réducteur au rootReducer

Le composant <ContactDetail> sera affiché le dernier, une fois terminé.

Pour connecter le composant au *Redux Store* , il est identique à celui d'avant: la fonction mapStateToProps est transmise en tant qu'argument à la fonction de connexion avec le composant et exporte le conteneur.

```
// inside containers/contact-detail.js
import { connect } from 'react-redux'
// ... ContactDetail component here ...
```

```
function mapStateToProps(state) {

  return {

    contact: state.activeContact
    //activeContact is defined in the rootReducer

  }

}

export default connect(.....)
```

Le réducteur est ajouté au rootReducer de la même manière que le ContactReducer

```
import { combineReducers } from 'redux';

import ContactsReducer from './reducer_contacts'

import ActiveContactReducer from './reducer_active_contact'

const rootReducer = combineReducers({

  contacts: ContactsReducer,

  activeContact: ActiveContactReducer

});

export default rootReducer;
```

Résumé du flux

Maintenant que tout est câblé, cliquer sur l'un des contacts de la liste devrait alors déclencher le créateur de l'action selectContact, qui sera transmis à tous les réducteurs du rootReducer. ActiveContactReducer réagira à cette action en modifiant l'état qu'il contrôle (*activeContact*) en le définissant sur selectedContact contenu dans action.payload.

Lorsque cela se produira, <ContactDetail> verra son état modifié et sera restitué, ayant une nouvelle valeur pour this.props.contact.

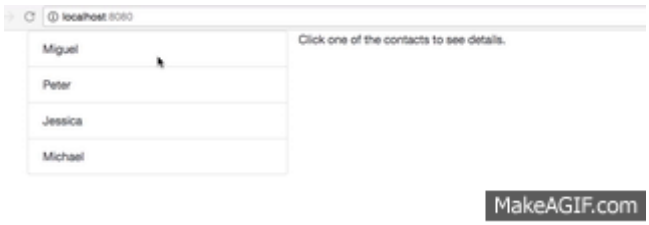
Le <ContactDetail> ressemblerait à ceci:

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
class ContactDetail extends Component {
 render() {
  if (!this.props.contact) {
   return (
    <div>Select a contact from the list to see its details</div>
   );
  }
  return (
   <div>
    <h3>Contact Details for: {this.props.contact.name}</h3>
    <div>Phone: {this.props.contact.phone}</div>
   </div>
  );
 }
}
function mapStateToProps(state) {
 return {
```

```
    contact: state.activeContact

  }

}

export default connect(mapStateToProps)(ContactDetail);
```

Résultat final

Et le résultat final n'est pas étonnant, mais ça marche, vous pouvez le voir ici dans cette image GIF:



L'application Contacts en action

Comment tout est câblé ensemble?

Ok, tout cela est comme par magie... comment mon application sait-elle initialement où et quels sont mes réducteurs? (rappelez-vous que l'état est défini en combinant tous les réducteurs avec combineReducers)

Cela se fait dans notre fichier principal, où nous définissons l'application, à l'aide du composant <Fournisseur>, qui reçoit le magasin, le rendant disponible pour le composant qu'il encapsule.

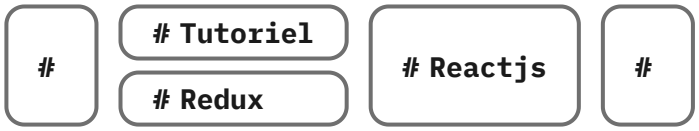
Pour passer le magasin (rappelez-vous que c'est le rootReducer), nous allons le combiner avec la fonction applyMiddleware, qui tiendra nos middlewares (quand nous en aurons besoin) et leur transmettra l' *état* avant d'atteindre l'application.

```
import React from 'react';
```

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

import { Provider } from 'react-redux';

import { createStore, applyMiddleware } from 'redux';

import App from './components/app';

import reducers from './reducers'; // this exports rootReducer!!!

const createStoreWithMiddleware = applyMiddleware()(createStore);

ReactDOM.render(

 <Provider store={createStoreWithMiddleware(reducers)}>

  <App />

 </Provider>

, document.querySelector('.container'));
```

Si vous trouvez cela utile, envisagez de m'acheter un café

https://www.buymeacoffee.com/mamoreno



Continuer la discussion

Plus d'histoires connexes

13 choses à savoir sur React

Krzysztof Jendrzyca

Javascript

5 Best React.js Admin Templates 2018

Aspirity
Sep 27

Reactjs Template

0–100 in Django: Starting an app the right way

Jeremy Spencer
Sep 04

Python