Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Oct 18, 2017 · 7 min read

A Basic React + Redux introductory tutorial

In this tutorial we are going to create a list of Contacts using React and Redux. We will have a list of contacts and once we click on one of them, the details of that contact will be displayed.

This will serve us as an excuse to understand the key and basic components of a *React+Redux* Application and the example is based on the books example from Modern React with Redux course by Stephen Grider on Udemy. I do encourage anyone to buy it.

The code starts also from his React Simple Starter project

Note: Do you like Syntax coloring? Try the gist here or check the repo here

. . .

Lets start:

The **key components** of a React+Redux App are the following:

1. Reducers
2. Containers
3. Actions (and ActionCreators)

1. Reducers

A reducer is a function that returns a piece of the application state

Let's say the application state is as follows

```
state = {
```

```
    contacts: [{
        "name": "Miguel",
        "phone": "123456789",
    },{
        "name": "Peter",
        "phone": "883292300348",
    },{
        "name": "Jessica",
        "phone": "8743847638473",
    },{
        "name": "Michael",
        "phone": "0988765553",
    }],
    activeContact: {
        "name": "Miguel",
        "phone": "123456789",
    }
}
```

We would need **two reducers**, on for *contacts* and another one for *activeContact*

For example, our list of contacts can be a piece of state in our application and a totally valid (but static) reducer to hold that list would be:

```
export default function () {
  return [{
    "name": "Miguel",
    "phone": "123456789",
  },{
    "name": "Peter",
    "phone": "883292300348",
  },{
    "name": "Jessica",
    "phone": "8743847638473",
  },{
    "name": "Michael",
    "phone": "0988765553",
  }];
}
```

We must have a *"RootReducer"* that is the combination of all the reducers of our application (our application state */ Redux Store*). For that, *Redux* offers the *combineReducers* function:

```
// 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;
```

How do we wire that state into a *React* component? We need ReactRedux package, which helps us to wire React and Redux. How? **Containers**

2. Containers

A container wraps a *React* component and makes available to him the piece of state we need to pass to it as *props* inside the component.

What components are supposed to be containers/smart components and which ones should be dumb components? **When a component needs to know about a particular piece of state from our *Redux Store*, then it needs to be a container.**

To connect a Component with the *Redux Store*, we need to import the connect function from react-redux library and pass it some parameters to be described.

Let's assume we have a list of contacts in a component called ContactList and we need to connect it to the *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)
```

Notice how mapStateToProps function will receive the **whole** state of the *Redux Store* and we must select from it the piece that is **interesting for us**, returning an object that will be the used as props of ContactList. If we returned the following object

```
function mapStateToProps(state) {

  return {

    amazingContacts: state.contacts

  }
}
```

Once connected to ContactList, it would be available via this.props.amazingContacts

3. Actions

If you think of this components carefully, you will realize of something very important: **We are simply reading**, but we do not have a way to modify the state from our component. For that, we need **Action creators** that will generate actions. Those actions will be then passed to the ALL of the reducers combined into our rootReducer and each one of them will return a new copy of the state modified (or not) according to the requested action. Our reducer would look now something like this (notice the default case, because we **always** must return a state

```
// 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

}
```

We still have work to do, we must create the action creators, which is a *fancy* name for a function that returns an object with 2 elements: *type* and *payload*. Of those 2 elements, the only mandatory one (including the name type) is the type.

```
// actions/action_select_contact.js

function selectContact(contact) {

  return {

    type: 'CONTACT_SELECTED',

    payload: contact

  }

}

export default selectContact;
```

Now we can have a function that creates actions, but those actions have to flow though *Redux*. How do we make the actions flow though *Redux Reducers* ? Via the **connect** function! That is the secret function parameter

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)
```

This function is the one that ensured that whenever the action selectContact is triggered, whatever it returns will flow though all the reducers. If we had more actions, it would do the same for each action. Notice the name can differ from the object:

```
function mapDispatchToProps(dispatch) {

  return bindActionCreators({

    myAction1: action1,

    myAction2: action2

  }, dispatch);

}
```

Now, our props inside the ContactList component will also contain the action/s we have wired (selectContact, myAction1, myAction2…)

The result of the ContactList component is now as follows:

```
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)
```

The next steps now would be:

1. Inside the reducers (note the plural) take action according to the type of action received
2. Update state (this triggers a re-render of the affected components)

In our example, the only reducer that cares about the selected contact is the ActiveContactReducer which we haven't created yet but we are going to create now.

Reducers receive two arguments: the **piece of state** they care (**not** the whole state) and the **current action** that is flowing through the reducers. Remember, they return a new **copy** of that piece of state that is modified accordingly to the action received.

In this case, the activeContact piece of state will be updated to the contact contained in the action.payload. Whatever the function returns, will be assigned as-is to the piece of state that reducer is handling. A Reducer must always return something:

```
//reducer_active_contact

export default function (state = null, action) {

  switch (action.type) {

    case 'CONTACT_SELECTED':

      return action.payload

  }

  return state;

 }
```

Now, to display one contact or another, we need a <ContactDetail> container (remember, it is a container because it cares about a piece of state of the *Redux Store)*.

So, we must now:

1. Create a <ContactDetail> component
2. Connect the component to the *Redux Store*
3. Add the reducer to the rootReducer

The <ContactDetail> component will be shown the last, once completed.

To connect the component to the *Redux Store*, it is the same as before: mapStateToProps function passed as argument to the connect function together with the component and export the container.

```
// 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(.....)
```

The reducer is added to the rootReducer the same way as the ContactReducer

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

import ContactsReducer from './reducer_contacts'

import ActiveContactReducer from './reducer_active_contact'

const rootReducer = combineReducers({

  contacts: ContactsReducer,

  activeConctact: ActiveContactReducer

});

export default rootReducer;
```

Summary of the flow

Now it is everything wired, so clicking in one of the contacts of the list should trigger selectContact action creator, that will be passed to every reducer in the rootReducer. The ActiveContactReducer will react to that action by altering the piece of state it controls (*activeContact*) setting it to the selectedContact that comes inside of action.payload

and when that happens, <ContactDetail> will see its state modified and will be re-rendered, having a new value for this.props.contact .

The <ContactDetail> would look as follows:

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

import { connect } from 'react-redux'

class ConctactDetail 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.pages}</div>

      </div>

    );

  }

}

function mapStateToProps(state) {

  return {
```

```
    contact: state.activeContact

  }

}

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

Final Result

And the final result is not so amazing, but it works, you can see it here in this GIF image:

The Contacts Application in action

How is everything wired together?

Ok, all this is like magic… how does my application initially know where and which my reducers are? (remember the state gets defined by combining all the reducers with combineReducers)

That is done in our main file, where we are defining the application, using the <Provider> component, that receives the store, making it available to the component it wraps.

To pass the store (remember, that is the rootReducer) we are going to combine it with the applyMiddleware function, that will hold our middlewares (when we need them) and pass the *state* through them before reaching the 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'));
```