Getting Started With React Redux: An Intro

At Codementor, we've used React + Redux to build several projects. There are projects made entirely with Redux, and there are projects where we've used Redux along with an existing Fluxxor project.

Redux seems to be pretty popular right now, so I shall write this tutorial before I forget anything.

This is the first part of the React & Redux series:

- Getting Started with Redux: this article will introduce readers to Redux based on what I know, and also mention what's so great about its design
- Sever-side Rendering a Redux App: this tutorial goes through how to use Redux and react-router for server rendering.
- Unit Testing a Redux App: this article talks about the problems I've ran into when testing Redux code and how I solved them, and also goes over how to make your unit tests co-exist with webpack loaders.

The goal of this article is not to teach you how to build a Redux app (ノ °□°）ノ Wait~ don't close teh tab~

By the end of reading this article, you should have a better idea of how to structure your apps and how

to solve problems after reading the Redux documentation.

What is Redux?

Redux is a framework that controls states in a JavaScript app. According to the official site:
Redux is a predictable state container for JavaScript apps.

There are many states in an app that will change depending on time, user behavior, or a plethora of different reasons. **Thus, if we consider an app as the process to change its own state, the view a user sees is how the states are presented.**

For example, in a TODO list app, when we create a new todo item, we actually changed the state of an app from one without that TODO item to one with the TODO item. In addition, because the app's state has changed, and the view is how a state is presented, we will see the new TODO item in our view.

If an app's state unfortunately does not behave as expected, then it's time to debug the app. E.g. If we add a new TODO item but our state appears to have added two items instead of just one, we will have to spend time on figuring out which state change went wrong. Sounds pretty simple on paper, right? If you think this process will be easy, you're too naïve…. Real-world apps are often much more complex, and there are a lot of
factors that will make this debugging process a nightmare, such as having a bad coding habit, flaws in the framework design, not investing time in writing unit tests, etc.

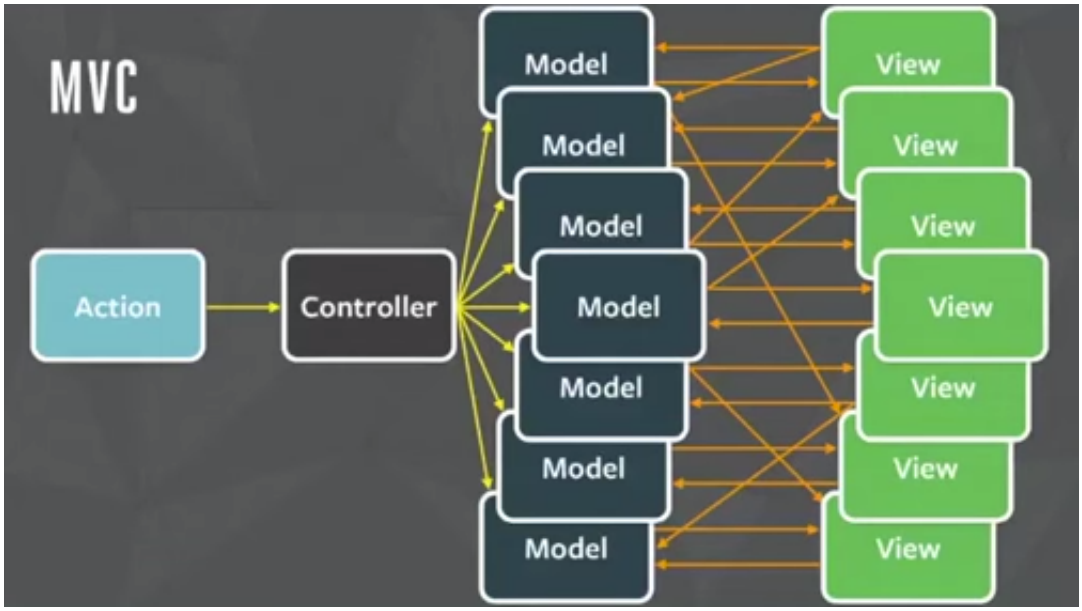
Redux's framework is designed to ease the debugging process. To be more specific, Redux is a framework that extends the ideas of Flux and simplifies redundant things. As it puts it on its official site:
Redux evolves the ideas of Flux, but avoids its complexity by taking cues from Elm.

What was wrong with the original MVC frameworks?

Originally, there were many MVC frameworks out there of different sizes and shapes, but generally can be either categorized as MVCs or an extension of MVC.

Flux/Redux attempts to resolve the problem where bugs are difficult to trace when a model has been continuously updated as an app is being used.

Just as illustrated in the following chart:



As seen in the image, any given view could affect any model, and vice versa. **The problem with this is when there are many views and models and one of the model's state changed into something we didn't expect, we are unable to efficiently trace which view or which model caused the problem**, since there are way too many possibilities.

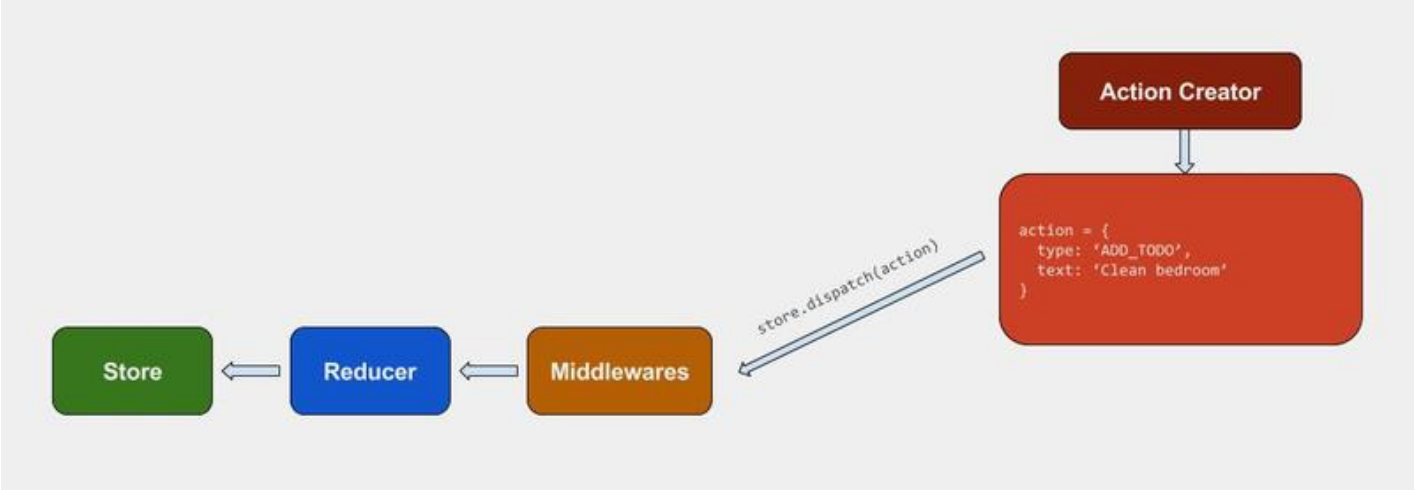
Flux/Redux makes debugging apps easier by removing the model (aka store) setter, and then letting the store update its own state via an `action` (which is the "unidirection data flow" mentioned on React's official site.) In addition, the dispatch of an `action` cannot dispatch another `action`. This way, a store will be actively updated as opposed to passively updated, and whenever there is a bug, we can refer to the problematic store to see what events have happened before. This makes bug hunting easier, since we've essentially zeroed down the possibilities that caused a bug.

Want to speed up your learning process? Learn React with a Live Expert

How does Redux Work?

Redux can be broken down into the following:
– `store`: manages the states. Mainly there is a `dispatch` method to dispatch an `action`. **In a Redux app, you can obtain its states via `store.getState()`**
– `action`: a simple, plain JavaScript object. **An action can also be considered as a command to change a state.**
– `reducer`: decides how to change a state after receiving an `action`, and thus can be considered the entrance of a state change. A `reducer` is comprised of functions, and it changes states by taking an `action` as an argument, in which it then `return`s a new state.
– `middleware`: the middleman between a `store.dispatch()` and a `reducer`. Its purpose is to intercept an `action` that has been `dispatch`ed, and modify or even cancel the `action` before it reaches the

`reducer`.



As shown above, if we added a new TODO item in a Redux app, we will first create an `action` with a type `ADD_TODO`, and then dispatch the action through `store.dispatch()`.

```
// actionCreator

export function addTodo(text) {
  return { type: types.ADD_TODO, text };
}

store.dispatch(addTodo({ text: 'Clean bedroom' }));
```

Afterwards, this `action` will enter the `middleware`, and finally enter the `reducer`. Inside the `reducer`, the state will change accordingly to the action's type.

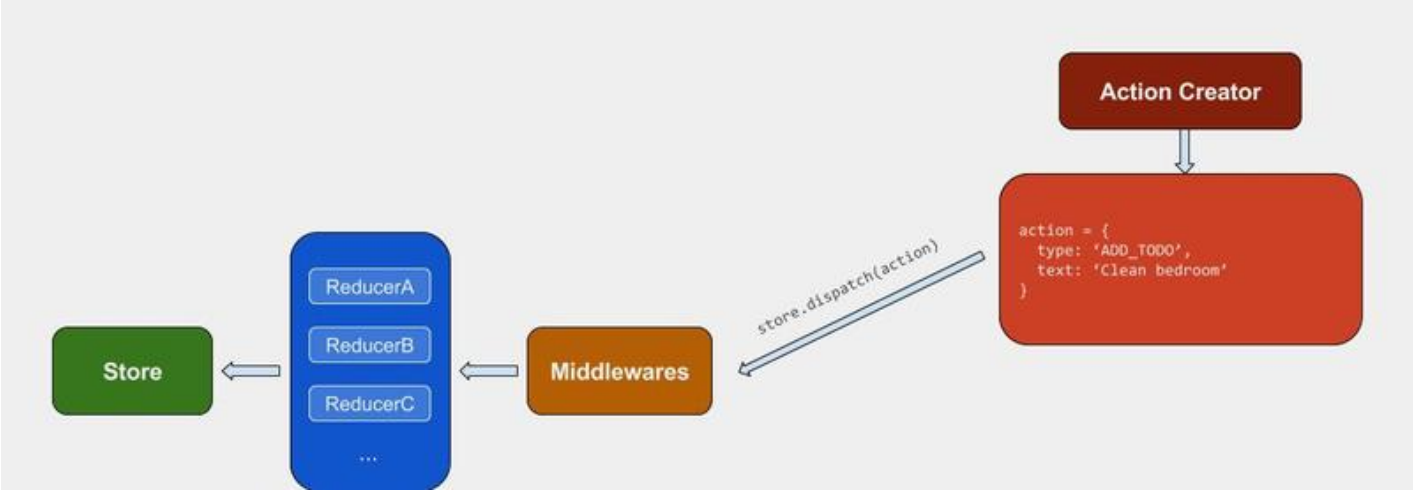
```
// reducer

function todos(state, action) {
  switch(action.type) {
    case 'ADD_TODO':
      // handle action and return new state here

  }
}
```

And so, we've completed the most basic behavior of updating a state.

In a more complex app, we will need to split up a store's state into different pieces like we'd do when

namespacing. You can do this in Redux by creating different reducers to manage different areas of a state, and then merge them together through `combineReducers`.

The process should look like this:



Async & Middleware

We've already introduced state updating, but a front-end app is never that simple~

Here we'll go over how to handle asynchronous behavior in a Redux app. In this section, we'll examine the process of sending an HTTP request to understand how async works in Redux.

The Sample Situation

Let's say we have an app that a list called `questions`. At some point (e.g. a user clicks a button), we will need to send a request to our server to obtain the data in `questions`. While sending this request, we need to reflect the `sending` state in our store, and if the request has been made successfully, we will need to put the data of `questions` in our store. If the HTTP request failed, we will need to reflect the failure info in our store.

The Straightforward Solution

One naive way to approach this situation is to `dispatch` different events at different times. In the code below, I used superagent to make my requests.

```
import request from 'superagent';

const SENDING_QUESTIONS = 'SENDING_QUESTIONS';
const LOAD_QUESTIONS_SUCCESS = 'LOAD_QUESTIONS_SUCCESS';
const LOAD_QUESTIONS_FAILED = 'LOAD_QUESTIONS_FAILED';
```

```
store.dispatch({ type: SENDING_QUESTIONS });
request.get('/questions')
  .end((err, res)=> {
    if (err) {
      store.dispatch({
        type: LOAD_QUESTIONS_FAILED,
        error: err
      });
    } else {
      store.dispatch({
        type: LOAD_QUESTIONS_SUCCESS,
        questions: res.response
      });
    }
  });
```

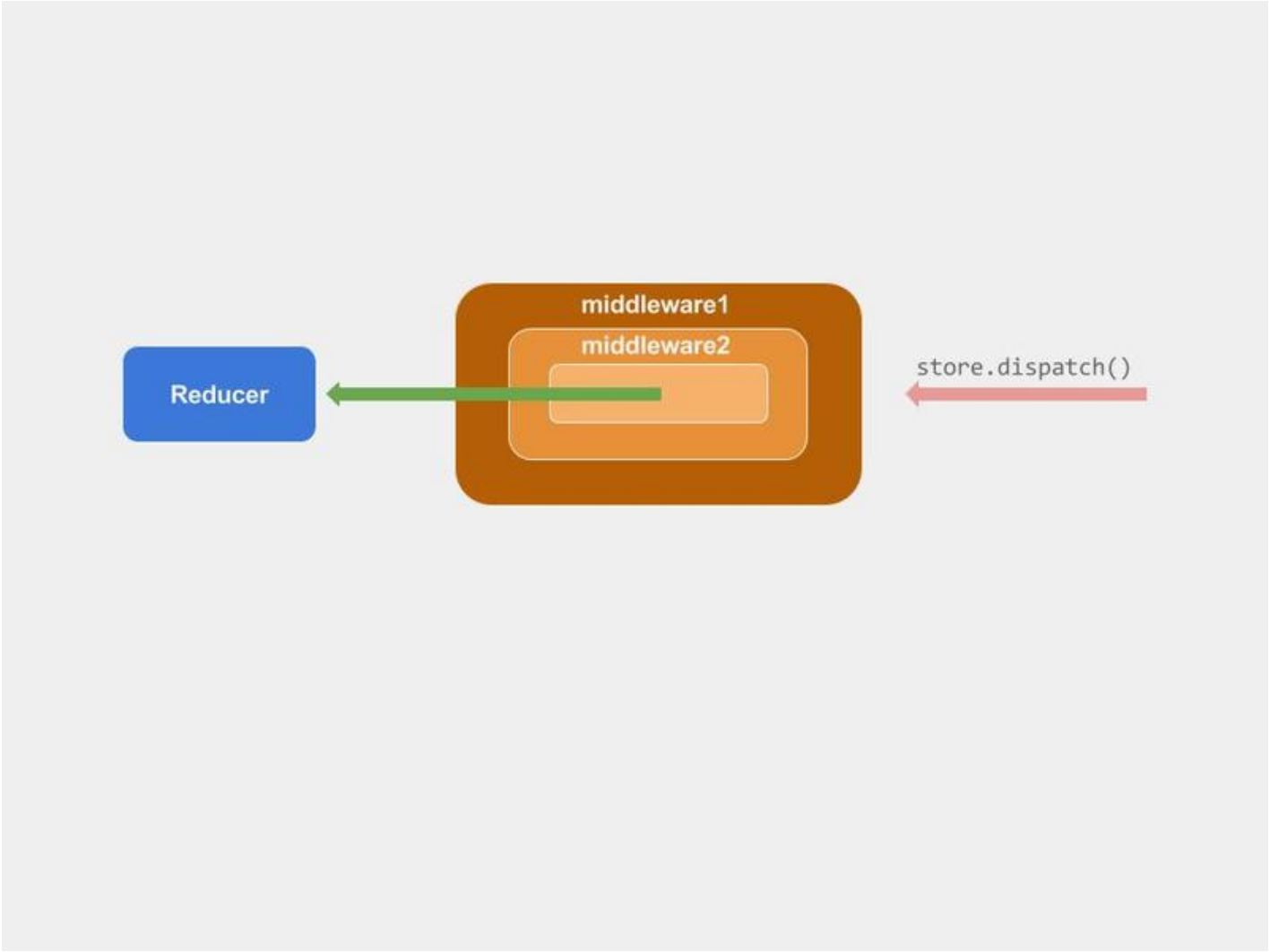
This way, we can achieve async behavior in a Redux app. However, this approach is not suitable if we have to send many HTTP requests, since we need to add async behavior to every request and this makes it difficult to maintain our code. Furthermore, this approach is also not easy to test, especially since asynchronous code is more difficult to understand and test in general. Finally, if we use this approach in a React app, we will be forced to code this logic into a React Component.

The most ideal way to solve this problem is to extract and gather all similar asynchronous behavior into the same place, and that's where `middleware` comes in!

What is a Middleware?

As stated before, in Redux a middleware is like the negotiator between `store.dispatch` and `reducer`. To be more specific, the `store.dispatch()` we call is actually comprised of layers of middleware, and the reducer is in the innermost layer.

We can visualize this via the following image:



Through a middleware, we can extract the above-mentioned asynchronous API request and place them in the same middleware. In other words, when sending an API request, we will need to `dispatch` an `action` of a specific type, and then let our middleware intercept such types of `action`s to send the respective API request.

Here's what our middleware can look like:

```
// middlewares/api.js

import superAgent from 'superagent';

export const CALL_API = Symbol('CALL_API');

export default store => next => action => {
  if ( ! action[CALL_API] ) {
    return next(action);
  }
  let request = action[CALL_API];
```

```
  let { method, path, query, failureType, successType, sendingType } = request;
  let { dispatch } = store;

  dispatch({ type: sendingType });
  superAgent[method](path)
    .query(query)
    .end((err, res)=> {
      if (err) {
        dispatch({
          type: failureType,
          response: err
        });
      } else {
        dispatch({
          type: successType,
          response: res.body
        });
      }
    });
};
```

In the code above, the middleware will intercept any `action` with the `CALL_API` key, and then send an API request based on the `method`, `path`, `query`, `successType`, etc stored inside the `action`.

This way, when we really need to send a request, we won't have to deal with asynchronous logic and we'll only need to `dispatch` an `action` that has `CALL_API`.

```
import { CALL_API} from 'middlewares/api';

const SENDING_QUESTIONS = 'SENDING_QUESTIONS';
const LOAD_QUESTIONS_SUCCESS = 'LOAD_QUESTIONS_SUCCESS';
const LOAD_QUESTIONS_FAILED = 'LOAD_QUESTIONS_FAILED';

store.dispatch({
  [CALL_API]: {
    method: 'get',
    path: '/questions',
    sendingType: SENDING_QUESTIONS,
    successType: LOAD_QUESTION_SUCCESS,
    failureType: LOAD_QUESTION_FAILED
  }
```

```
});
```

By grouping asynchronous behavior in a middleware, we eliminate the need to write duplicate logic in our code. We also make testing simpler, as the fact that "dispatching an `action` with a `CALL_API` is now synchronous behavior and we end up avoiding the need to test async code.

Redux's middleware design is one of the main reasons I find the framework elegant. Through a middleware, we are able to extract redundant logic like the API request mentioned above, and we can also log all sorts of behavior. I think the concept behind this design is close to the Chain of Responsibility Pattern. Other similar concepts have appeared in Express.js middleware and Rack middleware.

Hold on, What about React?

After reading an essay explaining what Redux does, you might realize that nothing has been said about React yet.

Yup, React and Redux don't have a direct relationship – Redux controls an app's state changes, while React renders the view of states.

React's strength is its virtual DOM to efficiently re-render a view whenever a state has been changed to a new state (which is packaged in `props`). In other words, an app developer will not have to manage which part of an altered state has been changed and just let React render it.

So how do you use React & Redux together? **We do this by finding the top-level React components, and inside these components we will set Redux's state as the component's state. When these states changes, we will use our handy `setState` component to trigger a re-render.** This way, React and Redux states will be bound together.

Once we have these "top-level" components, we can break them down into even smaller components by passing Redux states as React `props` to create the sub-components. However, **these sub-components are not directly related to Redux, because their behavior is completely determined by `props`.** Whether or not the `prop` came from Redux does not matter to these sub-components.

Actually, the "top-level components" I've mentioned are officially called **Smart Components** in Redux, and the "sub-components" are called **Dumb Components**. The `connect` function in React-Redux hooks together the Redux states and Smart Components.

What Does a React + Redux App Look Like?

The example below is from the official Redux site's TodoMVC example, and the entry point of `index.js` looks like this:

```
import 'babel-core/polyfill';
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import App from './containers/App';
import configureStore from './store/configureStore';
import 'todomvc-app-css/index.css';

const store = configureStore();

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

In the code above, we can see the app component `App` is wrapped inside the `Provider` component, and this `Provider` component uses a `prop` to point us to the app's state: `store`.

Here's what our `App`'s component `containers/App.js` should look like:

```
import React, { Component, PropTypes } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import Header from '../components/Header';
import MainSection from '../components/MainSection';
import * as TodoActions from '../actions/todos';

class App extends Component {
  render() {
    const { todos, actions } = this.props;
    return (
      <div>
        <Header addTodo={actions.addTodo} />
        <MainSection todos={todos} actions={actions} />
      </div>
    );
  }
}
```

```
App.propTypes = {
  todos: PropTypes.array.isRequired,
  actions: PropTypes.object.isRequired
};

function mapStateToProps(state) {
  return {
    todos: state.todos
  };
}

function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(TodoActions, dispatch)
  };
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

Here we'll see that `App` is actually just a simple React component and seems to have little to do with Redux. However, in the end the **.js** file does not `export` the `App` component directly, but exports the returned result of `connect`. Remember, the `connect` function will integrate Redux into our `App` component.

At the `connect` stage, we'll pass `mapStateToProps` and `mapDispatchToProps` into `connect`.

First of all, `mapStateToProps` filters the results of Redux states (store.getState()) into something that a Smart component needs, and then passes those into a component as a `prop`. Based on the sample above, `mapStateToProps` should return:

```
{
  todos: state.todos
};
```

Therefore, we should use `this.props.todos` to get `store,getState().todos` from an `App` component that has been `connect`ed.

Secondly, `mapDispatchToProps` takes a set of action creators (which simply return `action` objects) and

make them “dispatchable”. It then passes the object to components as a `prop`.

In the sample above, `TodoActions.addTodo` is an object that simply returns an `action`:

```
// actions/todos.js
export function addTodo(text) {
  return { type: types.ADD_TODO, text };
}
```

However, through a `connect` function, our `App` component will obtain a `this.props.action.addTodo`, and this function `dispatch`es the original action creator that was returned by an `action` object, i.e.:

```
dispatch(ActionCreators.addTodo(...))
```

How does `connect` get the `store`?
After examining the `containers/App.js` above, you’d probably realize that `store` does not appear anywhere in the code!

You may wonder how `connect` obtains `store` when it connects the React component to a Redux state.

The answer: `connect` gets the `store` through the `Provider` you saw in `index.js`.

In `index.js`, we passed the `store` into our `Provider` as a `prop`, and `Provider` will set the `store` as its own React `context`*. This way, all the child components of the `Provider` can obtain the `store` through `context`. Thus, an `App` that has been `connect`ed can also obtain the `store` through `context`.

Conclusion

Personally I think Redux has many elegantly designed features, and compared to Fluxxor you can write a lot less repetitive code and logic.

Out of all features, I like Redux’s `middleware` the most. After reading the source code for middleware, I realized there are a lot of functional programming concepts in the code. I am not familiar with functional programming so this opened my eyes to a new perspective and allowed me to understand JavaScript from a fresh point of view.

In addition, I think it was a great decision to extract the portion that connects React and Redux into a library, as this way Redux won’t be directly coupled with React.

In terms of testing, any test that involves views are usually rather complex, but Redux makes it easier by making the view components “dumb”. “Dumb” components only depend on `props` and thus can be

tested easily.

***Note:**

Before React 0.13, `context` was owner-based. After React 0.14, they were all changed to parent-based. This is also why in React 0.13 and earlier versions, we usually had to wrap a function around the `App` in our `Provider`, since doing so will prevent the error where parent-based and owner-based contexts differed.

```
<Provider store={store}>
    { ()=> {<App />} }
  </Provider>
```

This article was originally published in Chinese here by Yang-Hsing Lin, and was translated by Yi-Jirr Chen. Feel free to leave a comment below if you have any feedback or questions!