**Redux Tutorial #1 - ReactJS Tutorial - How Redux Works:**

(Redux is an open-source JavaScript library for managing application state. It is most commonly used with libraries such as React or Angular for building user interfaces. Similar to (and inspired by) Facebook's Flux architecture, it was created by Dan Abramov and Andrew Clark.)

Redux changes the Flux framework and provides some ways that help one to avoid a lot of complicated bugs

If you are going to build a small component, just use React and keep your data in a state on your layout level. If you are going to build more of an application, use Flux. When you start getting a complicated data scenario, a complicated chain of event, you may need to go to Redux.

((Smart) Components >> (Pass Data as Props) >> (Dumb) Components) >> Actions >> Reducers >> One Store >> Provider Component >> (Re-render when Store Changes) >> (Back to the First Item)

* In Redux, you do not have multiple stores, but you will have one big store
* Instead of multiple stores, you have multiple properties on that one big JavaScript object (store)
* In the to-do list application, we have a to-do list store, and we would have a setting store and a favourites stores
* We cannot change that JavaScript object. (JavaScript does not do immutability natively.) We only create a new version of the object. In this way, we always have a full history way back to the time our app booted up in the browser.
* We are going to wrap the entire application in one provider component. This is the only big change made to the components. When the store changes, the whole application may re-render.
  + If your base component is a layout, then your base component will be a provider component in Redux
* The only change to the component part is that you now get smart components and dumb components. Smart components are the top-level/page-level/container-level components. They are aware of your framework, they know how to pull a property out of your store.
  + E.g. the smart components of the to-do list application will take the to-do list array and they will inject that array into all the child properties. The dumb component only knows to receive a list of to-dos, either from Redux, Flux or Relay.
* The components can then trigger actions, just like Flux. Those actions can dispatch other actions. There may be asynchronous actions like action to fetch data and receive data. If there is an error, there may be a FETCH\_DATA\_ERROR action and the payload (裝載貨物) will be the error message.
* Reducers part is another change. Instead of having multiple stores that all manage their own data, you have multiple reducers that all modify the piece of data of the store. Reducer modifies the store in an immutable way. They always create a new chunk of data that go back to the store.
  + Go back to the to-do list example. One reducer will probably work on the to-do list portion of the store. One will work on the setting portion of the store, etc.
  + The reducer can react to the same action. When you click on the button which hides “completed” and “not completed” symbols. You trigger the action to hide the symbols. The to-do list reducer and setting reducer will care about the action. Later the page may update accordingly.
  + The reducers do not depend on one another

Redux really simplifies data. There is only one store, and it keeps the history. If you want to rollback the application while you are debugging, you can simply just require the older versions of the store.

**Immutable JS - Redux Tutorial #2 - ReactJS Tutorial:**

Immutability is something that JavaScript developers talk about a lot recently. Even though JavaScript does not handle immutability very well by default, we can write immutable data and it helps our programs a lot.

A piece of data being immutable means that we do not want to change it. We simply create a new piece of data based on its previous value if we want to update it.

* Enter “var a = {name: "Will"}” and then “a.name = "Fred"” in the console of Chrome’s DevTools. This change the variable a.
* An immutable change means that a.name still is “Will”, but there will be a new variable storing the new value of a.

There are two types in JavaScript: the reference type and the primitive type. We would like to implement immutability to those types.

* Enter “var a = 1;”, “var b = a;”, “b = 3” in the console. The value of a will still be 1.
* Object or array always create a reference
* Enter “var a = {name: "Will"};”, “var b = a;”, “b.name = "Fred";” in the console. The value of a will change as well.

We want to create a new object based on the existing one. (One can use commands like “Object.assign”, “$.extend”, “\_.extend” or “\_.assign”.)

* Enter “var a = {name: "Will", age: 35};” and “var b = {name: "David", age: 20};” in the console
* The line “var c = Object.assign({}, a, b);” will create a new, empty object as c and assign a, then b to c
* The value of c will be {name: "David", age: 20}
* The value of c after the line “var c = Object.assign({}, a, {name: "Fred"});” will be {name: "Fred", age: 35}. The value of a does not change.

When it comes to arrays…

* Enter “var a = [0, 1, 2];”, “var b = a.concat(3);”
* The value of a does not change. The value of b will be “[0, 1, 2, 3]”.
* Enter “var b = a.filter((val) => val !== 2);”. The value of b will be “[0, 1]”. The value of a does not change.
* One can also use map and reduce function to implement immutability

If we have an object that contains both data and an array…

* Enter “var a = {name: "Will", things: [0, 1, 2]};”, “var b = Object.assign({}, a, {name: "Fred"});” and “b.things.push(3);”
* Both a.things and b.things changes to [0, 1, 2, 3]. “b.things.push(3);” does not work.
* Enter “var a = {name: "Will", things: [0, 1, 2]};”, “var b = Object.assign({}, a, {name: "Fred"});” and “b.things = a.things.concat(3);” instead
* Now a.things remains unchanged, but b.things becomes [0, 1, 2, 3]

**Basic Redux Introduction - Redux Tutorial #3:**

Redux is quite simple, but an example on GitHub may seem to be very complicated

/\* ./package.json \*/

{

"name": "react-tutorials",

"version": "0.0.0",

"description": "",

"main": "webpack.config.js",

"dependencies": {

"axios": "^0.12.0",

"babel-core": "^6.17.0",

"babel-loader": "^6.2.0",

"babel-plugin-add-module-exports": "^0.1.2",

"babel-plugin-react-html-attrs": "^2.0.0",

"babel-plugin-transform-class-properties": "^6.3.13",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2015": "^6.3.13",

"babel-preset-react": "^6.3.13",

"babel-preset-stage-0": "^6.3.13",

"react": "^0.14.6",

"react-dom": "^0.14.6",

"redux": "^3.5.2", // Our Redux stuff

"redux-logger": "^2.6.1",

"redux-promise-middleware": "^3.2.0",

"redux-thunk": "^2.1.0",

"webpack": "^1.12.9",

"webpack-dev-server": "^1.14.1"

},

"devDependencies": {},

"scripts": {

"dev": "./node\_modules/.bin/webpack-dev-server --content-base src --inline --hot",

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

/\* ./webpack.config.js \*/

var debug = process.env.NODE\_ENV !== "production";

var webpack = require('webpack');

var path = require('path');

module.exports = {

context: path.join(\_\_dirname, "src"),

devtool: debug ? "inline-sourcemap" : null,

entry: "./js/client.js",

module: {

loaders: [

{

test: /\.jsx?$/,

exclude: /(node\_modules|bower\_components)/,

loader: 'babel-loader',

query: {

presets: ['react', 'es2015', 'stage-0'],

plugins: ['react-html-attrs', 'transform-class-properties',

'transform-decorators-legacy'],

}

}

]

},

output: {

path: \_\_dirname + "/src/",

filename: "client.min.js"

},

plugins: debug ? [] : [

new webpack.optimize.DedupePlugin(),

new webpack.optimize.OccurenceOrderPlugin(),

new webpack.optimize.UglifyJsPlugin({ mangle: false, sourcemap: false }),

],

};

<!-- ./src/index.html -->

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>React Tutorials</title>

<!-- change this up! http://www.bootstrapcdn.com/bootswatch/ -->

<link href="https://maxcdn.bootstrapcdn.com/bootswatch/3.3.6/cosmo/bootstrap.min.css" type="text/css" rel="stylesheet"/>

</head>

<body>

<div id="app"></div>

<script src="client.min.js"></script>

</body>

</html>

/\* ./src/js/client.js \*/

import { createStore } from "redux";

const reducer = function(state, action) {

if (action.type === "INC") {

return state + action.payload;

}

if (action.type === "DEC") {

return state - action.payload;

}

return state;

}

const store = createStore(reducer, 0);

// We pass a reducer and give the initial state. (We need a reducer and a store.)

// Normally, the initial state will be an object

// that has all the values of the application

store.subscribe(() => {

// We can listen to the store

console.log("Store changed, state = ", store.getState())

// The result:

// Store changed, state = 1

// Store changed, state = 3

// Store changed, state = 6

// Store changed, state = 2

})

store.dispatch({type: "INC", payload: 1}); // INC means increment

// The object dispatched goes through the reducer

// If "type" is replaced by "command", things do not work

// "payload" can be changed to anything else. However, "payload" is the standard name.

store.dispatch({type: "INC", payload: 2});

store.dispatch({type: "INC", payload: 3});

store.dispatch({type: "DEC", payload: 4});

**Multiple Reducers with Redux Reducers - Redux React Tutorial #4:**

We now have a basic Redux application. A real-world application has much more than one piece of data.

/\* ./src/js/client.js \*/

import { combineReducers, createStore } from "redux";

// We do not want one huge reducer because it will be hard to maintain

// It better to be separated in different parts in multiple files

// So, we have combineReducers from Redux

// I would live in a separate file

const userReducer = (state={}, action) => {

// The state will be the object user. By default, it is an empty object.

switch(action.type) {

case "SET\_NAME": {

return {...state, name: action.payload};

// This update the value of user. If we return nothing, there will be an error.

// We return a new state object instead of the original one

// If not, the first printout contains "age" as well

// Note that "return {name: action.payload, ...state};" will not work

// "name" will be overwritten

break;

}

case "SET\_AGE": {

return {...state, age: action.payload};

break;

}

}

return state;

}

// I would live in a separate file

const tweetsReducer = (state=[], action) => {

// The tweet portion of the data can act on the "CHANGE\_NAME" event as well,

// if there is such event

// One action can trigger multiple side effects

switch(action.type) {

case "ADD\_TWEET": {

return state.concat({

id: Date.now(), // Fake an ID by using a timestamp

text: action.payload,

});

break;

}

}

return state;

}

const reducers = combineReducers({

// The data we are modifying and the reducer functions

// that are responsible for the modification

user: userReducer,

tweets: tweetsReducer

})

const store = createStore(reducers);

// Originally, this is

// "const store = createStore(reducers, { /\* ... (The default values) \*/ });"

// We do not need to provide any default values here

// This is taken care by the reducer function now

store.subscribe(() => {

console.log("Store changed, state = ", store.getState())

})

store.dispatch({type: "SET\_NAME", payload: "Will"});

store.dispatch({type: "SET\_AGE", payload: 35});

store.dispatch({type: "SET\_AGE", payload: 34});

store.dispatch({type: "ADD\_TWEET", payload: "OMG LIKE LOL"});

store.dispatch({type: "ADD\_TWEET", payload: "I am so like seriously like totally like right now"});

**Redux Middleware Tutorial - Redux Tutorial #5:**

Redux middleware works similar to Express.js or Node.js application

Middleware intercepts every action that comes through it and then it can either modify that action or cancel that action

Middleware is also the solution for asynchronous actions in Redux like AJAX (Asynchronous JavaScript and XML (Extensible Markup Language)) XHR (XMLHttpRequest) request

/\* ./src/js/client.js \*/

import { applyMiddleware, createStore } from "redux";

const reducer = (initialState=0, action) => {

if (action.type === "INC") {

return initialState + 1;

} else if (action.type === "DEC") {

return initialState - 1;

} else if (action.type === "MULT") {

throw new Error("AHHHH!!");

}

return initialState;

}

const logger = (store) => (next) => (action) => {

// This is the syntax of creating a middleware

// We basically have a chain of thunks

// Middleware goes before the reducer

// Most middleware can be retrieved from npm

console.log("Logged", action);

// action.type = "DEC"; // One can modify the action type here

return next(action);

// Without this line, we are essentially terminating the action

};

const errorHandler = (store) => (next) => (action) => {

try {

return next(action);

} catch(e) {

console.log("ERROR!", e);

}

};

const middleware = applyMiddleware(logger, errorHandler);

// We just pass the middleware into applyMiddleware function

const store = createStore(reducer, 1, middleware);

// To add a middleware, one just needs to provide the third argument here

store.subscribe(() => {

console.log("Store changed, state = ", store.getState());

})

store.dispatch({type: "INC"});

store.dispatch({type: "INC"});

store.dispatch({type: "INC"});

store.dispatch({type: "DEC"});

store.dispatch({type: "DEC"});

store.dispatch({type: "DEC"});

store.dispatch({type: "MULT"})

store.dispatch({type: "DEC"})

**Redux Async Actions - Redux Tutorial #6:**

It is easy for Redux to handle asynchronous actions because Redux can be tied to React

React is a view representation of the state of the store

Redux move all of the representations out of the components and into the store, so one basically will never use state again

/\* ./src/js/client.js \*/

import { applyMiddleware, createStore } from "redux";

import axios from "axios"; // We want to have XHR request

import logger from "redux-logger";

import thunk from "redux-thunk";

const initialState = {

fetching: false,

// This can be a piece of useful information. It can be reflected in the UI.

fetched: false,

users: [],

error: null

}

const reducer = (state=initialState, action) => {

switch (action.type) {

case "FETCH\_USERS\_START": {

return {...state, fetching: true}

break;

}

case "FETCH\_USERS\_ERROR": {

// If a wrong URL is entered, there will be an error

return {...state, fetching: false, error: action.payload}

break;

}

case "RECEIVE\_USERS": {

return {

...state,

fetching: false,

fetched: true,

users: action.payload

// Some users can be got

}

break;

}

}

return state;

}

const middleware = applyMiddleware(thunk, logger());

// "redux-logger" generate a nice-looking log

// The parentheses after "logger" is necessary

// "thunk" helps us to handle asynchronous actions

const store = createStore(reducer, middleware);

// We dispatch a function

store.dispatch((dispatch) => {

dispatch({type: "FETCH\_USERS\_START"});

// There is no payload

axios.get("http://rest.learncode.academy/api/wstern/users")

// Try "http://rest.learncodeabc.academy/api/wstern/users"

// axios.get returns a Promise object,

// which has a then method for one to supply the callback function

.then((response) => {

// After there is a response, we dispatch something

dispatch({type: "RECEIVE\_USERS", payload: response.data});

})

// We do something asynchronous. We can show a loading bar, etc.

// There are some APIs from the URL

// "wstern" can be changed

.catch((err) => {

// If there is an error, we can also catch that

dispatch({type: "FETCH\_USERS\_ERROR", payload: err})

})

}); // This is how we handle asynchronous actions

If we use “redux-promise-middleware”, the way we dispatch something will be different

/\* ./src/js/client.js \*/

import { applyMiddleware, createStore } from "redux";

import axios from "axios";

import logger from "redux-logger";

import thunk from "redux-thunk";

import promise from "redux-promise-middleware" // New

const initialState = {

fetching: false,

fetched: false,

users: [],

error: null

}

const reducer = (state=initialState, action) => {

switch (action.type) {

case "FOO\_PENDING": { // Instead of "FETCH\_USERS\_START"

return {...state, fetching: true}

break;

}

case "FOO\_REJECTED": { // "Instead of FETCH\_USERS\_ERROR"

return {...state, fetching: false, error: action.payload}

break;

}

case "FOO\_FULFILLED": { // Instead of "RECEIVE\_USERS"

return {

...state,

fetching: false,

fetched: true,

users: action.payload

}

break;

}

}

return state;

}

const middleware = applyMiddleware(promise(), thunk, logger());

const store = createStore(reducer, middleware);

store.dispatch({

type: "FOO", // Or to be clearer, "FETCH\_USERS"

payload: axios.get("http://rest.learncodeabc.academy/api/wstern/users")

// The payload is of the type "promise"

})

**Connecting React & Redux - Redux Tutorial #7:**

We would like to tie Redux to React app application

/\* ./package.json \*/

{

"name": "react-tutorials",

"version": "0.0.0",

"description": "",

"main": "webpack.config.js",

"dependencies": {

"axios": "^0.12.0",

"babel-core": "^6.18.2",

"babel-loader": "^6.2.0",

"babel-plugin-add-module-exports": "^0.1.2",

"babel-plugin-react-html-attrs": "^2.0.0",

"babel-plugin-transform-class-properties": "^6.3.13",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2015": "^6.3.13",

"babel-preset-react": "^6.3.13",

"babel-preset-stage-0": "^6.3.13",

"react": "^0.14.6",

"react-dom": "^0.14.6",

"react-redux": "^4.4.5", // New

"redux": "^3.5.2",

"redux-logger": "^2.6.1",

"redux-promise-middleware": "^3.2.0",

"redux-thunk": "^2.1.0",

"webpack": "^1.12.9",

"webpack-dev-server": "^1.14.1"

},

"devDependencies": {},

"scripts": {

"dev": "./node\_modules/.bin/webpack-dev-server --content-base src --inline --hot",

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

/\* ./webpack.config.js \*/

/\* ... \*/

<!-- ./src/index.html -->

<!-- ... -->

/\* ./src/js/client.js \*/

import React from "react"

import ReactDOM from "react-dom"

import Layout from "./components/Layout"

import store from "./store"

const app = document.getElementById('app')

ReactDOM.render(<Layout/>, app);

/\* ./src/js/components/Layout.js \*/

import React from "react"

export default class Layout extends React.Component {

render() {

return null;

}

}

/\* ./src/js/store.js \*/

import { applyMiddleware, createStore } from "redux"

import logger from "redux-logger"

import thunk from "redux-thunk"

import promise from "redux-promise-middleware"

import reducer from "./reducers"

// This will import "./reducers/index.js"

const middleware = applyMiddleware(promise(), thunk, logger())

export default createStore(reducer, middleware)

/\* ./src/js/reducers/index.js \*/

import { combineReducers } from "redux"

import tweets from "./tweetsReducer"

import user from "./userReducer"

export default combineReducers({

tweets,

user,

})

/\* ./src/js/reducers/tweetsReducer.js \*/

export default function reducer(state={

tweets: [],

fetching: false,

fetched: false,

error: null,

}, action) {

switch (action.type) {

case "FETCH\_TWEETS": {

return {...state, fetching: true}

// We always return a new object

}

case "FETCH\_TWEETS\_REJECTED": {

return {...state, fetching: false, error: action.payload}

}

case "FETCH\_TWEETS\_FULFILLED": {

return {

...state,

fetching: false,

fetched: true,

tweets: action.payload,

}

}

case "ADD\_TWEET": {

return {

...state,

tweets: [...state.tweets, action.payload],

}

}

case "UPDATE\_TWEET": {

const { id, text } = action.payload

const newTweets = [...state.tweets]

const tweetToUpdate = newTweets.findIndex(tweet => tweet.id === id)

newTweets[tweetToUpdate] = action.payload;

return {

...state,

tweets: newTweets,

}

}

case "DELETE\_TWEET": {

return {

...state,

tweets: state.tweets.filter(tweet => tweet.id !== action.payload),

}

}

}

return state

}

/\* ./src/js/reducers/userReducer.js \*/

export default function reducer(state={

user: {

id: null,

name: null,

age: null,

},

fetching: false,

fetched: false,

error: null,

}, action) {

switch (action.type) {

case "FETCH\_USER": {

return {...state, fetching: true}

}

case "FETCH\_USER\_REJECTED": {

return {...state, fetching: false, error: action.payload}

}

case "FETCH\_USER\_FULFILLED": {

return {

...state,

fetching: false,

fetched: true,

user: action.payload,

}

}

case "SET\_USER\_NAME": {

return {

...state,

user: {...state.user, name: action.payload},

}

}

case "SET\_USER\_AGE": {

return {

...state,

user: {...state.user, age: action.payload},

}

}

}

return state

}

/\* ./src/js/actions/tweetsActions.js \*/

import axios from "axios";

export function fetchTweets() {

return function(dispatch) {

dispatch({type: "FETCH\_TWEETS"});

/\*

http://rest.learncode.academy is a public test server,

so another user's experimentation can break your tests

If you get console errors due to bad data:

- change "reacttest" below to any other username

- post some tweets to http://rest.learncode.academy/api/yourusername/tweets

\*/

axios.get("http://rest.learncode.academy/api/reacttest/tweets")

.then((response) => {

dispatch({type: "FETCH\_TWEETS\_FULFILLED", payload: response.data})

})

.catch((err) => {

dispatch({type: "FETCH\_TWEETS\_REJECTED", payload: err})

})

}

}

export function addTweet(id, text) {

// Redux is not opinionated on how one fires actions

// You just need to make sure that every action has a type

return {

type: 'ADD\_TWEET',

payload: {

id,

text,

},

}

}

export function updateTweet(id, text) {

return {

type: 'UPDATE\_TWEET',

payload: {

id,

text,

},

}

}

export function deleteTweet(id) {

return { type: 'DELETE\_TWEET', payload: id}

}

/\* ./src/js/actions/userActions.js \*/

export function fetchUser() {

return {

type: "FETCH\_USER\_FULFILLED",

payload: {

name: "Will",

age: 35,

}

}

}

export function setUserName(name) {

return {

type: 'SET\_USER\_NAME',

payload: name,

}

}

export function setUserAge(age) {

return {

type: 'SET\_USER\_AGE',

payload: age,

}

}

We now want to tie Redux to React...

/\* ./src/js/client.js \*/

import React from "react"

import ReactDOM from "react-dom"

import { Provider } from "react-redux" // New

import Layout from "./components/Layout"

import store from "./store" // We need the store

const app = document.getElementById('app')

ReactDOM.render(

<Provider store={store}>

<Layout/>

</Provider>,

app);

// To tie Redux to React, there are two steps

// One of the step is to wrap the top-level component with React-Redux provider

// Another step is to add the "@connect(/\* ... \*/)" part in Layout.js

// Now any component down the chain can import data from the Redux store

// and can also dispatch store actions

/\* ./src/js/components/Layout.js \*/

import React from "react"

import { connect } from "react-redux"

import { fetchUser } from "../actions/userActions"

import { fetchTweets } from "../actions/tweetsActions"

@connect((store) => {

// "connect" is going to take two functions

// The first function is for getting store values in as props

// The second function is supposed to be the one that maps dispatch to props

return {

// The return items will be regarded as props

// Just pull in the data we want

user: store.user.user,

userFetched: store.user.fetched,

tweets: store.tweets.tweets

}

})

// We transpile with ES6

// Note that in package.json and webpack.config.js,

// we have "babel-plugin-transform-decorators-legacy", so we can use decorators

// Using decorator is a great way to wrap a component

// Here, connect wraps the Layout component

// This is a simple way to inject props into layout without messing with

// the Layout component

export default class Layout extends React.Component {

componentWillMount() {

// We want to fetch the user

// Actually, nothing asynchronous happens here

this.props.dispatch(fetchUser());

}

fetchTweets() {

this.props.dispatch(fetchTweets());

// We have asynchronous action here

}

render() {

const { user, tweets } = this.props;

const mappedTweets = tweets.map(tweet => <li>{tweet.text}</li>)

// Do not forget the curly braces!

// console.log(this.props);

// We can see that props contains a user object and a dispatch function

// so that we can dispatch some actions

if (!tweets.length) {

return <button onClick={this.fetchTweets.bind(this)}>Load tweets</button>

}

// We would like to show the data fetched

return (

<div>

<h1>Hello, {user.name}.</h1>

<u1>{mappedTweets}</u1>

</div>

);

}

}

The smart component use “connect” and then they pass everything down as props to all the dumb components. The dumb components are not aware of Redux at all, and some of the components are not even aware of the methods like this.props.dispatch.

We want to have as few smart components as possible, but we also want to avoid passing props ten levels down so that they can get used. It can be frustrating to pass props many times and it also makes React component unit test harder.