Thermo Fisher

Notification Manager Using React And Redux

Project Summary

There was an already built notification manager made using Angular js framework but it has been facing some binding issues when that was being used with another framework.

So on further exploration it concluded that that same notification manager can be made using React and Redux.

Introduction

What is react?

React JS — is a JavaScript library for building user interfaces developed by Facebook. It actually populates the virtual DOM idea.

Instagram and Facebook are developed using react.

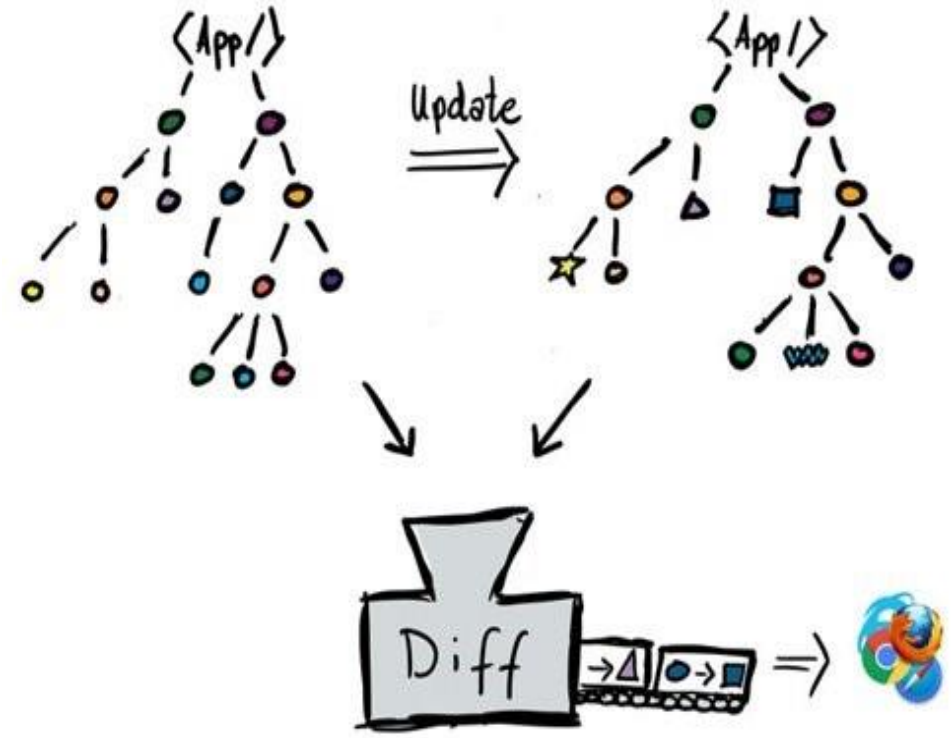
Why react?

Think about modern social networks like Twitter, Facebook or Pinterest. After scrolling a little bit, the user will have tens of thousands of nodes. Interact with them efficiently is a huge problem. Try to move a 1000 divs 5 pixel left for example. It may take more than a second. It's a lot of time for the modern web. You can optimize the script and use some tricks, but in the end, it's a pain to work with huge pages and dynamic UI.

Magic of React - *Virtual DOM*

Rather than touching the DOM directly, we're building an abstract version of it. That's it. We working with some kind of lightweight copy of our DOM. We can change it as we want and then save to our real DOM tree. While saving we should compare, find difference and change (re-render) what should be changed.

It is much faster than working directly with DOM, because it doesn't require all the heavyweight parts that go into a real DOM. It works great, but only if we are working with it in a right way.



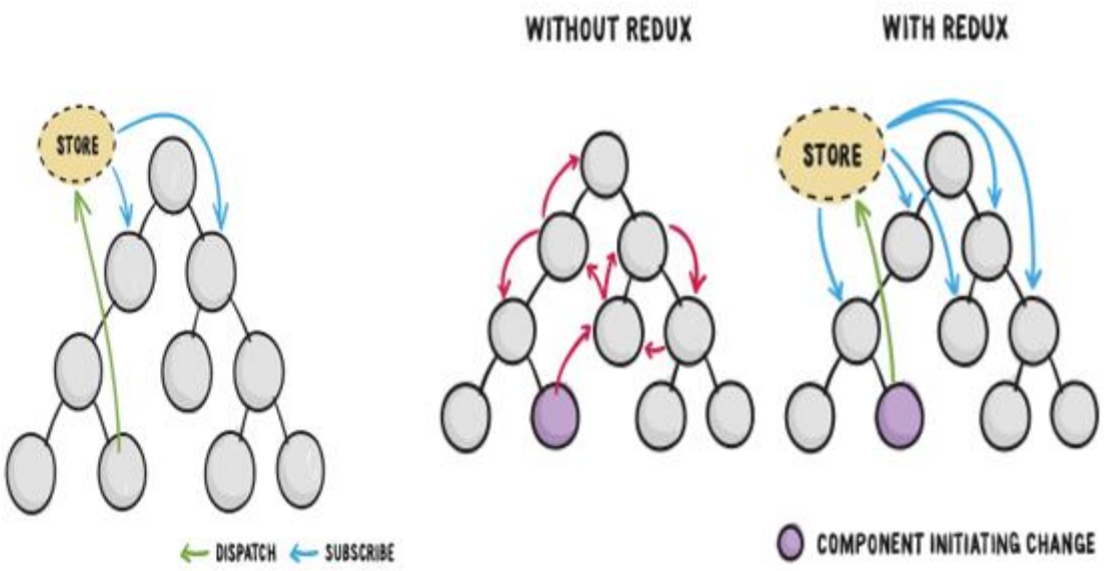
So what actually happen is whenever any node encounter any change then we don't have to worry about where all we need those changes , react helps us in this direction. React runs a diff algorithm and finds the minimum changes needed to be done and thus do changes only to those component rather than the whole webpage.

What is Redux?

Redux is a tool for managing both data-state and UI-state in JavaScript applications.

React doesn't recommend direct component-to-component communication this way. Even if it did have features to support this approach, it's considered poor practice by many because direct component-to-component communication is error prone.

This is where Redux comes in handy. Redux offers a solution of storing all your application state in one place, called a "store". Components then "dispatch" state changes to the store, not directly to other components. The components that need to be aware of state changes can "subscribe" to the store.



Special Features of Redux:

Single source of truth-- app state stored in an object tree inside a single store

State is read only-- state is immutable and the only way to mutate the state is to emit an action and a new state object will be returned.

Mutation are written as pure functions-- specify how the state tree is transformed by actions using pure reducers.

Thinking in React

React is the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes us think about apps as we build them.

Start With A Mock

Imagine that we already have a JSON API and a mock from our designer. The mock looks like this:

React To Do Stuff

Enter Add to list

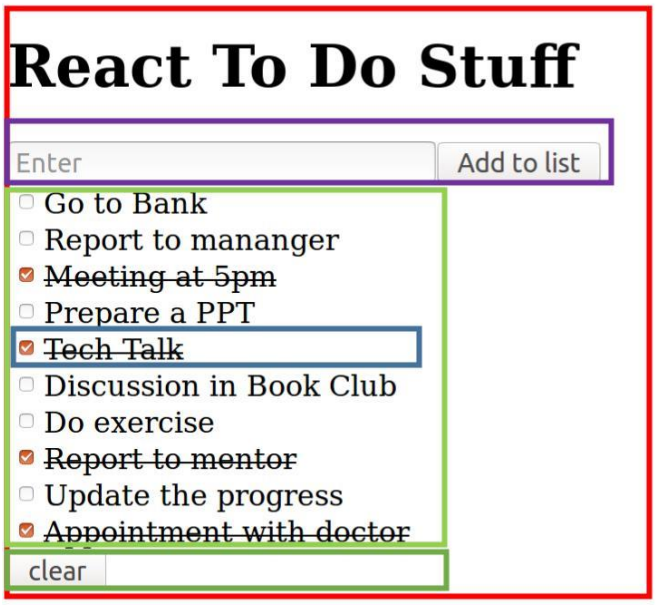
- [ ] Go to Bank
- [ ] Report to mananger
- [x] ~~Meeting at 5pm~~
- [ ] Prepare a PPT
- [x] ~~Tech Talk~~
- [ ] Discussion in Book Club
- [ ] Do exercise
- [x] ~~Report to mentor~~
- [ ] Update the progress
- [x] ~~Appointment with doctor~~

clear

Step 1: Break The UI Into A Component Hierarchy

Think about what all component will be require and an easy way to do that is to draw boxes around each component.

But how do we know what should be its own component? Just use the same techniques for deciding if you should create a new function or object. One such technique is that a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.



You'll see here that we have five components in our simple app. We've italicized the data each component represents.

1. **App (red):** contains the entirety of the example
2. **SearchBar (violet):** receives all *user input*
3. **Listable (green):** displays and filters the *data collection* based on *user input*
4. **ListableRow (dark blue):** displays a heading of each todo list
5. **ClearListable(dark green):** clears all the checked todolist.

Now that we've identified the components in our mock, let's arrange them into a hierarchy. This is easy. Components that appear within another component in the mock should appear as a child in the hierarchy:

- App

  --SearchBar

  --Listable

  1.ListableRow

  --ClearListable

Step 2: Build A Static Version in React

Now that we have our component hierarchy, it's time to implement our app. The easiest way is to build a version that takes your data model and renders the UI but has no interactivity.

To build a static version of your app that renders your data model, we'll want to build components that reuse other components and pass data using *props*. *props* are a way of passing data from parent to child. D**on't use state at all** to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, we don't need it.

We can build top-down or bottom-up. That is, we can either start with building the components higher up in the hierarchy (i.e. starting with App) or with the ones lower in it (ListableRow). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up and write tests as you build.

At the end of this step, we have a library of reusable components that render our data model. The components will only have render() methods since this is a static version of your app. The component at the top of the hierarchy (App) will take our data model as a prop. If we make a change to our underlying data model and call ReactDOM.render() again, the UI will be updated. It's easy to see how your UI is updated and where to make changes since there's nothing complicated going on. React's **one-way data flow** (also called *one-way binding*) keeps everything modular and fast.

Step 3: Identify The Minimal (but complete) Representation Of UI State

To make your UI interactive, we need to be able to trigger changes to our underlying data model. React makes this easy with **state**.

To build our app correctly, we first need to think of the minimal set of mutable state that our app needs. The key here is *Don't Repeat Yourself*. We have to figure out the absolute minimal representation of the state our application needs and compute everything else we need on-demand. For example, if we're building a TODO list, just keep an array of the TODO items around; don't keep a separate state variable for the count. Instead, when we want to render the TODO count, simply take the length of the TODO items array.

Think of all of the pieces of data in our example application. We have:

- The original list of todo list.
- The search text the user has entered
- The value of the checkbox
- Clear button to remove all the checked one's,

Let's go through each one and figure out which one is state. Simply ask three questions about each piece of data:

6. Is it passed in from a parent via props? If so, it probably isn't state.
7. Does it remain unchanged over time? If so, it probably isn't state.
8. Can you compute it based on any other state or props in your component? If so, it isn't state.

The original todolist is fetched from some api and stored in props so that's not state. The search text and the checkbox seem to be state since they change over time and can't be computed from anything. And finally, the displayed todo list isn't state because it can be computed by combining the original todolist with the search text and value of the checkbox.

So finally, our state is:

- The search text the user has entered
- The value of the checkbox

Step 4: Identify Where Your State Should Live

OK, so we've identified what the minimal set of app state is. Next, we need to identify which component mutates, or *owns*, this state.

Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state. **This is often the most challenging part for newcomers to understand,** so follow these steps to figure it out:

For each piece of state in your application:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

Let's run through this strategy for our application:

- Listable needs to filter the TODO list based on state and SearchBar needs to display the search text and clicking ADD button need to add that to the TODO list.
- The common owner component is App.
- It conceptually makes sense for the filter text and checked value to live in App component.

Here it is all and we are done with react now one can easily build any react application.

WHY Redux

So how does the component interact with each other?

- By **PROPS** and **STATE**.
- Props - data flows from parent to child. Props are immutable.

Components receive data from the parent.

- What happens when a component receives data from someone other than the parent? What if the user inputs data directly to the component? Well, this is why we have state.
- State - Props shouldn't change, so state steps up. So state is used so that a component can keep track of information in between any renders that it does.

Sometimes distant parts of the app want to have access to the same state, for example, if you cache fetched data and want to consistently update it everywhere at the same time. In this case, if you follow the React model, you'll end up with a bunch of very large components at the top of the component tree that pass a **myriad of props** down through some intermediate components that don't use them, just to reach a few leaf components that actually care about that data.

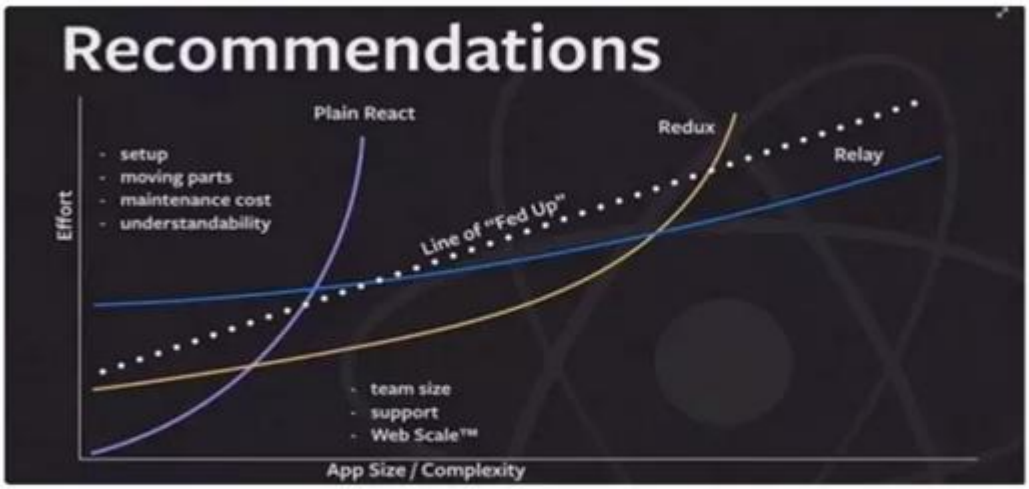
Here comes Redux

- All the State will be stored in a STORE.
- And where ever the component may be , it will get the required state change.
- Redux is like "a backend's database on the client-side where you store all the required information that are required in order to generate the view". It helps you to query SELECT, INSERT and UPDATE the data (a record) into the JSON's database (called also single-state tree).

So when to use redux?

- If we have an app that remembers the state only **locally** and **synchronizes with the server periodically**, or **only saves when user clicks a button or something**, use **Redux**, especially if you need the undo/redo functionality.
- It really shines when you do have to keep track of the state of many and many components that need to send messages and update dependently.
- Avoid many props!!!.
- It will have **predictable state**.
- Managing them becomes easy.

C) WHY YOU NEED REDUX? When not to use Redux.



The graph can be useful in order to judge when to use Redux. My current rule is: if you want to make a component which will be published on NPM, then use only Plain React. If you build any application, then use Redux because it's easy to learn and you can make sure that your client-side architecture is first-grade even if the React Redux project grows.

Starting the project

1.Environment Setup

Step 1 - Install Global Packages

1. npm install -g babel
2. npm install -g babel-cli

Create empty **package.json** file inside by running **npm init** from the **command prompt**

Step 2 - Add Dependencies and plugins

Install **webpack** and **webpack-dev-server**.

1. npm install webpack --save
2. npm install webpack-dev-server –save

Since we want to use React, we need to install it first.

1. npm install react –save
2. npm install react-dom –save

We already mentioned that we will need some **babel** plugins so let's install it too.

1. npm install babel-core --save
2. npm install babel-loader --save
3. npm install babel-preset-react --save
4. npm install babel-preset-es2015 --save

So we are all set to use react.

In order to add our own styles using bootstrap , we need to install react-bootstrap

- npm install --save react-bootstrap

In order to apply it to our component we need to install className module:

- npm install --save react-classname-module

In order our webpack to be able to load .css file we need to install css-loader in webpack:

1. npm install react react-dom bootstrap react-bootstrap babel-preset-react –save
2. npm install webpack css-loader style-loader file-loader url-loader babel-core babel-loader babel-preset-es2015 --save-dev

Then, add the loaders in your **webpack.conf.js:**

```
module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
```

```
      query: {
        presets: ['es2015', 'react']
      }
    },
{
  test: /\.css$/,
  loader: "style-loader!css-loader"
},
{
  test: /\.png$/,
  loader: "url-loader?limit=100000"
},
{
  test: /\.jpg$/,
  loader: "file-loader"
},
{
  test: /\.(woff|woff2)(\?v=\d+\.\d+\.\d+)?$/,
  loader: 'url?limit=10000&mimetype=application/font-woff'
},
{
  test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,
  loader: 'url?limit=10000&mimetype=application/octet-stream'
},
{
  test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,
  loader: 'file'
},
{
  test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,
  loader: 'url?limit=10000&mimetype=image/svg+xml'
}
```

```
    ]
  },
```

Include popular icons in your React projects easily with react-icons, which utilizes ES6 imports that allows you to include only the icons that your project is using.

- npm install react-icons –save

Including react-scrollbar separately install

- npm install react-scrollbar --save

In order to be able to make XMLHttpRequests from the browser install:

- npm install axios –save

In order to create basic CSS transition and animation install

- npm install react-addons-css-transition-group

In order to use redux install

- npm install --save redux

In order to conect react with redux install

- npm install --save react-redux

2.Including the script

In the main html page include the jquery script , bootstrap cdn and the bundle name which we will get from webpack after building everything.

Suppose we want to include the notification manager in the *div* with id as '***app'*** so in the project a *button* with id as ***'changeName'*** has been added which on clicking will render our ***PopupTable*** component.

Here is the script:

```
$(document).ready(function() {
  //Handle the click event on the button
  $('#changeName').click(function(e) {
    e.preventDefault()
```

```
    reactComponents('app').message.setState({
      userToMessage: 'It Changed!'
    });

  })
})
```

3.Working into the Code

There is only one main component that is PopupTable.jsx and an animation has been added that will notify the user about the new notification

To run the server type following in the command line:

----> webpack-dev-server

4.Building the bundle for production mode

Before building the bundle we need to add the **plugin** to webpack and **devTool** too:

```
const webpack = require('webpack'); // this is constant

plugins: [
  new webpack.DefinePlugin({
    'process.env': {
      'NODE_ENV': JSON.stringify('production')
    }
  }),
```

And add the devTool too in the configuration:

```
devtool: 'source-map',
```

In order to build the bundle for the production type the following in command line:

--->webpack –progress –p

Here when the bundling of project was done the bundle size was approximately ~500kb

5.Building the bundle using grunt

One folder has been created grunt which has webpack.js file whose configuration is like:

```
const webpack = require('webpack');
var plugins = [
  new webpack.DefinePlugin({
    'process.env': {
      'NODE_ENV': JSON.stringify('production')
    }
  }), // Helps identify common parts of a require hierarchy
    new webpack.optimize.UglifyJsPlugin(),
    new webpack.optimize.AggressiveMergingPlugin(),
    ];
var webpackModules = {
  loaders: [
  {
          test: /\.jsx?$/,
          exclude: /node_modules/,
          loader: 'babel-loader',

          query: {
             presets: ['es2015', 'react']
          }
        },// loaders can take parameters as a querystring

      {
        test: /\.css$/,
        loader: "style-loader!css-loader"
      },
      {
        test: /\.png$/,
        loader: "url-loader?limit=100000"
```

```
      },
      {
        test: /\.jpg$/,
        loader: "file-loader"
      },
      {
        test: /\.(woff|woff2)(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'url?limit=10000&mimetype=application/font-woff'
      },
      {
        test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'url?limit=10000&mimetype=application/octet-stream'
      },
      {
        test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'file'
      },
      {
        test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'url?limit=10000&mimetype=image/svg+xml'
      }
  ]
};
var config = {
    dev: {
     entry: './main.js',
   output: {
      filename: './build/bundle.js',
   },
   plugins: plugins,
    module: webpackModules,
    // Configure the console output
```

```
    stats: {
      colors: true,
     modules: true,
    },
    progress: true,
    keepalive: true,
    bail: true,
    devtool: 'source-map'
  },
  // Exclude any dev like configuration for any production like env
  prerelease: {
   entry: './main.js',

   output: {

     filename: './build/bundle.js',
   },
 }
}
 module.exports = config;
```

Here is the Gruntfile.js:

```
module.exports = function(grunt) {
  require('load-grunt-tasks')(grunt);
    grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    });
    grunt.config( 'webpack', require('./grunt/webpack.js') );
    grunt.registerTask('run',function(){
    console.log("running");
  });
```

```
    grunt.registerTask('dev', ['webpack:dev']);
}
```

Build the bundle using grunt via webpack by typing the following in command line:

---->grunt dev

References:

- https://www.tutorialspoint.com/reactjs/index.htm
- http://redux.js.org/
- https://react-bootstrap.github.io/
- https://gorangajic.github.io/react-icons/index.html
- https://www.npmjs.com/package/react-scrollbar
- https://www.npmjs.com/package/react-loaders
- http://moduscreate.com/optimizing-react-es6-webpack-production-build/
- https://www.quora.com/Should-I-learn-Redux-if-I-know-React-and-why-should-I-learn-it
- https://github.com/uberVU/react-guide/blob/master/props-vs-state.md
- https://cardlife.blog/what-is-virtual-dom-c0ec6d6a925c
- https://www.youtube.com/watch?v=BYbgopx44vo
- https://www.npmjs.com/package/axios