**REACT JS TUTORIAL #1 - ReactJS JavaScript Introduction & Workspace Setup:**

ReactJS changes the way you approach building web apps.

Some Babel stuff is installed. This can be seen in the package.json file. Babel is a tool for transpiling (轉譯，將一種語言編寫的源代碼轉換為另一種具有相同抽象層次的語言) the React code (.jsx) and ES6 code at the same time. The JS file gets through the Babel loader. Babel will transpile React JSX and ES6 code. There are also some bleeding edge features that might not be fully incorporated into ES2015.

We are going to convert some React attributes to HTML attributes and add class properties. We are also going to use decorators.

/\* ./package.json (Actually, comments are not supported in JSON file) \*/

{

"name": "react-tutorials",

"version": "0.0.0",

"description": "",

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

"dependencies": {

// Some Babel stuffs are installed

"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",

"webpack": "^1.12.9",

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

},

"devDependencies": {},

"scripts": {

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

/\* Entering “npm run dev” will now be equivalent to entering

"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 = {

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

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

// false -> null

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

// Original version: "entry: "./js/client.js""

entry: \_\_dirname + "/src/js/client.js",

module: {

loaders: [

{

// jsx -> js

test: /\.js?$/,

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

loader: 'babel-loader',

query: {

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

plugins: ['react-html-attrs', 'transform-decorators-legacy',

'transform-class-properties'],

}

}

]

},

output: {

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

filename: "client.min.js"

},

plugins: debug ? [] : [

new webpack.optimize.DedupePlugin(),

new webpack.optimize.OccurrenceOrderPlugin(),

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>

<!-- The entire app is going to render into here -->

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

<!-- “client.min.js” is also loaded -->

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

</body>

</html>

* Run “npm install”
* Run “node\_modules\.bin\webpack” (or ““node\_modules/.bin/webpack”” in order to use an older version of Webpack and avoid problems of incompatibility) and “node\_modules\.bin\webpack --watch”. (“node\_modules\.bin\webpack --watch" will watch our code for changes. If there are any changes, it transpiles everything.)
* Open ./src/index.html to see the result

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

import React from "react";

// “react-dom” is the rendering engine

import ReactDOM from "react-dom";

// Original version: “import Layout from "./components/Layout";”

class Layout extends React.Component {

render() {

return (

<h1>It works!</h1>

);

}

}

// Get the app element. See “./src/index.html”.

const app = document.getElementById('app');

ReactDOM.render(<Layout/>, app); // Render the layout into “app”

With “react-dom”, you do not have to render things to HTML. You can render things to a string, you can render them to the DOM (Document Object Model), which is our active web page.

* Actual rendering engine being separated from the React syntax is a very cool concept that makes the code one writes versatile even towards native apps

The core of React is that everything is a component. If you are looking in (把注意力轉向) ./src/index.html, you will find elements like div (it defines a division or a section in an HTML document). This is also the concept behind React components. In this case, we have a layout component.

* “<h1>It works!</h1>” is going to transpile to something like “var div = document.createElement(“div”); div.innerHTML = “some content”;”
* For the HTML syntaxes to be highlighted correctly, one needs to install a JSX plugin

If you want to render a component, you use it as if it was an HTML tag

We want to have live reload

* Enter “npm install -S webpack-dev-server”. This is kind of the standard way of doing things. (Actually, webpack-dev-server should be installed after running “npm install”.)
* Install the thing globally as well (npm install -g webpack-dev-server) so that one can use the Webpack command.
* Enter “webpack-dev-server --content-base src” to serve the “src” folder. “index.html” will be a root.
* Type “localhost:8080” in the browser address bar to see the result
  + Type “localhost:8080/webpack-dev-server/index.html” in the bar. An extra loader bar can be seen. Every time the code is changed, the iframe will automatically be updated.
  + Enter “webpack-dev-server --content-base src --inline --hot” or just “npm run dev” (this is defined in “./package.json”) so that one can just type “localhost:8080” in the browser address bar
* Change “"dev": "webpack-dev-server --content-base src --inline --hot"” to “"dev": "./node\_module/.bin/webpack-dev-server --content-base src --inline --hot"” if one wants to avoid installing webpack-dev-server globally
* To uninstall global webpack-dev-server, type “npm uninstall -g webpack-dev-server”. (It is cleaner to install things locally to one’s app so that the exact version of the package is installed.)

**Side note: What local host 8080 is:**

Short answer: “this machine, port 8080”.

Slightly longer short answer: an application, typically a web application, that is running on the user’s own computer, utilizing UDP or typically TCP port 8080. If such software is, in fact running, and if it is web software (HTTP), you should be able to access it using the URL: http://localhost:8080/

Complete answer:

On the Internet, everything connected has an IP address which uniquely identifies Internet devices. IP addresses also come in two flavours: public and private.

* The public addresses are accessible to anyone on the Internet.
* The private addresses are only accessible to your LAN, or Local Area Network.
* In any environment, there is usually a device called a router that translates Internet packets between the local network and the public Internet. Every such device is able to reuse private IP addresses within their own private environment—homes reuse the same IP addresses that company employees use at their workstations.
* For example, 192.168.1.100 is a private IP address. It will never be a public IP address, and the same IP address 192.168.1.100 exists hundreds of thousands if not millions of times all over the world. NAT translation, a function of the router, ensures that your Internet messages, called packets, have their private IP address self-identity swapped out for a shared public IP address, and vice-versa for packets coming back.

Most public IP addresses have names, called DNS hostnames, and many private IP addresses do, too.

* For example, www.google.com is a DNS hostname that has an IP address. If you open a command prompt and type “ping www.google.com”, you will find that your computer translated that hostname to an IP address, such as 172.217.2.228, and it is sending ICMP (Internet Control Message Protocol) packets to it and reporting the time taken to get a response.

The universal hostname name for “myself” on any Internet device is “localhost”, and the universal IP address for “localhost” is the private IP address 127.0.0.1. This address is not recognizable to your router; only your device/computer ever “sees” it.

The often optional but nonetheless conventional “www.” prefix part of the typical website hostname suggests that the Internet device, or host, is responsible for serving a World Wide Web application.

* There are many types of applications that run on the Internet, not just World Wide Web, including, for example
  + E-mail
  + FTP (file transfer protocol)
* When data packets are sent over the Internet to devices, they almost always have a port assigned to them. This is a number, ranging generally from 0 to 64k (65536), except for some reserved numbers and ranges.
* The World Wide Web application uses port 80. E-mail, which is sent over the SMTP protocol, uses port 25 (among others, such as 465 and 587)
* Ports are just identifiers on the packets indicating what application the data packets are intended for. Network software will bind to and listen on the given port.

Often when web software developers develop or deploy web applications, they will bind their app on an alternate port, not just port 80. The port 8080 is often used for this purpose.

* It is just a “random” port that web software is listening on because someone chose to set it up that way.
* Only one software application can listen on any port on the same device, so the uniqueness of the port number is required.
* The number could have been almost any port, such as, say, 2018, or 12345. But 8080 is handy because it’s distinct from port 80, the default port for the World Wide Web, and yet it looks a lot like 80.

So, “localhost 8080” means “this computer, or 127.0.0.1, application on port 8080”, or if it is web-related (and typically is), http://localhost:8080/. The construction of that URL is: {scheme}://{hostname}:{port}{path}, where {scheme} is “http”, {hostname} is “localhost”, {port} is “8080”, and {path} is “/”.

Note that the HTTP protocol, which is the protocol used by the World Wide Web, specifies that the hostname is included as a request header, or passed directly to the application. So, http://127.0.0.1:8080/, where “localhost” is replaced with the IP address “127.0.0.1”, will reach the same application, but it might not work correctly, or not the same, if the web software is waiting for a request for the hostname “localhost”.

**REACT JS TUTORIAL #2 - ReactJS Components & Rendering:**

Just like an HTML, everything has to have one parent DOM element. They cannot have two parent DOM elements. Having a pair of “<h1> … </h1>” only will not work. A pair of “<h1> … </h1>” should be wrapped inside “div” tags.

One can import dynamic information in the curly braces after render(). The JavaScript variables/constants should be wrapped by a pair of curly braces in the JSX portion.

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

import React from "react";

import ReactDOM from "react-dom";

class Layout extends React.Component {

render() {

const name = "Will";

return (

// This is the JSX portion

<h1>It’s {name}!</h1>

// <h1> It’s {3+2}!</h1>

// <h1> It’s {(function(){return 3;})()} // This is a self-invoking function

);

}

}

/\* The following is also possible

class Layout extends React.Component {

getVal() { return "Will"; }

render() {

return (

<h1>It’s {this.getVal()}!</h1>

);

}

}

\*/

/\* Also...

class Layout extends React.Component {

constructor() {

super(); // The only thing one have to do in the constructor

this.name = "Will";

}

render() {

return (

<h1>It’s {this.name}!</h1>

);

}

}

\*/

const app = document.getElementById('app');

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

**REACT JS TUTORIAL #3 - Composing Multiple React.js Components:**

We are going to build an entire layout made completely of React components

Our client.js does not need to have “class Layout extends React Component { /\* … \*/ }” chunk. We need a folder in the js directory. The chunk lives in ./src/js/components/Layout.js now.

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

import React from "react";

import Footer from "./Footer"; // If there is “Footer.js”

import Header from "./Header";

// import ReactDOM from "react-dom"; // We do not need that

export default class Layout extends React.Component { // We export this class

render() {

return(

<div>

<Header/>

// Header is not a rendered DOM element yet, but it has the capacity to

// render out a DOM element

<Footer/> // If there is "Footer.js"

// One can create a body, and so forth

</div>

);

}

}

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

import React from "react";

import ReactDOM from "react-dom";

import Layout from "./components/Layout";

const app = document.getElementById('app');

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

/\* ./src/js/components/Header.js \*/

import React from "react";

export default class Header extends React.Component {

render() {

return (

<header>header</header>

);

}

}

We can pass unique details and information into Layout.js. (This will be mentioned later.)

More details:

/\* ./src/js/components/Header.js \*/

import React from "react";

export default class Header extends React.Component {

render() {

// An array is very useful for something like a list of li’s

var list = [

<Header/>,

<Header/>,

<Header/>,

];

<div>

{list} // JSX will know “list” is an array of three components

</div>

);

}

}

/\* ./src/js/components/Footer.js \*/

import React from "react";

export default class Header extends React.Component {

render() {

<footer>footer</footer>

);

}

}

Some people will create a “pages” (or “routes”) folder instead of “components” folder that has all of the very top-level components. In this case, those will be layouts.

If a component (in ./js/components) has a component within it, you will probably want to create a new folder (e.g. ./js/components/new\_folder)

/\* ./src/js/components/Header.js \*/

import React from "react";

import Title from "./Header/Title";

export default class Header extends React.Component {

render() {

return (

<Title/>

);

}

}

/\* ./src/js/components/Header/Title.js \*/

import React from "react";

export default class Title extends React.Component {

render() {

return (

<h1>Welcome!</h1>

);

}

}

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

<!DOCTYPE html>

<html>

<!-- ... -->

<body>

<div id="app">

<!-- Some DOM will be created here, which can be seen in Chrome’s DevTools -->

<div data-reactid=".0">

<!-- Note that there are no DOM elements created because of “Header.js” -->

<h1 data-reactid=".0.0">Welcome!</h1>

<footer data-reactid=.0.1>footer</footer>

</div>

</div>

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

</body>

</html>

**REACT JS TUTORIAL #4 - State vs Props & Application Data:**

We are going to see how data gets managed within React there are two ways—they get handled through state or props (or context, which is a little advanced and the API is not guaranteed stabilized since it is something new to React)

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

import React from "react";

import Footer from "./Footer";

import Header from "./Header";

export default class Layout extends React.Component {

constructor() {

super();

this.state = {name: "Will"};

// “state” is available through this.state, which by default, is null

// The only place you want to set state this way is in the constructor method

// You literally just set your initial state

}

render() {

return(

<div>

<Header/>

{this.state.name}

// This is similar to what is done before, but this is the proper way to do so

<Footer/>

</div>

);

}

}

Whenever the state changes on a component, the component will automatically re-render and the DOM will be updated

* React manages a virtual DOM for you, so whenever React renders out all the components, the component tree renders the layout (the header, the footer, and so forth)
* Then the virtual DOM looks for changes from the virtual DOM to the actual DOM. If there are changes, it will update only the affected nodes in the most efficient way. If there are no changes at all, it does not even touch the webpage.
* This is great because JavaScript is very fast, but DOM is very slow. Whenever we have to go to the DOM and update elements, that is the slow part.
* By having a virtual DOM where everything is updated behind the scenes and only changing the real DOM if there are differences allows our applications to become really fast

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

/\* ... \*/

export default class Layout extends React.Component {

constructor() {

super();

this.state = {name: "Will"};

}

render() {

// Let’s pretend that after one second, the user did some action and

// we want to change our state

setTimeout(() => { // Note that it is not “setTimeOut”

this.setState({name: "Bob"}); // This is the method you need to know with states

}, 1000);

return(

<div>

<Header/>

{this.state.name}

// This is similar to what is done before, but this is the proper way to do so

<Footer/>

</div>

);

}

}

To see the DOM being updated more clearly, type escape key in the Chrome’s DevTools to open the console drawer and find the “Rendering” tab. Then check the “Paint flashing” option.

* So now the screen will flash green whenever a node got changed

Even though we did a complete now render of the header, the footer, etc., the only actual DOM element that got changed was the one that contains the state’s name

* We can see that in DevTools, a span tag containing Bob is automatically made
* If you did Backbone, you will know that there is always a question of at what point I should update DOM, and when I do, who should update that. Should the list component update the whole list and re-render the whole list, or should each list item manage rendering itself.
* React re-render everything all the time on any change, but we are only going to do it in JavaScript and will touch the DOM if something is actually changed
* The mentality behind state is that state only gets used if a component has an internal value that only affects the component and does not affect any of the rest of the app

Aside from state, you may want to use props. “props” are injected into every other component.

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

/\* ... \*/

export default class Layout extends React.Component {

render() {

const title = "Welcome Will!";

// We created a prop, just like you would on an actual DOM element

return(

<div>

<Header name={"Something"} title={title}/>

// We inject a title. We can access the prop in Header.js.

<Header title={"Another title"}>

<Footer/>

</div>

);

}

}

/\* ./src/js/components/Header.js \*/

import React from "react";

import Title from "./Header/Title";

export default class Header extends React.Component {

render() {

console.log(this.props);

// Object {name: "Something", title: "Welcome Will!"}

// Object {title: "Another title"}

// The previous line is actually the printout from another version of Header

// (The printout can be seen in DevTools console)

return (

<div>

<Title/>

</div>

);

}

}

Let’s make the header use a different title

/\* ./src/js/components/Header.js \*/

/\* ... \*/

export default class Header extends React.Component {

render() {

return (

<div>

<Title title={this.props.title}/>

</div>

);

}

}

/\* ./src/js/components/Header/Title.js \*/

import React from "react";

export default class Title extends React.Component {

render() {

return (

<h1>{this.props.title}</h1>

);

}

}

Let’s change the Layout.js so that both state and props

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

/\* ... \*/

export default class Layout extends React.Component {

constructor() {

super();

this.state = {title: "Welcome"};

// By default, our title will be “Welcome”

}

render() {

setTimeout(() => {

this.setState({title: "Welcome Will!"});

}, 2000);

return(

<div>

<Header title={this.state.title}/>

<Header title={"Another title"}>

<Footer/>

</div>

);

}

}

One can see that after 2 seconds in DevTools, only the line “<h1 data-reactid=".0.0.0>Welcome Will!</h1>” after the tag “<div id="app">” changes. This is because even our header component received a new prop, React rendered out the exact same div, so it did not re-render this div to the DOM. It re-rendered a child component of title that changed, so it updated the h1 only.

* No matter how many components changes there are, we are only going to update the DOM with the actual element changes that got created when the entire tree is rendered

**REACT JS TUTORIAL #5 - JavaScript Events & Data Changes in React:**

Let’s look at actually binding (有約束力的) events interacting with user inputs

We want to add an input that is going to allow us to change the title

/\* ./src/js/components/Header.js \*/

import React from "react";

import Title from "./Header/Title";

export default class Header extends React.Component {

render() {

return (

<div>

<Title title={this.props.title}/>

<input/> // Two input fields can be seen in the webpage

</div>

);

}

}

As the header input gets typed, we want to trigger some kind of event on the Layout.js that updates the state

The title from the state will then get passed into the header and the whole rendering process will begin because the state changed. The goal is, we got to (informal, = have to) get this state to change

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

import React from "react";

import Footer from "./Footer";

import Header from "./Header";

export default class Layout extends React.Component {

constructor() {

super();

this.state = {title: "Welcome"};

// By default, our title will be “Welcome”

}

changeTitle(title) { // Our change title method

this.setState({title});

// We can do this because we use ES6

// This is the same as “this.setState({title: title});”

}

render() {

/\*

setTimeout(() => {

this.setState({title: "Welcome Will!"});

}, 2000);

\*/

return(

<div>

<Header changeTitle={this.changeTitle.bind(this)} title={this.state.title}/>

// We past the changeTitle method

// <Header title={"Another title"}> // We do not need this header anymore

<Footer/>

</div>

);

}

}

If we use “changeTitle={this.changeTitle}” but not “changeTitle={this.changeTitle.bind(this)}”, the method changeTitle will be executed in the context of whoever that is calling it

If it is “changeTitle={this.changeTitle}”…

/\* ./src/js/components/Header.js \*/

/\* ... \*/

// 2. The changeTitle method will appear to be here

// The context of changeTitle here will be different from the one in Layout.js

// To use changeTitle in Layout.js, use “changeTitle={this.changeTitle.bind(this)}”

/\*

changeTitle(title) {

this.setState({title}); // This will fail silently

}

\*/

export default class Header extends React.Component {

render() {

this.props.changeTitle(/\* ... \*/); // 1. The function is called

return (

<div>

<Title title={this.props.title}/>

<input/>

</div>

);

}

}

With “changeTitle={this.changeTitle.bind(this)}”…

/\* ./src/js/components/Header.js \*/

/\* ... \*/

export default class Header extends React.Component {

handleChange(e) {

const title = e.target.value;

this.props.changeTitle(title);

}

// handleChange will receive a JavaScript event

// It is just like a normal event in jQuery or Vanilla JS

render() {

// this.props.changeTitle("New title"); // This will work

return (

<div>

<Title title={this.props.title}/>

<input value={this.props.title} onChange={this.handleChange.bind(this)}/>

// We want the change to happen when the user types

// Again, we want to make sure that it is bound to the right context

// Other than “onChange”, we have “onClick”, “onBlur”, etc.

// By default, the input field will show “Welcome” (this.props.title)

// If onChange={...} part is missing, when one hit a key,

// there is a re-render, so the input field seems to be not responding

// Therefore, whenever you are binding the value, you also want to set up

// a change listener for that

</div>

);

}

}

React allows us to make dumb components. Header.js does not really care where the data comes from or what data it is. Title.js is even dumber. It does not aware of the data. You can use Title.js to a completely different application. The data could come from a completely different place. You get much more reusability and stability.

Our data only lives on Layout.js now. If there are any changes, we update the DOM. This makes your application development much simpler because you do not have to care about that many moving parts.

**REACT JS TUTORIAL #6 - React Router & Intro to Single Page Apps with React JS:**

Single page application means that one is able to navigate the webpage but not leaving it. JavaScript implements the navigation.

Some HTML is grabbed from the “Start Bootstrap”, a website that provides themes. jQuery is stripped out of a basic theme. A different looking CSS theme is grabbed from Bootswatch, which provides free themes for Bootstrap. (Bootstrap looks a little generic on its own.)

Bootstrap requires JavaScript to implement responsive web design. Normally, jQuery is needed for that. (Using jQuery overkills (過度地殺傷) a little.)

/\* ./package.json \*/

{

"name": "react-tutorials",

"version": "0.0.0",

"description": "",

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

"dependencies": {

"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",

"history": "^1.17.0", // New

// If the above line is missing, one can enter

// “npm install -S history@1” to get it

"react": "^0.14.6",

"react-dom": "^0.14.6",

"react-router": "^1.0.3", // New

"webpack": "^1.12.9",

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

},

"devDependencies": {},

"scripts": {

"dev": "webpack-dev-server --content-base src --inline --hot"

// The previous line is the same as

// "dev": "./node\_modules/.bin/webpack-dev-server ..."

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

},

"author": "",

"license": "ISC"

}

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

/\* This webpack.config.js works and is actually very similar to the original version of webpack.config.js appeared in tutorial 1. \*/

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 }),

],

};

To set up the thing, run “npm install” and “npm run dev”

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

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<meta name="description" content="">

<meta name="author" content="">

<title>React</title>

<!-- Bootstrap Core CSS -->

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

<!-- Custom Fonts -->

<link href="font-awesome/css/font-awesome.min.css" rel="stylesheet" type="text/css">

<link href="http://fonts.googleapis.com/css?family=Source+Sans+Pro:300,400,700,300italic,400italic,700italic" rel="stylesheet" type="text/css">

</head>

<body>

<!-- Navigation -->

<nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">

<div class="container">

<!-- Brand and toggle get grouped for better mobile display -->

<div class="navbar-header">

<button type="button" class="navbar-toggle"

data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

</div>

<!-- Collect the new links, forms, and other content for toggling -->

<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">

<ul class="nav navbar-nav">

<li>

<a href="#">Featured</a>

</li>

<li>

<a href="#">Archives</a>

</li>

<li>

<a href="#">Settings</a>

</li>

</ul>

</div>

<!-- /.navbar-collapse -->

</div>

<!-- /.container -->

</nav>

<!-- Page Content -->

<div class="container" style="margin-top:60px">

<div class="row">

<div class="col-lg-12">

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

</div>

</div>

<!-- Call to Action Well -->

<div class="row">

<div class="col-lg-12">

<div class="well text-center">

Ad spot goes here

</div>

<!-- /.col-lg-12 -->

</div>

<!-- /.row -->

<!-- Content Row -->

<div class="row">

<div class="col-md-4">

<h2>Heading 1</h2>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

<!-- /.col-md-4 -->

<div class="col-md-4">

<h2>Heading 2</h2>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

<!-- /.col-md-4 -->

<div class="col-md-4">

<h2>Heading 3</h2>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

<!-- /.col-md-4 -->

</div>

<!-- /.row -->

<!-- Footer -->

<footer>

<div class="row">

<div class="col-lg-12">

<p>Copyright &copy; KillerNews.net</p>

</div>

</div>

</footer>

</div>

<!-- /.container -->

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

</body>

</html>

We want to set up the routing so that when we click on buttons, we can actually route to the different state of the application without leaving the HTML page

When we click on the navigation bar elements, the part after the hash of the address (in the address bar) will change and this allows us to go to different states of our application

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

import Bootstrap from "./vendor/bootstrap-without-jquery";

import React from "react";

import ReactDOM from "react-dom";

import Layout from "./pages/Layout";

const app = document.getElementById('app');

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

/\* ./src/js/pages/Layout.js \*/

import React from "react";

export default class Layout extends React.Component {

render() {

return (

<h1>KillerNews.net</h1>

);

}

}

“KillerNews.net” is currently the only thing that is dynamic on the page, anything else is static HTML. We will convert the static HTML to dynamic one.

“Featured”, “Archives”, “Setting” and “Layout” are in the page folder

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

import Bootstrap from "./vendor/bootstrap-without-jquery";

import React from "react";

import ReactDOM from "react-dom";

import { Router, Route, IndexRoute, hashHistory } from "react-router";

import Archives from "./pages/Archives";

import Featured from "./pages/Featured";

import Layout from "./pages/Layout";

import Settings from "./pages/Settings";

const app = document.getElementById('app');

ReactDOM.render(

// We render our “Router” and get history booted up

<Router history={hashHistory}>

// (Actually, comments do not work here)

// Then one can render out all the routes and point them to different components

// Our basic route is going to be Layout

// Layout is going to be whatever on the entire page,

// and different components are loaded

<Route path="/" component={Layout}>

// We are going to have some sub-routes here

// The child routes will get loaded on the page depending on

// which page one is on

// For IndexRoute, the path will be the original one (“/”)

<IndexRoute component={Featured}></IndexRoute>

<Route path="archives" component={Archives}></Route>

<Route path="settings" component={Settings}></Route>

</Route>

</Router>,

app);

/\* ./src/js/pages/Featured.js \*/

import React from "react";

export default class Featured extends React.Component {

render() {

return (

<h1>Featured</h1>

);

}

}

/\* ./src/js/pages/Archives.js \*/

import React from "react";

export default class Archives extends React.Component {

render() {

return (

<h1>Archives</h1>

);

}

/\* ./src/js/pages/Settings.js \*/

import React from "react";

export default class Settings extends React.Component {

render() {

return (

<h1>Settings</h1>

);

}

}

/\* ./src/js/pages/Layout.js \*/

import { Link } from "react-router"; // So that we have the “Link” tag

import React from "react";

export default class Layout extends React.Component {

render() {

return (

<h1>KillerNews.net</h1>

{this.props.children}

// Result: There will be a text under the text “KillerNews.net”

// The displaying text depends on the path

// For example, if the address is “localhost:8080/index.html#/archives...”,

// the text will be “Archive”

<Link to="archives">archives</Link>

// Layout.js will know the component “Archives”

// because of the set up in client.js

// Result: There will be a link with text “archives”

// under the text “Archives”. After clicking on it,

// the address will be changed to “localhost:8080/index.html#/archives...”

<br><Link to="settings">settings</Link>

// To stylise the link, we have some options like

// 1. “<br><Link to="settings"><button>settings</button></Link>”

// 2. “...<button class="btn btn-success>...</button>...” (A Bootstrap button)

// 3. “<br><Link to="settings" class="btn btn-danger">settings</Link>”

// (Bootstrap allows you to do this as well)

// 4. (See below)

);

}

}

/\* 4. One can also create the button the following way

export default class Layout extends React.Component {

navigate() {

// console.log(this.props); // Check out this

this.props.history.pushState(null, "settings");

// The browser’s back button will work

this.props.history.replaceState(null, "settings");

// The browser’s back button will not work

}

render() {

return (

<div>

/\* ... \*/

<button onClick={this.navigate.bind(this)}>settings</button>

</div>

);

}

}

\*/

**REACT JS TUTORIAL #7 - React Router Params & Queries:**

Before getting into handling complicated data with React, which is related to Flux and Redux, we will cover a little more on handling route information

We want to go to a specific archive, like the one at “localhost:8080/index.html#/archives/some-article/some-filter…”

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

import Bootstrap from "./vendor/bootstrap-without-jquery";

import React from "react";

import ReactDOM from "react-dom";

import { Router, Route, IndexRoute, hashHistory } from "react-router";

import Archives from "./pages/Archives";

import Featured from "./pages/Featured";

import Layout from "./pages/Layout";

import Settings from "./pages/Settings";

const app = document.getElementById('app');

ReactDOM.render(

<Router history={hashHistory}>

<Route path="/" component={Layout}>

<IndexRoute component={Featured}></IndexRoute>

<Route path="archives/:article" component={Archives}></Route>

// We add more details (like parameters) to the route

// “localhost:8080/index.html#/archives...” will not match anything now,

// but it should be something like

// “localhost:8080/index.html#/archives/some-article...”

// unless one wraps “/:article” with parentheses

<Route path="settings" component={Settings}></Route>

</Route>

</Router>,

app);

/\* ./src/js/pages/Archives.js \*/

import React from "react";

export default class Archives extends React.Component {

render() {

console.log(this.props)

// “localhost:8080/index.html#/archives/some-article...”

// is going to render this component

// Result: The message in the console log will be

// “Object { ... params: Object ... }”

// (params: {article: "some-article", \_\_proto\_\_: Object})

const { params } = this.props; // This is called object destructuring

const { article } = params;

return (

<h1>Archives ({article})</h1>

);

}

}

Another thing you can do is that you can use query variables

* Change the address to something like “localhost:8080/index.html#/archives/some-article…?date=today?filter=none”
* Now if you look at the object printed out in the console log, one can find the “date” and “filter” inside “location.query” object

/\* ./src/js/pages/Archives.js \*/

import React from "react";

export default class Archives extends React.Component {

render() {

console.log(this.props)

const { params } = this.props; // This is called object destructuring

const { article } = params;

const { query } = this.props.location;

const { date, filter } = query;

return (

<div>

<h1>Archives ({article})</h1>

<h4>date: {date}, filter: {filter}</h4>

</div>

);

}

}

After clicking into the link, we want the element in the navigation bar to have the active effect

/\* ./src/js/pages/Layout.js \*/

import { Link } from "react-router"; // So that we have the “Link” tag

import React from "react";

export default class Layout extends React.Component {

navigate() {

this.props.history.replaceState(null, "/");

}

render() {

const { history } = this.props;

console.log(history.isActive("archives"));

// 2. Another way to tell if the link “archives” is clicked.

// If so, the console will print “true”, if not, “false” will be printed.

return (

<h1>KillerNews.net</h1>

{this.props.children}

<Link to="archives" activeClassName="test">archives</Link>

// 1. “activeClassName” is added

/\* When the link “setting” is clicked, the DOM representing the link archive

will be “<a class href="#archives" data-reactid=".0.2">archives</a>”.

If the link “archives” is clicked, the DOM will be

“<a class="test" href="#archives" data-reactid=".0.2">archives</a>”. \*/

<br><Link to="settings">settings</Link>

<br><button onClick={this.navigate.bind(this)}>featured</>

);

}

}

**REACT JS TUTORIAL #8 - React Inline Styles & Component Arrays:**

<!-- ./src/index.html (Before cleaning) -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<meta name="description" content="">

<meta name="author" content="">

<title>React</title>

<!-- Bootstrap Core CSS -->

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

<!-- Custom Fonts -->

<link href="font-awesome/css/font-awesome.min.css" rel="stylesheet" type="text/css">

<link href="http://fonts.googleapis.com/css?family=Source+Sans+Pro:300,400,700,300italic,400italic,700italic" rel="stylesheet" type="text/css">

</head>

<body>

<!-- Navigation -->

<!-- This part will be moved to ./src/js/components/layout/Nav.js -->

<nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">

<div class="container">

<!-- Brand and toggle get grouped for better mobile display -->

<div class="navbar-header">

<button type="button" class="navbar-toggle"

data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

</div>

<!-- Collect the new links, forms, and other content for toggling -->

<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">

<ul class="nav navbar-nav">

<li>

<a href="#">Featured</a>

</li>

<li>

<a href="#">Archives</a>

</li>

<li>

<a href="#">Settings</a>

</li>

</ul>

</div>

<!-- /.navbar-collapse -->

</div>

<!-- /.container -->

</nav>

<!-- Page Content -->

<!-- This part will be moved to -->

<!-- ./src/js/pages/Featured.js and ./src/js/components/Article.js -->

<div class="container" style="margin-top:60px">

<div class="row">

<div class="col-lg-12">

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

</div>

</div>

<!-- Call to Action Well -->

<div class="row">

<div class="col-lg-12">

<div class="well text-center">

Ad spot goes here

</div>

<!-- /.col-lg-12 -->

</div>

<!-- /.row -->

<!-- Content Row -->

<div class="row">

<div class="col-md-4">

<h2>Heading 1</h2>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

<!-- /.col-md-4 -->

<div class="col-md-4">

<h2>Heading 2</h2>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

<!-- /.col-md-4 -->

<div class="col-md-4">

<h2>Heading 3</h2>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

<!-- /.col-md-4 -->

</div>

<!-- /.row -->

<!-- Footer -->

<!-- This part will be moved to ./src/js/components/layout/Footer.js -->

<footer>

<div class="row">

<div class="col-lg-12">

<p>Copyright &copy; KillerNews.net</p>

</div>

</div>

</footer>

</div>

<!-- /.container -->

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

</body>

</html>

<!-- ./src/index.html (After cleaning) -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1">

<meta name="description" content="">

<meta name="author" content="">

<title>React</title>

<!-- Bootstrap Core CSS -->

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

<!-- Custom Fonts -->

<link href="font-awesome/css/font-awesome.min.css" rel="stylesheet" type="text/css">

<link href="http://fonts.googleapis.com/css?family=Source+Sans+Pro:300,400,700,300italic,400italic,700italic" rel="stylesheet" type="text/css">

</head>

<body>

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

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

</body>

</html>

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

// import Bootstrap from "./vendor/bootstrap-without-jquery";

// We get rid of “bootstrap-without-jquery”

import React from "react";

import ReactDOM from "react-dom";

import { Router, Route, IndexRoute, hashHistory } from "react-router";

import Archives from "./pages/Archives";

import Featured from "./pages/Featured";

import Layout from "./pages/Layout";

import Settings from "./pages/Settings";

const app = document.getElementById('app');

ReactDOM.render(

<Router history={hashHistory}>

<Route path="/" component={Layout}>

<IndexRoute component={Featured}></IndexRoute>

<Route path="archives/:article" component={Archives}></Route>

<Route path="settings" component={Settings}></Route>

</Route>

</Router>,

app);

In Layout.js, we render a Nav component at the top, a footer component at the bottom and a child in the middle

/\* ./src/js/pages/Layout.js \*/

import React from "react";

import { Link } from "react-router";

import Footer from "../components/layout/Footer"; // New

import Nav from "../components/layout/Nav"; // New

export default class Layout extends React.Component {

render() {

const { location } = this.props;

const containerStyle = {

marginTop: "60px"

// “"margin-top": "60px"”, or “"marginTop": "60px"” works too

};

console.log("layout");

return (

<div>

<Nav location={location}/>

<div class="container" style={containerStyle}>

// Check out the style we have here

// “<div ... style="margin-top:60px">” will not work, but we need to give

// a style object to style

// You may want to have a style sheet if there are, say, 50 rules

<div class="row">

<div class="col-lg-12">

<h1>KillerNews.net</h1>

{this.props.children}

</div>

</div>

<Footer/>

</div>

</div>

);

}

}

/\* ./src/js/components/layout/Nav.js \*/

import React from "react";

import { IndexLink, Link } from "react-router";

export default class Nav extends React.Component {

constructor() {

super()

this.state = {

collapsed: true,

};

}

toggleCollapse() {

const collapsed = !this.state.collapsed;

this.setState({collapsed});

}

render() {

const { location } = this.props;

const { collapsed } = this.state;

const featuredClass = location.pathname === "/" ? "active" : "";

const archivesClass = location.pathname.match(/^\/archives/) ? "active" : "";

// Check if the path name starts with “/archives”

const settingsClass = location.pathname.match(/^\/settings/) ? "active" : "";

const navClass = collapsed ? "collapse" : "";

return (

// Some straight up Bootstrap code

<nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">

// Q: “class” is a reserved word. Why it can be used here?

// You are supposed to use “className” instead.

// A: Recall that we loaded some plugins in webpack.config.js.

// “react-html-attrs” helps us to avoid the problem by

// changing “class” to “className” when we are transpiling

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle"

onClick={this.toggleCollapse.bind(this)} >

// Originally, in the static version of index.html, we have

// “<button type="button" class="navbar-toggle"

// data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">”

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

</div>

<div class={"navbar-collapse " + navClass}

id="bs-example-navbar-collapse-1">

// Originally version: “<div class="collapse navbar-collapse"

// id="bs-example-navbar-collapse-1">”

<ul class="nav navbar-nav">

<li class={featuredClass}>

// Note that featuredClass can be "active" or ""

// If it is “active”, the element will be shown to be pushed down

// This is required by Bootstrap

// Original version: “<li>”

<IndexLink to="/" onClick={this.toggleCollapse.bind(this)}>

Featured</IndexLink>

// Original version: “<a href="#">Featured</a>”

</li>

<li class={archivesClass}>

// (Etc.)

<Link to="archives" onClick={this.toggleCollapse.bind(this)}>

Archives</Link>

</li>

<li class={settingsClass}>

<Link to="settings" onClick={this.toggleCollapse.bind(this)}>

Settings</Link>

</li>

</ul>

</div>

</div>

</nav>

);

}

}

/\* ./src/js/pages/Featured.js \*/

import React from "react";

import Article from "../components/Article";

export default class Featured extends React.Component {

render() {

const Articles = [

"Some Article",

"Some Other Article",

"Yet Another Article",

"Still More",

"Some Article",

"Some Other Article",

"Yet Another Article",

"Still More",

"Some Article",

"Some Other Article",

"Yet Another Article",

"Still More",

].map((title, i) => <Article key={i} title={title}/> );

// Check out what map function does

/\* Compare:

Const Articles = [

<Article key={1} title={"Some Title"}>,

// If key is missing, React will have a warning which prompts the user that

// “Each child in an array or iterator should have a unique “key” prop

// “key” help React to render the page a little faster

// The value of “key” can be anything

<Article key={2} title={"Some Title"}>,

<Article key={3} title={"Some Title"}>,

];

\*/

const adText = [

"Ad spot #1",

"Ad spot #2",

"Ad spot #3",

"Ad spot #4",

"Ad spot #5",

];

const randomAd = adText[Math.round( Math.random() \* (adText.length-1) )];

console.log("featured");

return (

<div>

<div class="row">

<div class="col-lg-12">

<div class="well text-center">

{randomAd}

// We have a randomAd text

</div>

</div>

</div>

<div class="row">{Articles}</div>

// We print out a bunch of articles

// Bootstrap helps us with the arrangement of the articles

</div>

);

}

}

/\* ./src/js/components/Article.js \*/

import React from "react";

export default class Article extends React.Component {

render() {

const { title } = this.props;

return (

<div class="col-md-4">

<h4>{title}</h4>

// The only thing that is dynamic

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe rem nisi accusamus error velit animi non ipsa placeat. Recusandae, suscipit, soluta quibusdam accusamus a veniam quaerat eveniet eligendi dolor consectetur.</p>

<a class="btn btn-default" href="#">More Info</a>

</div>

);

}

}

/\* ./src/js/pages/Archives.js \*/

import React from "react";

import Article from "../components/Article";

export default class Archives extends React.Component {

render() {

const { query } = this.props.location;

const { params } = this.props;

const { article } = params;

const { date, filter } = query;

const Articles = [

"Some Article",

"Some Other Article",

"Yet Another Article",

"Still More",

"Fake Article",

"Partial Article",

"American Article",

"Mexican Article",

].map((title, i) => <Article key={i} title={title}/> );

return (

<div>

<h1>Archives</h1>

article: {article}, date: {date}, filter: {filter}

<div class="row">{Articles}</div>

</div>

);

}

}

/\* ./src/js/pages/Settings.js \*/

import React from "react";

export default class Settings extends React.Component {

render() {

console.log("settings");

return (

<h1>Settings</h1>

);

}

}

/\* ./src/js/components/layout/Footer.js \*/

import React from "react";

export default class Footer extends React.Component {

render() {

return (

<footer>

<div class="row">

<div class="col-lg-12">

<p>Copyright &copy; KillerNews.net</p>

</div>

</div>

</footer>

);

}

}

**REACT FLUX TUTORIAL #9 - React Flux Introduction & Flux Stores:**

React, by default, does not give you any specific way to manage all the data for your application. It just has the ability to receive props. You can inject props into a component and the component handles that, but the React library does not tell you how to set where your data store. This is why it is called just a view layer React.

There is no actual framework when compared to Angular or Backbone. You get to create your own or use another framework. That is where Flux comes in. Flux is a pattern that Facebook has laid out (設計) for building React framework. There are also other Flux-based frameworks out there.

(React) component >> Actions >> Dispatcher (收發) >> Stores >> (Components)

* The component does two things—they fire off (to shoot a weapon) actions and they listen to (or interacts with) stores
* For example, when we take an action of adding a new to-do to a to-do list, the stores later update
* Actions are only aware of the dispatcher
* In Backbone, a store is a collection and there is no concept of “store” in Angular at this point
* The dispatcher is basically like a PubSub (a pattern of communicating events). There is one major difference: in PubSub, every module can subscribe to a specific event or a specific set of events and get notified when the event takes place, whereas a Flux dispatcher takes every event that comes through it and sends it to every subscriber.
  + If there are four stores registered to the dispatcher, those four stores get every event. The stores can choose to react to it or not.
  + The reason that this is a good idea is that React gives everybody the chance to change, re-render when there are any changes to our application

Actions <> Constants <> Stores

* Some frameworks use constants, some frameworks do not
* It is a way of storing action names

The appearance of actions:

* Dispatcher.dispatch({ type: "CREATE\_TODO", title: "Some Title" });
  + The dispatcher does not care about what the action is. It is just going to pass it to everybody registered, so some people use “type”, and some people use “actionType”.
  + “type” is what the action called, and created to do
  + An action can have some additional data. In this case, the additional data is “title”.
* Dispatcher.dispatch({ type: "DELETE\_TODO ", id: 4124 });

Let’s go ahead and actually make our basic to-do list application and convert it to Flux

Our to-do list has the ability to set favourite things and to change settings

We will work with two different components first:

1. The entire to-do list
2. A component (a to-do item)

/\* ./package.json \*/

{

"name": "react-tutorials",

"version": "0.0.0",

"description": "",

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

"dependencies": {

"babel-core": "^6.18.2", // "^6.17.0" -> "^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",

"flux": "^2.1.1", // New

"history": "^1.17.0",

"react": "^0.14.6",

"react-dom": "^0.14.6",

"react-router": "^1.0.3",

"webpack": "^1.12.9",

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

},

"devDependencies": {},

"scripts": {

"dev": "webpack-dev-server --content-base src --inline --hot"

},

"author": "",

"license": "ISC"

}

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

/\* ... (The file is the same as before) \*/

/\* ./src/index.html \*/

/\* ... \*/

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

/\* ... (The file contains the path information) \*/

/\* ./src/js/components/layout/Nav.js \*/

/\* ... \*/

/\* ./src/js/pages/Layout.js \*/

/\* ... \*/

/\* ./src/js/pages/Todos.js \*/

import React from "react";

import Todo from "../components/Todo";

export default class Todos extends React.Component {

constructor() {

super();

this.state = {

todos: [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Bills",

complete: false

},

],

};

}

render() {

const { todos } = this.state;

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

// We use ID as the key. This is a better key than the index in the array.

});

return (

<div>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

/\* ./src/js/components/Todo.js \*/

import React from "react";

export default class Todo extends React.Component {

constructor(props) {

super();

}

render() {

const { complete, edit, text } = this.props;

const icon = complete ? "\u2714" : "\u2716"

if (edit) {

return (

<li>

<input value={text} focus="focused"/>

</li>

);

}

return (

<li>

<span>{text}</span>

<span>{icon}</span>

</li>

);

}

}

/\* ./src/js/pages/Favourites.js \*/

/\* ... \*/

/\* ./src/js/pages/Settings.js \*/

/\* ... \*/

/\* ./src/js/components/layout/Footer.js \*/

/\* ... \*/

We want to make the thing dynamic. The data (to-do items) should appear in the store.

We want to be able to listen to the changes in the store.

/\* ./src/js/stores/TodoStore.js \*/

import { EventEmitter } from "events";

// We can thus listen to the changes in the store

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Water Bill",

complete: false

},

];

}

getAll() {

return this.todos;

}

}

const todoStore = new TodoStore;

// We can then do something like todoStore.on("change", someHandler)

export default todoStore;

// Whenever you import todoStore,

// you are going to get the created instance of todoStore

/\* ./src/js/pages/Todos.js \*/

import React from "react";

import Todo from "../components/Todo";

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.state = {

todos: TodoStore.getAll()

};

}

render() { /\* ... \*/ }

}

**REACT FLUX TUTORIAL #10 - Flux Store Events:**

Let’s get the application to be more dynamic so that when our store changes, it emits a change method and then the components can automatically be updated

/\* ./src/js/stores/TodoStore.js \*/

import { EventEmitter } from "events";

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Water Bill",

complete: false

},

];

}

createTodo(text) { // A newly added method

const id = Date.now(); // Date.now() returns the time stamp

this.todos.push({

id,

text,

complete: false,

})

this.emit("change");

// We fire off a change event

}

getAll() {

return this.todos;

}

}

const todoStore = new TodoStore;

window.todoStore = todoStore

// We expose the todoStore globally in order to test only

export default todoStore;

/\* ./src/js/components/Todo.js \*/

import React from "react";

import Todo from "../components/Todo";

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.state = {

todos: TodoStore.getAll(),

};

}

componentWillMount() {

// Whenever the component is about to render to the DOM for the first time,

// this function will be called

// Here is a great place to add event listeners

// because we only have to do this once

TodoStore.on("change", () => {

// Arrow function binds "this" automatically

this.setState({

todos: TodoStore.getAll()

});

})

}

render() {

const { todos } = this.state;

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

});

return (

<div>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

To test the application, enter a command like “todoStore.createTodo("Smile")” in Chrome’s DevTools’ console. An extra item should be added to the to-do list.

**REACT FLUX TUTORIAL #11 - The Flux Dispatcher:**

Currently, our components pull the initial information from the TodoStore.js and then we listen to the TodoStore.js and update whenever it changes. Let’s register our store to a Flux dispatcher.

/\* ./src/js/dispatcher.js \*/

import { Dispatcher } from "flux";

export default new Dispatcher;

There are two dispatcher methods we will use: dispatcher.register() and dispatcher.dispatch(). You use the former one to register a new listener and the latter one to dispatch actions.

/\* ./src/js/stores/TodoStore.js \*/

import { EventEmitter } from "events";

import dispatcher from "../dispatcher"; // New

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Water Bill",

complete: false

},

];

}

createTodo(text) {

const id = Date.now();

this.todos.push({

id,

text,

complete: false,

})

this.emit("change");

}

getAll() {

return this.todos;

}

handleActions(action) { // A new method

console.log("TodoStore received an action", action) // For testing only

switch(action.type) {

case "CREATE\_TODO": {

this.createTodo(action.text);

}

}

}

}

const todoStore = new TodoStore;

dispatcher.register(todoStore.handleActions.bind(todoStore));

window.dispatcher = dispatcher; // For testing only

// Enter command like dispatcher.dispatch({type: "some event"})

// in Chrome's DevTools console

// Result: "TodoStore received an action {type: "some event"}"

// Enter command like dispatcher.dispatch({type: "CREATE\_TODO", text: "Have Lunch"})

// The printout will be

// "TodoStore received an action {type: "CREATE\_TODO", text: "Have Lunch"}",

// and a new to-do can be seen in the list

export default todoStore;

We finished “dispatcher” and “stores”, and we only need “actions” in place now

**REACT FLUX TUTORIAL #12 - Flux Actions:**

(React) component >> Actions >> Dispatcher >> Stores >> (Components); Actions <> Constants <> Stores

We have this decoupled framework and the application re-renders all the time. Flux apps are very stable.

As the application grows in complexity up to hundreds and thousands of different types of actions, our stores only listen to the actions they care about. Our component only listens to the stores they care about.

We will now work on the “action” part:

/\* ./action/TodoActions.js \*/

import dispatch from "../dispatcher";

export function createTodo(text) {

dispatcher.dispatch({

type: "CREATE\_TODO",

text,

})

}

export function createTodo(id) {

dispatcher.dispatch({

type: "DELETE\_TODO",

id,

})

}

/\* ./src/js/pages/Todos.js \*/

import React from "react";

import Todo from "../components/Todo";

import \* as TodoActions from "../actions/TodoActions";

// We import every object from the file

// TodoActions will be an object with the functions in it

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.state = {

todos: TodoStore.getAll(),

};

}

componentWillMount() {

TodoStore.on("change", () => {

this.setState({

todos: TodoStore.getAll()

});

})

}

createTodo() {

TodoActions.createTodo(Date.now());

// Every time one hit the button,

// a to-do item with timestamp as its value will be created

}

render() {

const { todos } = this.state;

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

});

return (

// A button is added

<div>

<button onClick={this.createTodo.bind(this)}>Create!</button>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

**REACT FLUX TUTORIAL #13 - Asynchronous & AJAX Flux Actions:**

One of the first question one will get when people are learning Flux is that Flux seems to be great for synchronous actions that happen inside of the application, how one should handle asynchronous actions.

* For instance, we want to load data from another page and then create an action when it is done

/\* ./src/js/pages/Todos.js \*/

import React from "react";

import Todo from "../components/Todo";

import \* as TodoActions from "../actions/TodoActions";

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.state = {

todos: TodoStore.getAll(),

};

}

componentWillMount() {

TodoStore.on("change", () => {

this.setState({

todos: TodoStore.getAll()

});

})

}

reloadTodos() { // Originally, this is "createTodo"

TodoActions.reloadTodos();

}

render() {

const { todos } = this.state;

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

});

return (

// The button is changed

<div>

<button onClick={this.reloadTodos.bind(this)}>Reload!</button>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}

/\* ./action/TodoActions.js \*/

import dispatcher from "../dispatcher";

export function createTodo(text) {

dispatcher.dispatch({

type: "CREATE\_TODO",

text,

});

}

export function deleteTodo(id) {

dispatcher.dispatch({

type: "DELETE\_TODO",

id,

});

}

export function reloadTodos() {

// This will call something like Axios, jQuery, AJAX

// axios("http://someurl.com/somedataendpoint").then((data) => {

// console.log("Got the data!", data);

// }

dispatcher.dispatch({type: "FETCH\_TODOS"}); // We fake the fetching process

setTimeout(() => {

// We dispatch a second event

dispatcher.dispatch({type: "RECEIVE\_TODOS", todos: [

{

id: 421254297,

text: "Go Shopping Again",

complete: false

},

{

id: 423552694,

text: "Hug Wife",

complete: true

}

]});

if (false) {

// If something goes wrong

dispatcher.dispatch({type: "FETCH\_TODOS\_ERROR"});

}

}, 1000);

// Maybe there is a loader that cares about this event

// After this event, the "loading" property (if there is such property)

// in the state of the class Todos may be changed to true

}

/\* ./src/js/stores/TodoStore.js \*/

import { EventEmitter } from "events";

import dispatcher from "../dispatcher";

class TodoStore extends EventEmitter {

constructor() {

super()

this.todos = [

{

id: 113464613,

text: "Go Shopping",

complete: false

},

{

id: 235684679,

text: "Pay Water Bill",

complete: false

},

];

}

createTodo(text) {

const id = Date.now();

this.todos.push({

id,

text,

complete: false,

})

this.emit("change");

}

getAll() {

return this.todos;

}

handleActions(action) {

switch(action.type) {

case "CREATE\_TODO": {

this.createTodo(action.text);

}

case "RECEIVE\_TODOS": {

// Our new type of event

this.todos = action.todos;

this.emit("change");

}

}

}

}

const todoStore = new TodoStore;

dispatcher.register(todoStore.handleActions.bind(todoStore));

export default todoStore;

**REACT FLUX TUTORIAL #14 - React & Flux Memory Leaks:**

A memory leak happens when you fail to unbind events in JavaScript, JavaScript keeps your objects in memory and it keeps building up. This may crash one’s browser. As browsers consume memory on one’s computer, the entire computer system could run poorly.

Try the following:

1. Hit “Reload!” in the Todos page
2. Go to other pages
3. Go back to Todos page and hit “Reload!” again, an error message “setState(...): Can only update a mounted or mounting component. This usually means you called setState() on an unmounted component. This is a no-op. Please check the code for the undefined component.” shows up.

/\* ./src/js/pages/Todos.js \*/

import React from "react";

import Todo from "../components/Todo";

import \* as TodoActions from "../actions/TodoActions";

import TodoStore from "../stores/TodoStore";

export default class Todos extends React.Component {

constructor() {

super();

this.getTodos = this.getTodos.bind(this); // To type less

this.state = {

todos: TodoStore.getAll(),

};

}

componentWillMount() {

// When componentWillUnmount() does not exist

// The following is the offending code

// Every time we change our route and change back,

// a new Todos component get created

// There may be two components (the new and the original Todos component)

// listen to the change, but the original one is not connected to the DOM tree

TodoStore.on("change", this.getTodos);

// console.log("Count:", TodoStore.listenerCount("change"))

// This shows that every time we change our route and change back,

// the count of listener increases by one

}

componentWillUnmount() {

TodoStore.removeListener("change", this.getTodos);

}

getTodos() {

this.setState({

todos: TodoStore.getAll()

});

}

reloadTodos() {

TodoActions.reloadTodos();

}

render() {

const { todos } = this.state;

const TodoComponents = todos.map((todo) => {

return <Todo key={todo.id} {...todo}/>;

});

return (

<div>

<button onClick={this.reloadTodos.bind(this)}>Reload!</button>

<h1>Todos</h1>

<ul>{TodoComponents}</ul>

</div>

);

}

}