ReactJS

# NodeJS

## What is NodeJS?

Makers of NodeJS took JavaScript that runs in browser and made it run in

# Webpack

## Setup

This is the preferred module loader for ReactJS.

Initially we had require.js and later came in browserify and Webpack. Now both Browserify and Webpack are competing.

For doing React Webpack is recommended because it has more features and for react we need those features.

For this we need NodeJS. So install Nodejs.

And later we can do to have package.json.

* npm init

We can install Webpack using npm.

* npm install –S webpack

To run Webpack command, we can run

* npm install –g webpack

Now we need Webpack.config.json file.

* npm install touch-cli -g
* touch webpack.config.js

Then copy paste Webpack.config.js from <https://gist.githubusercontent.com/learncodeacademy/25092d8f1daf5e4a6fd3/raw/7f42be48f6474eec5d6e51e164229197ab0baa42/webpack.config.js>

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

var webpack = require('webpack');

module.exports = {

context: \_\_dirname,

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

entry: "./js/scripts.js",

output: {

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

filename: "scripts.min.js"

},

plugins: debug ? [] : [

new webpack.optimize.DedupePlugin(),

new webpack.optimize.OccurenceOrderPlugin(),

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

],

};

Now we can run Webpack

* Webpack

Then we will get scripts.min.js that is not minified.

### Module1.js

console.log("module1");

### Module2.js

console.log("module2");

### Script.js

//this code won't run in browser this is nodeJs or commonJS codes

require('./module1.js');

require('./module2.js');

To minify we can run this

* SET NODE\_ENV=production
* Webpack

# Introduction to ReactJS

## Workspace setup

Check out the link for code

<https://github.com/learncodeacademy/react-js-tutorials>

Installed some babel stuff for transpiling

{

"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",**

"react": "^0.14.6",

"react-dom": "^0.14.6",

"webpack": "^1.12.9",

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

},

"devDependencies": {},

"scripts": {

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

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

},

"author": "",

"license": "ISC"

}

This will transpile both our JSX and ES6 code at same time such that it will support at least till IE8.

In Webpack.config.js we have added loaders

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" : false,

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

module: {

loaders: [

{

**//01. anything that is .js file goes through this babel loader**

test: /\.js?$/,

**//02. Excludes these modules**

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 }),

],

};

We can run it with these steps

* npm install
* Webpack –watch

In index.html we are having client.min.js that is obtained after the bundling of js files is done by the Webpack.

<!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>

We have a div with id “app” where our entire app will be rendered.

Now we can do

* npm install
* webpack -w/ --watch

Let’s look at client.js.

We are importing React and ReactDOM in this component. ReactDOM is a rendering engine.

First thing we are doing here is creating a react component called Layout by extending React.Component.

import React from "react";

import ReactDOM from "react-dom";

class Layout extends React.Component {

render() {

return (

<h1>It works!</h1>

);

}

}

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

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

The core of react is that everything is a component. In html file everything is an element, where as in React Component we are having a render method that returns HTML stuff i.e. the JSX part of it. And anything here that is that is wrapped in parenthesis is converted into JS to create respective HTML elements.

So in JS it is transpiled into something like this.

Var h1 = Document.createElement(“h1”);

div.innerHTML = “It works!”;

Here we are getting div element with id “app” where we are supposed to render our component and then adding our Layout component to it.

For live reload we have Webpack-dev-server.

We can run this by globally installing it

* npm install -g webpack-dev-server

and run it in cmd(src: folder with content)

* webpack-dev-server --content-base src

and use it with the URL: <http://localhost:8080/webpack-dev-server/> or even <http://localhost:8080> (without )where index.html is the root.

So here any change in the component initiates transpiling and bundling of the files to obtain new client.min.js, but to get updated webpage we have to refresh in browser. So we can add “—inline --hot”, this will add live reload in the browser and the top frame is removed.

Else we can just have this in package.json in scripts.

"scripts": {

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

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

},

Here we can just do

* npm run dev

# ReactJS Components & Rendering

Everything in React is a component and returns exactly one DOM element.

import React from "react";

import ReactDOM from "react-dom";

class Layout extends React.Component {

render() {

return (

**<h1>**It works!**</h1>**

);

}

}

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

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

We can import dynamic information very easily using ReactJS. We can have a constant “name” and we can display it dynamically using {}.

import React from "react";

import ReactDOM from "react-dom";

class Layout extends React.Component {

render() {

**const name = “Neha”;**

return (

<h1>It **{name}**!</h1>

);

}

}

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

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

Anything in curly braces in JSX portion of the file will just execute as normal JavaScript. So in this case it will evaluate name.

We can even do mathematical function or self-executing anonymous function.

Example

* {3+5-4}
* {(function () {return 3;}) ()}

Instead of having these constants defined in render function and making it heavy we can define stuff outside render and access is with the help of “this” keyword.

import React from "react";

import ReactDOM from "react-dom";

class Layout extends React.Component {

**getName () {**

**return “Neha”;**

**}**

render () {

return (

<h1>It **{this.getName()}**! </h1>

);

}

}

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

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

We can even use constructor method in JavaScript class. In this constructor we have to call the super class’s constructor i.e. React.Component’s constructor on the first line. In constructor we can initialize this.name and can be access from the render function.

import React from "react";

import ReactDOM from "react-dom";

class Layout extends React.Component {

**constructor(){**

**super();**

**this.name = “Neha”;**

**}**

render () {

return (

<h1>It **{this.name}**! </h1>

);

}

}

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

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

# Composing Multiple React.js Components

Our client.js shouldn’t have Layout component, we need to have it in a separate component. So we can cut the Layout component from client.js and paste it in a separate file named “Layout.js” in some separate folder components. All we have to do is import react and export Layout class. So we can import this client.js and everything will work same as before.

**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);

**components/Layout.js**

**import React from "react";**

**export** **default** class Layout extends React.Component {

constructor(){

super();

this.name = “Neha”;

}

render () {

return (

<h1>It {this.name}! </h1>

);

}

}

So we can go forward and create rest of the components as we got to know how to create and use multiple components.

Let’s create header component.

**components/Header.js**

**import React from "react";**

**export** **default** class Header extends React.Component {

render () {

return (

<header>header </header>

);

}

}

Now let’s render this component in Layout.js. We can just import Header in the Layout component and return one div with HTML component inside it. We can use it straight in the HTML as components can be used anywhere in HTML.

import React from "react";

**import Header from "./Header";**

export default class Layout extends React.Component {

render () {

return (

**<div>**

**<Header />**

**</div>**

);

}

}

We can call this header multiple times in Layout. Instead of manually having many headers typed in return we can also have an array of DOM elements. Inside return we can just give the array in curly braces and react will automatically evaluate the list and display the array of headers.

import React from "react";

import Header from "./Header";

export default class Layout extends React.Component {

render () {

**var list = [**

**<Header />**

**<Header />**

**<Header />**

**];**

return (

<div>

**{list}**

</div>

);

}

}

Now we can also have a footer.

**components/Footer.js**

import React from "react";

export default class Footer extends React.Component {

render () {

return (

<footer>footer</footer>

);

}

}

And now we can have footer in layout as well.

import React from "react";

import Footer from "./Footer";

import Header from "./Header";

export default class Layout extends React.Component {

render () {

return (

<div>

<Header />

<Footer />

</div>

);

}

}