Table of Contents

[**part 1 – Verify nodejs** 2](#_Toc49518772)

[**part 2 – Building a Simple nodejs API** 3](#_Toc49518773)

[**part 3 – Including and Working with node Packages** 5](#_Toc49518774)

[**part 4 – Create Simple Routing** 6](#_Toc49518775)

[**part 5 – Data Layer (Simulated)** 8](#_Toc49518776)

[**part 6 – Nodemon and Scripts** 10](#_Toc49518777)

[**part 7 – Webpack** 11](#_Toc49518778)

[**part 8 – Babel** 13](#_Toc49518779)

[**part 9 – React** 14](#_Toc49518780)

[**part 10 – Static Pages** 17](#_Toc49518781)

[**part 11 – Styling** 18](#_Toc49518782)

[**part 12 – React Components** 19](#_Toc49518783)

[**APIendix A – Simple HTTP Server** 22](#_Toc49518784)

[**APIendix B – Permission Denied Errrors** 22](#_Toc49518785)

API NodeJS Express Webpack and Babel

Today you will get an ideas of how to build a basic NodeJS server, adding packages to that server then build a quick ReactJS app. We get a feel for the two essential packages that are needed for ReactJS development, Webpack and Babel. Along the way, we add some basic data interaction, organize our files into folders and add styling to our application.

## part 1 – Verify nodejs

1. Create a folder called **API** (or any name you wish)
2. Open a terminal inside of **API** and run the command **npm init**
3. Follow the prompts and just hit **enter** for each question, this is just to create a package.json file (or you can just type in **npm init -y**)

|  |
| --- |
|  |

1. According to the .json file, node will look for index.js in order to execute the code inside, so use the >**touch** command to create index.js inside of the **API** folder.
2. Add the following code in your index.js file. This is just to make sure that node is working and it is executing properly.

|  |
| --- |
| **console.log("Hello from Skillsoft!");** |

1. Execute index.js by typing in the command >**node index** from the command prompt. It should show “Hello from Skillsoft”. This step confirms that we can move on to other parts.

|  |
| --- |
|  |

**------end of part 01-------**

## part 2 – Building a Simple nodejs API

1. Open index.js inside of a text editor and type the following lines, replacing the line we had before:

|  |
| --- |
| **const http = require('http');**  **const hostname = "localhost";**  **const port = 8000;** |

This code means that we are using the http module of **nodejs**, and we will define the other two parameters that the **http service** requires, *hostname* and *port*.

1. Next we will define a variable to point to the **createServer()** method which will hold a reference to the server

|  |
| --- |
| **const SkillServer = http.createServer();** |

|  |
| --- |
| **A special note on the http.createServer() method**.  The createServer() method returns a web server object, which will listen for requests and then handle those requests by returning responses to the client, which could be a browser.  createServer() takes a function that is called each time a request is made.  Once a request is made and that request gets to the server, it is considered a request object and it is based on an HTTP method or verb. The headers object also exist on that request, but it is a separate object.  There are some requests that need special handling, such as POST and PUT. These need special handlers that can work with the ReadableStream interface. When the incoming data hAPIens to be string, then it is possible to handle this string data as an array.  The response object on the other hand is an instance of the ServerResponse class. It is a WritableStream. To send back a response to the client means dealing with the stream methods such as write() and end(). |

1. The **createServer()** method takes a function that handles both the **request** and **response** objects. Extend the method to include that function as an anonymous function within the parenthesis of the **createServer()** method.

|  |
| --- |
| **const myServer = http.createServer(function(request, response){**    **});** |

1. This function now gives us access to these two objects, **request** and **response**, so we can interrogate the **request** object for things like **parameter** values or HTML form values and we can use the **response** object to send data back to the client. In this case we will only use the **response** object to send a **header** message as well as some text to the client

|  |
| --- |
| **const myServer = http.createServer(function(request, response){**  **response.writeHead(200, {'Content-Type':'text/plain'});**  **response.write("Hello from Skillsoft");**  **response.end();**  **});** |

1. Finally we can call the **listen()** method and pass it the **port** and **hostname**

|  |
| --- |
| **myServer.listen(port, hostname);** |

Here is the entire index.js file

|  |
| --- |
| **const http = require('http');**  **const hostname = "localhost";**  **const port = 8000;**  **const myServer = http.createServer(function(request, response){**  **response.writeHead(200, {'Content-Type':'text/plain'});**  **response.write("Hello from Skillsoft");**  **response.end();**  **});**  **myServer.listen(port, hostname);** |

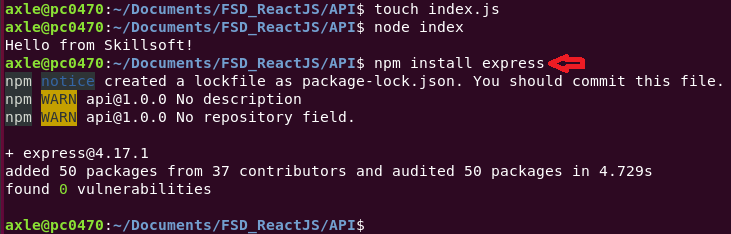
1. Run this APIlication by typing in node **index.js**. In a browser navigate to <http://localhost:8000> and you should see the message from the **response.write()** method call. You won’t get any feedback in the terminal, but if you want to stop the server, just use your keyboard CTRL-C

**------end of part 02-------**

## part 3 – Including and Working with node Packages

1. Install the **express** package by running this command from a terminal window that is pointing to your directory: **npm install express**

**Express** is a thin layer of fundamental web APIlication features, it is considered a web framework that uses NodeJS in the background.



1. Open the index.js and replace the existing code with these lines

|  |
| --- |
| **const express= require('express');**  **const port = 8000;** |

1. Create a new variable and point it to the constructor of **express**

|  |
| --- |
| **const express= require('express'); const port = 8000; const api = express();** |

1. At this point we can use the the **API** object to create a route to handle a get, post, put requests. Note, the ‘/’ indicate the root of our site.

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const api = express(); api.get('/', function(request, response){});** |

1. With this code in place, we can use it to make typical get requests and wait for a response.

|  |
| --- |
| **api.get('/', function(request, response){**  **response.send('you are on the root route');**  **});** |

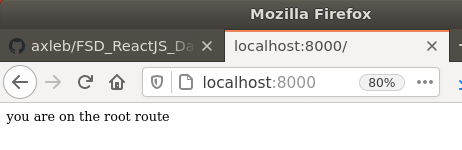
1. Finally for this file, lets add the **listen()** method to use **API** and also to inform the developer that the service has started. This code goes to the bottom of the server file.

|  |
| --- |
| **api.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

Run the APIlication by typing in the console window: **node index**

1. Open any browser and go to the address in our app, so usually it is this:  
   <http://localhost:8000/>

Run the APIlication by typing in the console window: **node index**



To stop the server type **CTRL-c**

## part 4 – Create Simple Routing

1. At this point we can use the the **api** object again to create a route to handle a POST request.

|  |
| --- |
| **api.get('/', function(request, response){**  **response.send('you are on the root route');**  **});**  **api.post('/addnewemployee', function(request, response){});**  **//**  **api.listen(port, function(){** |

1. With this code in place, we can use it to now get values from a form. For example on the form there is a field called **empName**. We can get the value that the user put into that field by interrogating the **body** property of the **request** object.

|  |
| --- |
| **api.post('/addnewemployee, function(request, response){**  **let empName = request.body.empName;**    **});** |

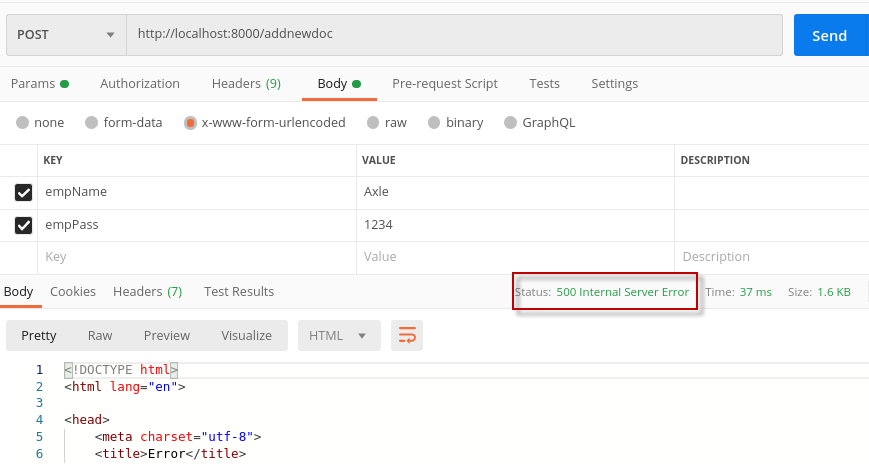
1. We can extend this to the **weight** value as well. Also for now lets just use the **log** to show that we did receive those values on the server end

|  |
| --- |
| **api.post('/addnewemployee, function(request, response){**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **console.log(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **response.end(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **});** |

1. This is the entire file so far

|  |
| --- |
| **const express= require('express');**  **const port = 8000;**  **const api = express();**  **//**  **api.get('/', function(request, response){**  **response.send('you are on the root route');**  **});**  **api.post('/addnewemployee', function(request, response){**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **console.log(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **response.end(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **});**  **//**  **api.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

1. Now we have to test this out using a REST client, so first run the API using >**node index** then use your REST client to hit the endpoint. Remember to pass along a header, **x-www-form-urlencoded**



This should throw an error in Postman and in our terminal window.

1. The reason is that we did not add code to handle **json** data and form handling via a POST request. First add this line in the index.js file and test again.

|  |
| --- |
| **const express= require('express');**  **const port = 8000;**  **const api = express();**  **api.use(express.json());**  **api.get('/', function(request, response){**  **response.send('you are on the root route');** |

**Remember to restart the server each time**

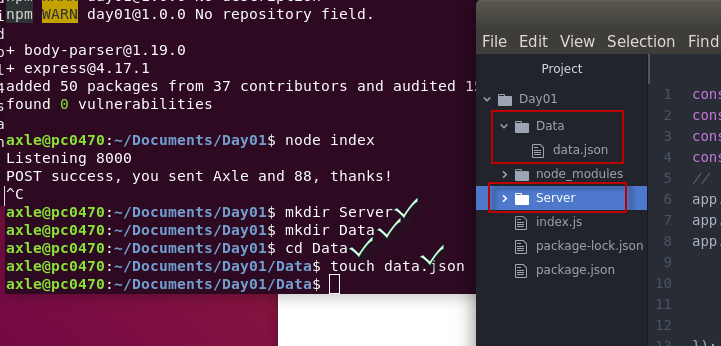
1. Although we did not get an error both fields are *undefined*, so add in the ability to parse form data with this line

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const api = express();**  **api.use(express.json());**  **api.use(express.urlencoded({extended:false}));**  **//**  **api.get('/', function(request, response){** |

Alternatively you can do this:  
**api.use(express.json(), express.urlencoded({extended:false}));**

**-----end of part 04-------**

## part 5 – Data Layer (Simulated)

1. Create a folder called **Server** and put the index.js file in there, then create a folder called **Data** and **>touch** a file called data.json to create this file inside of the data folder.  
     
   
2. Inside of the data.json file, add the following json structure:

|  |
| --- |
| **{**  **"empName": "James",**  **"empPass": "James"**  **}** |

1. Back in index.js we will require this data.json file, and represent it in a variable called **data**

|  |
| --- |
| **const express= require('express');**  **const port = 8000;**  **const api = express();**  **const data = require('./Data/data.json');**  **api.use(express.json());** |

1. Run the API then test, make sure you navigate using the terminal window into the Server folder. Test the routes by going to <http://localhost:8000/> in the browser. Remember to stop the service, type **CTRL-c** in the console window.  
   At this point, the command node index wont work you would have to do:  
   **node Server/index**

Notice that *casing* maters with Linux systems.  
also, if you move index.js into the Server folder, the Data folder will be unreachable, you would need to change the line that requires the data.json file to this: **const data = require('./../Data/data.json');**

1. We now create a different route to serve our data. For now you will need to stop and then start the API again to see the changes.

|  |
| --- |
| **api.use(bodyParser.json())**  **//**  **api.get('/getemployees', function(request, response){**  **response.send(data);**  **})**  **api.get('/', function(request, response){**  **response.send("Hello from Express");**  **})**  **api.post('/addnewemployee, function(request, response){** |

Test it by going to <http://localhost:8000/getemployees> in the browser

Here is the entire index.js file so far:

|  |
| --- |
| **const express= require('express');**  **const port = 8000;**  **const api = express();**  **const data = require('./../Data/data.json');**  **api.use(express.json());**  **api.use(express.urlencoded({extended:false}));**  **//**  **api.get('/getemployees', function(request, response){**  **response.send(data);**  **})**  **api.get('/', function(request, response){**  **response.send('you are on the root route');**  **});**  **api.post('/addnewemployee', function(request, response){**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **console.log(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **response.end(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **});**  **//**  **api.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

**------end of part 05-------**

## part 6 – Nodemon and Scripts

So far each time we need to make a change to our server or any file, we had to stop the server using CTRL-C make the change and then start the server again. Well there is a package called **Nodemon**, that will watch our files for changes and restart the server automatically.

1. First stop the server, then install **Nodemon** globally, then each time you use it for development, install it locally as a develepment dependency. To install globally, you may need to be the admin, so in my case it is**: >sudo npm install -g nodemon**. Then in the current case I would type >**npm install nodemon --save-dev** from within the current directory/folder, so from within API in my case.
2. Now use **Nodemon** to start the same file (index), then make a change to the file, then watch the console window, **Nodemon** restarts the server. You can stop **Nodemon** in the usual way, with **CTRL-c**.
3. Normally though, we start **Nodemon** as a script, so in the package.json file, add this line

|  |
| --- |
| **"main": "index.js",**  **"scripts": {**  **"test": "echo \"Error: no test specified\" && exit 1",**  **"start": "nodemon Server/index"**  **},**  **"keywords": [],** |

**Notice the comma after the last script line**. Now we can test this out by typing in from the command line (you must be in the root folder): >**npm start**

1. Optional. You can make a change just to see how Nodemon works. So you can go into index.js change the return statement from the root path and see that Nodemon automaticall restarts your API – once you save it.

**------end of part 06-------**

## part 7 – Webpack

Webpack is considered a bundler. WP will bundle two or more javascript files, and all its dependencies into a single javascript file that can be referenced from a HTML page. It does more than this but this is a start.

1. Install **Webpack** locally like you did for **Nodemon**, so >**npm install webpack webpack-cli --save-dev.** Since we are installing a package, you will need to stop the service manually, so back to CTRL-C
2. Open the package.json file in an editor and add a new script:

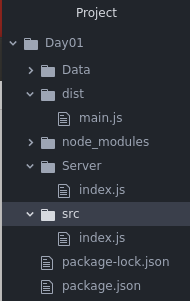
|  |
| --- |
| **"scripts": {**  **"test": "echo \"Error: no test specified\" && exit 1",**  **"start": "nodemon Server/index",**  **"build":"webpack --mode development"**  **},**  **"keywords": [],**  **"author": "",** |

Mode will be production when you deploy.

1. **Webpack** does depend on a file called index.js being in a folder called **src**, so let’s create this before we test. Use either **Atom** or the file system to create a folder called **src** in the **root** of your project, then create an empty index.js file inside of that folder, for now. Note, we now have to index.js files, but in different folders.

Now run >**npm run build**

|  |
| --- |
|  |

1. After running the command above, check your file structure, WP would have created a new folder called **dist** and inside of that there will be a main.js file with quite a bit of code.
2. (optional) If you want you can change the mode in package.json from *production* to *development* and check the main.js file again.

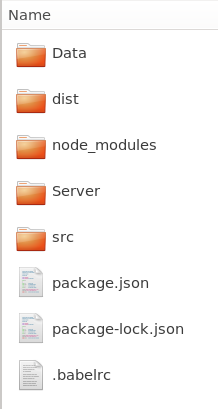
**------end of part 07-------**

## part 8 – Babel

Babel is a toolchain that is used to convert ES6+ code into a version of JavaScript that both current and older browsers can interpret. Here are a few things Babel can do for you:

* Transform syntax
* Polyfill features that are missing in your target environment (through [@babel/polyfill](https://babeljs.io/docs/en/babel-polyfill))
* Source code transformations (codemods)

1. Install **Babel** locally like you did for **Nodemon**, so:  
   >**npm install --save-dev @babel/core @babel/preset-env @babel/preset-react babel-loader**
2. Usually in a project like ours, **Babel** would look for a configuration file called .babelrc, so create that file now in the root of the project. For larger projects we will use a babel.config.js file instead. Note, there is nothing in front of the period.

  
  
Use your >**touch** command to create this file, or use Atom.

1. The configuration of Babel comes in the form of presets and there is usually a preset for a given situation. For our project, we need to transform ES6 into ES5 and JSX into plain old JS, so we need the following preset in our .babelrc file.

|  |
| --- |
| **{**  **"presets": ["@babel/preset-env", "@babel/preset-react"]**  **}** |

1. Let’s create the config file for Webpack and insert some boilerplate code. From the command line or from Atom create a file in the root directory called webpack.config.js and insert the following code.

|  |
| --- |
| **module.exports = {**  **module: {**  **rules: [**  **{**  **test: /\.(js|jsx)$/,**  **exclude: /node\_modules/,**  **use: {**  **loader: "babel-loader"**  **}**  **}**  **]**  **}**  **};** |

The config file is instructing Webpack to use babel to transform .js and .jsx files to ES5 standards

1. You may do an >**npm run build** just to make sure there are no errors.

**------end of part 08-------**

## 

## part 9 – React

1. Install **react** and **react-dom** locally but NOT as a development dependency, so   
   **npm install react react-dom**  
   We installed @babel/preset-react in Part 7 above.
2. In the **src** folder, add a new index.html file with some boilerplate code but with at least one **div** tag with an **id** of root:

|  |
| --- |
| **<!DOCTYPE html>**  **<html lang="en">**  **<head>**  **<meta charset="UTF-8">**  **<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">**  **<title>React Boilerplate</title>**  **</head>**  **<body>**  **<div id="root"></div>**  **</body>**  **</html>** |

1. In Part1 we created an empty index.js file in the **src** folder, well its now time to insert code that will act as the entry point for Weback to start bundling our final file. It is also the file that will act as the main *API* file. Add all the code below to index.js

|  |
| --- |
| **import React from 'react';**  **import ReactDOM from 'react-dom';**  **//**  **const Index = () => {**  **return <div> <h2>Welcome to Skillsoft Live Learning</h2></div>;**  **};**  **//**  **ReactDOM.render(<Index />, document.getElementById('root'));** |

This file uses an arrow function. Also it will look for the HTML element having an **id** of ‘root’ and replace that element with its own message. In this case however, we say that **react-dom** is going to *render* a component called *index* into the **div** with **id** of root.

1. Next we will add some configuration so that weback knows to handle the html files APIropriately and bundle it into the **dist** folder. First install the Webpack plugin for html, so: >**npm install --save-dev html-webpack-plugin**
2. In the webpack.config.js file, import the plugin as well as the *path* module, which is part of NodeJS

|  |
| --- |
| **const HtmlWebpackPlugin = require("html-webpack-plugin");**  **const path = require('path');**  **//**  **module.exports = {**  **module: {**  **rules: [** |

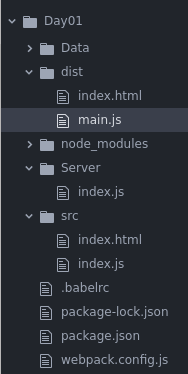
1. Now we tell **Webpack** where to pickup the template file. This will create an index.html file inside of dist folder and reference the compiled, combined JS file.

|  |
| --- |
| **const HtmlWebpackPlugin = require("html-Webpack-plugin");**  **const path = require('path');**  **//**  **const htmlPlugin = new HtmlWebpackPlugin({**  **template: "./src/index.html"**  **});**  **//**  **module.exports = {**  **module: {** |

1. Now that we have an entry point in src/index.js, let us inform **Webpack** to start its journey there. We also tell it where to output the bundled file and what name to give it.

|  |
| --- |
| **const htmlPlugin = new HtmlWebpackPlugin({**  **template: "./src/index.html",**  **filename: "./index.html"**  **});**  **//**  **module.exports = {**  **entry: "./src/index.js",**  **output: {**  **path: path.join(\_\_dirname, 'dist'),**  **filename: "[name].js"**  **},**  **plugins: [htmlPlugin],**  **module: {**  **rules: [** |

Note that we also need a **plugins** section to list all plugins, we have only one so far. Also by passing [name].js as a value of filename, WP will choose the default name which is main.js

1. Test what we have so far. First remove the entire **dist** folder and run **>npm run build**. Then run **>npm run start** and test via the browser or Postman.
2. Note if you get an error about build failed (from ./node\_modules/babel-loader/lib/index.js), make sure you install this version (3.4.7) **npm list core-js-compat**, the older version has a bug.
3. If you get an error about ‘yargs’, just install it globally: **npm i -g yargs**
4. If you get this error Error: Cannot find module 'supports-color', you may need to remove the entire node\_modules folder and the package-lock.json file and reinstall everything with >**npm install**
5. After running the command from #8, you should now see a new **dist** folder with 2 files in it. Open the HTML file in your browser directly and you should see the message from #3 above.

**------end of part 09-------**

## part 10 – Static Pages

If we ran the server now, and go to the root path with a browser, we would get the “You are on the root route!”. We want instead for the server to serve the HTML page we inserted into the **src** folder. For this we need to do 2 things, instruct **Express** to serve a static HTML file via index.js and configure package.json to bundle and then serve our files. We installed Express in Part03.

1. In Server/index.js, first require the **path** module

|  |
| --- |
| **const port = 8000;**  **const api = express();**  **const data = require('./../Data/data.json');**  **const path = require('path');**  **api.use(express.json());** |

1. Create a variable to hold the path to the /dist directory, then another variable will hold the path to the /dist/html file, which is the bundled file from Webpack.

|  |
| --- |
| **const data = require('./../Data/data.json');**  **const path = require('path');**  **const DIR\_DIST = path.join(\_\_dirname, '../dist');**  **const HTML\_STATIC = path.join(DIR\_DIST, 'index.html');**  **//**  **api.use(express.json());** |

1. Next, we will add a new script in the package.json file, so:

|  |
| --- |
| **"main": "index.js",**  **"scripts": {**  **"test": "echo \"Error: no test specified\" && exit 1",**  **"start": "nodemon Server/index",**  **"build": "webpack --mode development",**  **"dev": "webpack --mode development && nodemon Server/index.js"**  **},**  **"keywords": [],** |

Test by running the command >**npm run dev**

With this command, we first build the API, then run the API with Nodemon

1. We need to tell the **Express** API to use that static directory

|  |
| --- |
| **api.use(express.json());**  **api.use(express.urlencoded({extended:false}));**  **api.use(express.static(DIR\_DIST));**  **//**  **api.get('/', function(request, response){** |

1. Finally for this part, we change what we serve at the root route, now we serve the static HTML file instead of a single line.

|  |
| --- |
| **api.get('/getweights', function(request, response){**  **response.send(data);**  **})**  **api.get('/', function(request, response){**  **response.sendFile('HTML\_STATIC');**  **})**  **api.post('/addnewemployee, function(request, response){{** |

Notice it is **sendFile** not just **send**

1. Now refresh the browser (localhost:800) and you should see the **React** message being served via the HTML document. Note, if you do not see anything in the browser, remember to run the command >**npm run dev**.

**------end of part 10-------**

## part 11 – Styling

Adding CSS follows the same pattern as HTML, add a .css file, install the necessary packages and configure webpack.config.js.

1. We will first create a .css file in the **src** folder and add a simple style just to see the change in the root path. You can call your file styles.css.

|  |
| --- |
| **body {**  **font-family: "Verdana", sans-serif;**  **color: brown;**  **background-color: lightyellow;**  **font-size: large;**  **}** |

1. In the same **src** folder, inside of index.js, import the styles file.

|  |
| --- |
| **import React from 'react';**  **import ReactDOM from 'react-dom';**  **import './styles.css';** |

1. Remember to *install* the loaders, so >**npm install style-loader css-loader**
2. Next we configure Webpack config file with a new rule for css files. Basically copy the rule for js and jsx files and rename accordingly

|  |
| --- |
| **module: {**  **rules: [**  **{**  **test: /\.(js|jsx)$/,**  **exclude: /node\_modules/,**  **use: {**  **loader: "babel-loader"**  **}**  **},**  **{**  **test: /\.css$/,**  **use: ["style-loader", "css-loader"]**  **}**  **]**  **}** |

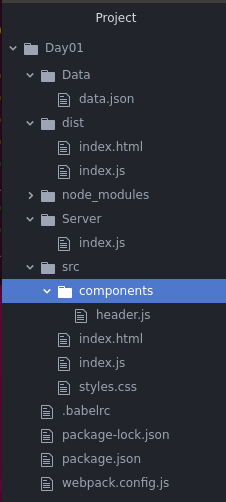
Things to notice, there is only one extension to check, CSS, so no parenthesis. We are using 2 loaders, so put them into an array and the last loader is actually loaded first, **this is important**.

1. Run the server and test

|  |
| --- |
|  |

**------end of part 11-------**

## part 12 – (Optional) React Components

1. Adding React components is easy, first create a folder inside of the **src** folder for components. After that create a header.js file inside of components.  
     
   
2. The first thing we do in any React component is to import react from react. So inside header.js

|  |
| --- |
| **import React from 'react';** |

1. Next using either a *function* or *class* return jsx from the component, in this case we use a function

|  |
| --- |
| **import React from 'react';**  **function Header() {**  **return <h3>Hello Function Component!</h3>;**  **}** |

1. Finally, export the function as **Header**

|  |
| --- |
| **import React from 'react';**  **//**  **function Header() {**  **return <h3>Hello Function Component!</h3>;**  **}**  **//**  **export default Header;** |

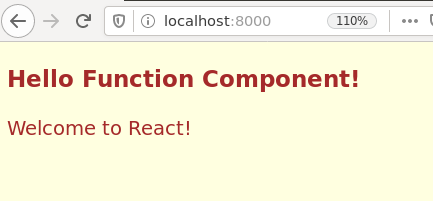
1. Now in the src/index.js import the component

|  |
| --- |
| **import React from 'react';**  **import ReactDOM from 'react-dom';**  **import Header from './components/header';**  **import './styles.css';** |

1. Now that we have the component, we can simply display it on the root route, however we can return only one parent HTML element, so we need to make sure that **<Header />** is inside of the **div** being returned.

|  |
| --- |
| **import './styles.css';**  **//**  **const Index = () => {**  **return <div> <Header />Welcome to React!</div>;**  **};**  **//**  **ReactDOM.render(<Index />, document.getElementById('root'));** |

You would have to stop the service and run **npm run dev** again.

1. You should now see something like this:  
     
   

|  |
| --- |
|  |

**------end of part 12-------**

## Appendix A – Simple HTTP Server

* 1. Choose a directory and run: npm install http-server -g
  2. To start the server, find a directory and type http-server .

(note the period signifies that you are starting the server on the current directory that you are in at the moment)

* 1. If the above does not work, you can try installing the server as a dev server
  2. If as a dev server, then your command will be:   
     sudo npm install –save-dev http-server
  3. Then in your package.json file, use the script section to point to that package, so:  
      "scripts": {

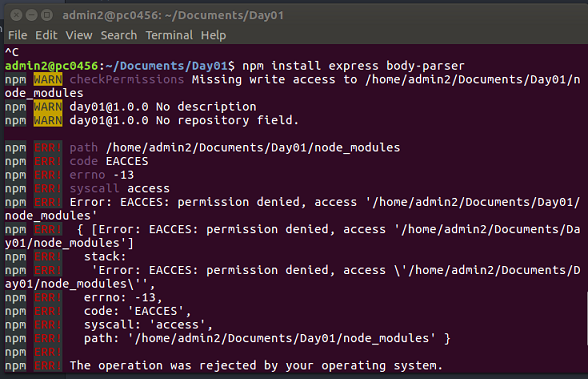
"start": "http-server ."

},

* 1. Now from any directory just type npm start

## Appendix B – Permission Denied Errrors

You may get into a situation like this:



If this hAPIens on a Linux system, try running this command:  
sudo chown -R $USER /usr/local/lib

OR

sudo chown -R $USER /home/admin2/.node\_modules\_global/lib/node\_modules