**Why create your own configuration?**

There are certain reasons that make creating your own React configuration make sense.

* You want to learn how to use tools like webpack and Babel on your own.
* In case if you need more tools to integrate in your project.
  + Adding webpack loaders for less, sass
  + Doing server side rendering
  + Using new ES versions
  + Adding MobX and Redux
  + Making your own configuration just for learning sake

**Step1: Initialize NPM project by running:**

yarn init -y // -y parameter answers all your questions.

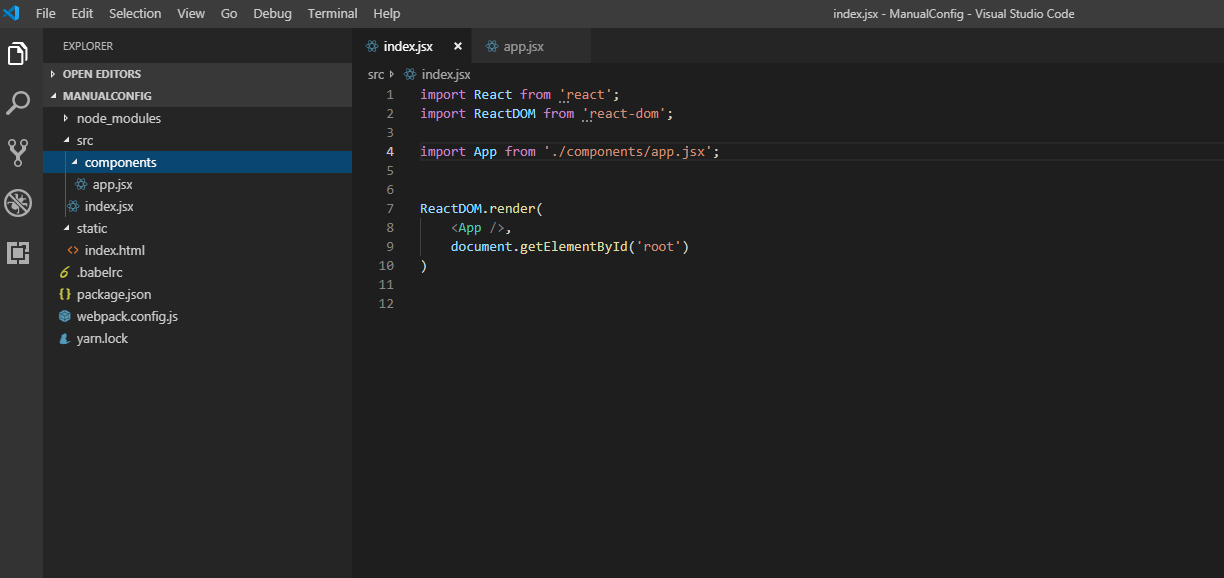
Or

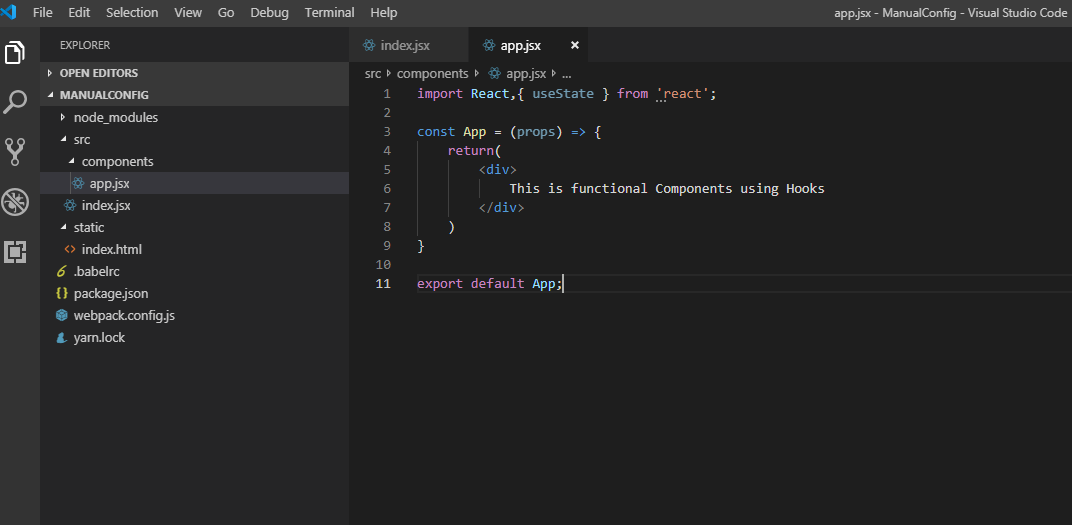
npm init

**Step2: Install react:**

yarn add react react-dom or npm i react react-dom

**Step3: Create a “Folder Structure” for our component.**





root

| ----- src

| |- components

| | |-- app.jsx – a sample component which prints a message.

| |

| |- index.jsx -- This is our entry point

|

| ------static

| -- index.html

app.jsx

import React,{ useState } from 'react';

const App = (props) => {

return(

<div>

This is functional Components using Hooks

</div>

)

}

export default App;

index.jsx

import React from 'react';

import ReactDOM from 'react-dom';

import App from './components/app.jsx';

ReactDOM.render(

<App />,

document.getElementById('root')

)

Index.html

<!DOCTYPE html>

<html>

<head>

<title>My React Configuration Setup </title>

</head>

<body>

<div id="root"> </div>

</body>

</html>

**Step3: Configure Webpack:**

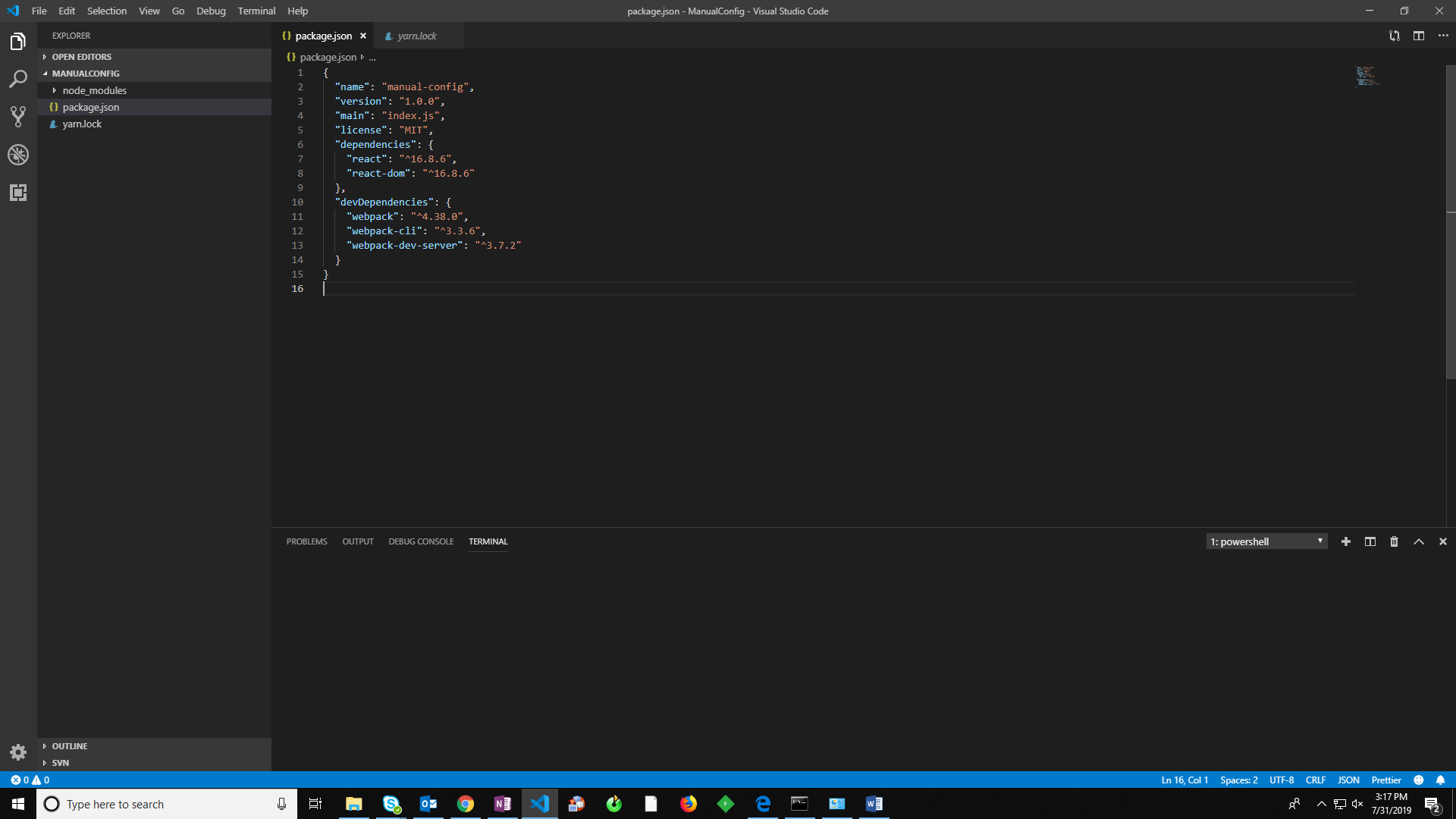
**Webpack:** It’s a very popular and powerful tool for configuring not only React, but almost all front-end projects. The core function of webpack is that it takes a bunch of JavaScript files we write in our project and turns them into a single, minified file, so that it will be quick to serve.

yarn add webpack webpack-dev-server webpack-cli html-webpack-plugin -D

// -D installs packages as dev dependencies

* **Webpack** — which include all core webpack functionality
* **webpack-dev-server** — this development server automatically rerun webpack when our file is changed
* **webpack-cli** — enable running webpack from the command line
* **html-webpack-plugin -** The **[HtmlWebpackPlugin](https://github.com/jantimon/html-webpack-plugin)** simplifies creation of HTML files to serve your webpack bundles. This is especially useful for webpack bundles that include a hash in the filename which changes every compilation. You can either let the plugin generate an HTML file for you, supply your own template.

After installing above packages your package.json look like this(Please refer below screen shot):



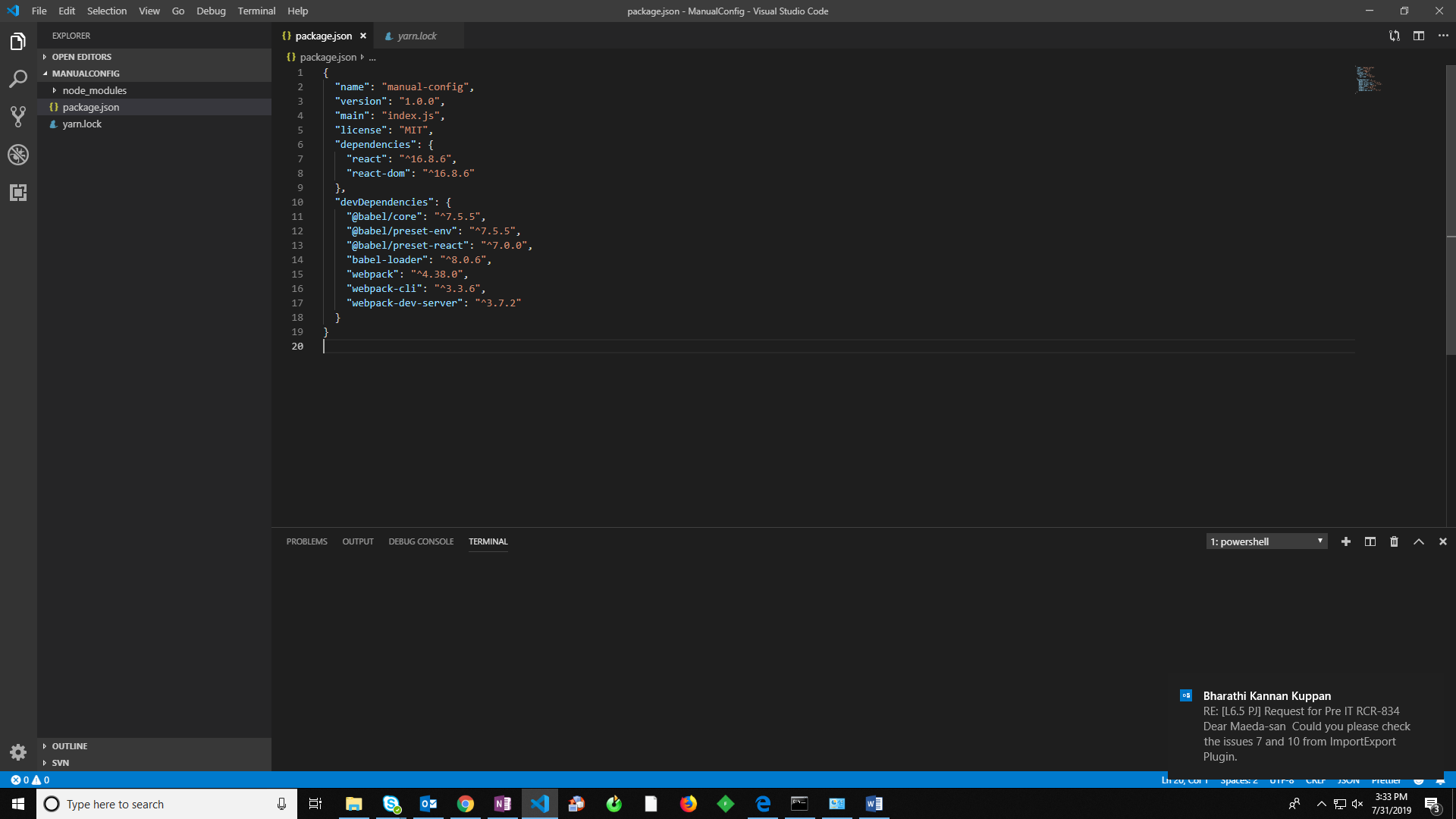
**Step 4: Babel**

**Why do we need babel?**

The current version of React uses ES6 to ES8 syntax. We need [Babel](https://babeljs.io/) to compile the code written in those syntaxes back to code the browser can understand. Babel is there to ensure backward compatibility.

* **@babel/core** is the main dependency that includes babel transform script.
* **@babel/preset-env** is the default Babel preset used to transform ES6+ into valid ES5 code. Optionally configures browser polyfills automatically.
* **@babel/preset-react** is used for transforming JSX and React class syntax into valid JavaScript code.
* **babel-loader** is a webpack loader that hooks Babel into webpack. We will run Babel from webpack with this package.

After installing above packages your package.json look like this(Please refer below screen shot):



Once that is done, open your package.json file and add the following to your scripts:

“scripts”: {

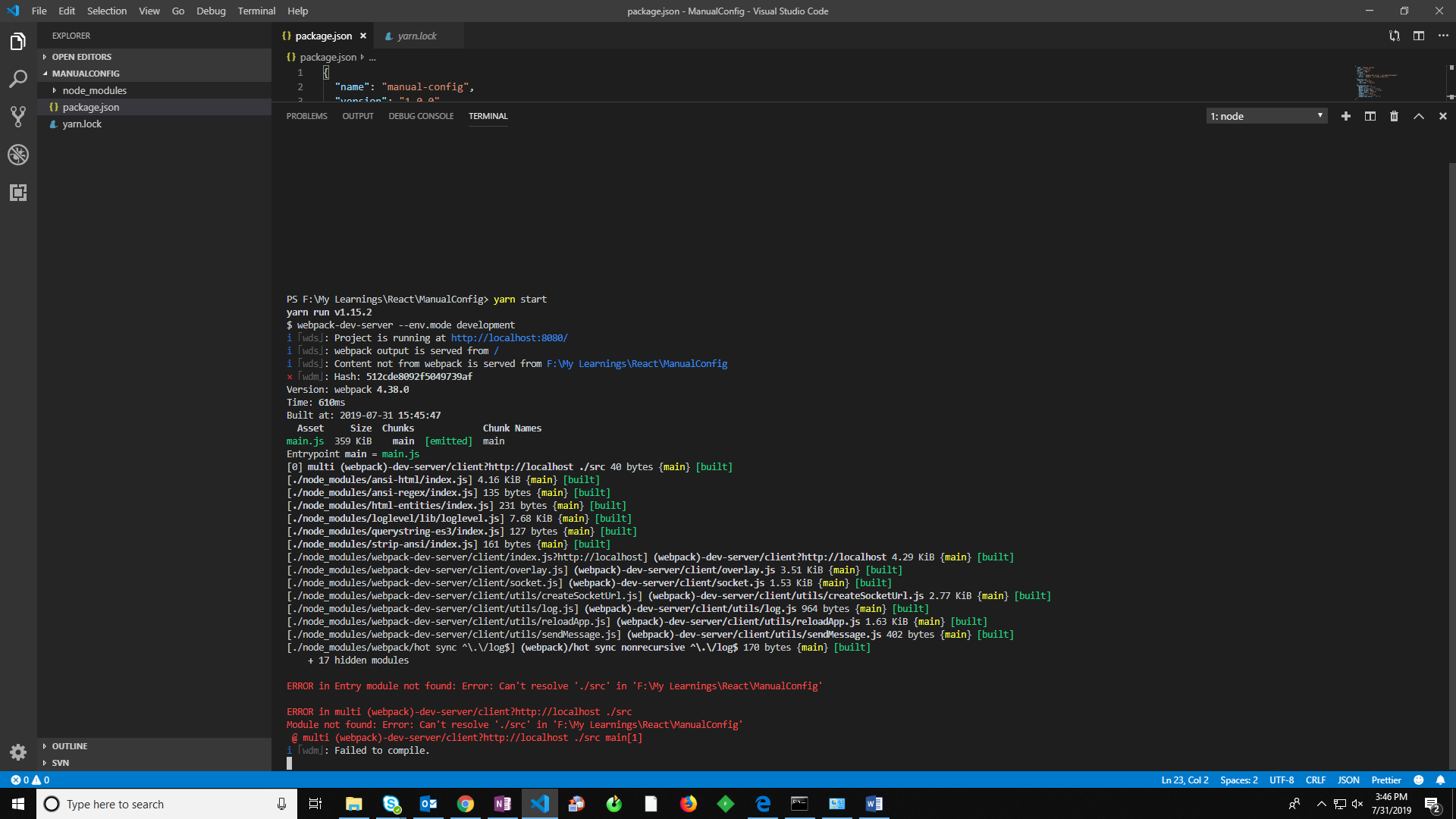
“start”: “webpack-dev-server –env.mode development”,

“build”: “webpack –env.mode production”

}

Now save it, go back to your terminal and we will try to test the newly added code.

Run: yarn start below error we will get when we run the script “start”



The script we added to our package.json is trying to run Webpack configuration. so it breaks.

### Configuring webpack.config.js & .babelrc files

Let’s write the configurations for webpack.config.js file and .babelrc file.

**Configuring Babel:**

In the root folder, create a .babelrc file to hold all of the configurations.



**What the presets do:**

* **“babel-preset-env”** tells Webpack to compile all syntax to ES5 (which browsers understand)
* **“babel-preset-react”** adds support for jsx syntax.

**Configuring Webpack:**

We also need a file to hold our generic Webpack configurations for our app. Let’s create webpack.config.js file in the root folder.

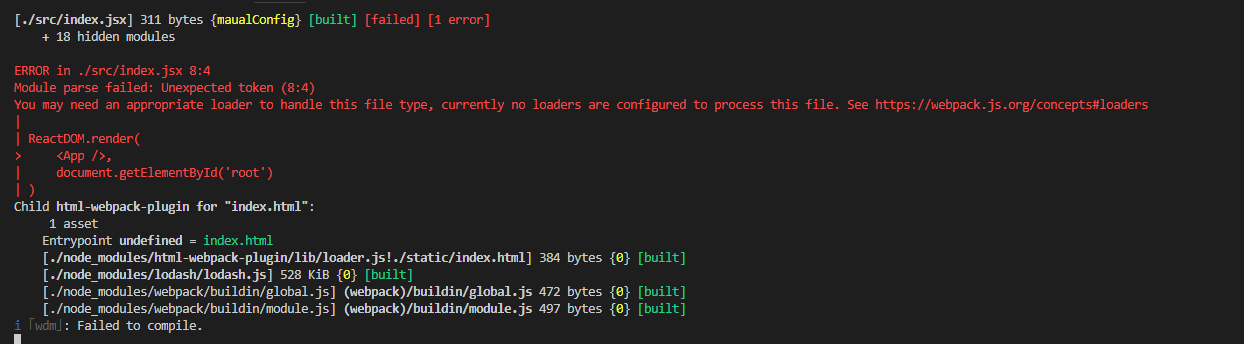


**mode:** tells webpack what mode we are currently in.

**entry:** We also set entry point which is going to be “index.jsx” because i.e the top file in react app. It renders out our app into DOM.

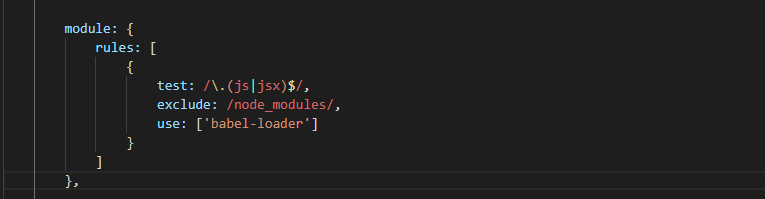
**output:** This option instructing webpack on how and where it should output your bundles, assets and anything else you bundle or load with webpack. (In this case all our bundles will be placed under **“Public”** folder.

Run: yarn start below error we will get when we run the script “start”



Error clearly telling us, we have not configured appropriate loaders. So, what are those? The answer is simple babel-loader

Let’s configure it. The best way to use loaders is to specify them in the **webpack.config.js** file. To do that, you need to add a **module.rules** property.

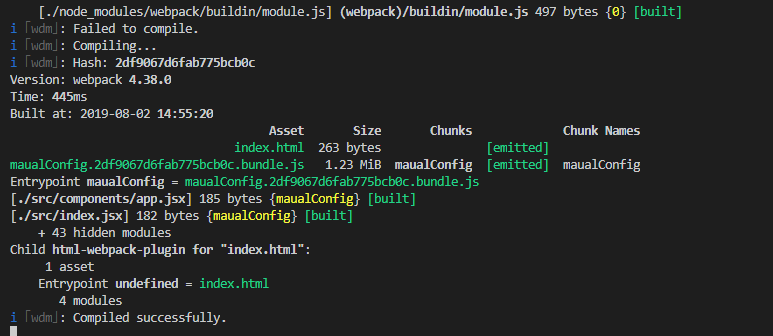


**rules:** The property ***rules*** is an array of all of your loaders. These rules will be applied to every file, that matches the ***test***property of the rule. This is, in fact, a **regular expression**.

**use:** The property ***use***is an indicator of which loader should be used for matching files.

**exclude:** we used ***exclude*** property here, which is also a **regular expression**. If the path of any file will match this expression, the file will not be transpiled.

Run: yarn start: Awesome, Our project gets compiles successfully!



Open chrome browse and type : <http://localhost:8080/> and you see the message.

**Handling Styles with Webpack:**

We need to style our application, so we can customize webpack configuration to handle styles in our application.

* Create style folder in your “src” folder.
* Create an app.scss file in src folder.

root

| ----- src

| **|- - style**

| | **| - app.scss**

| |- components

| | |-- app.jsx – a sample component which prints a message.

| |

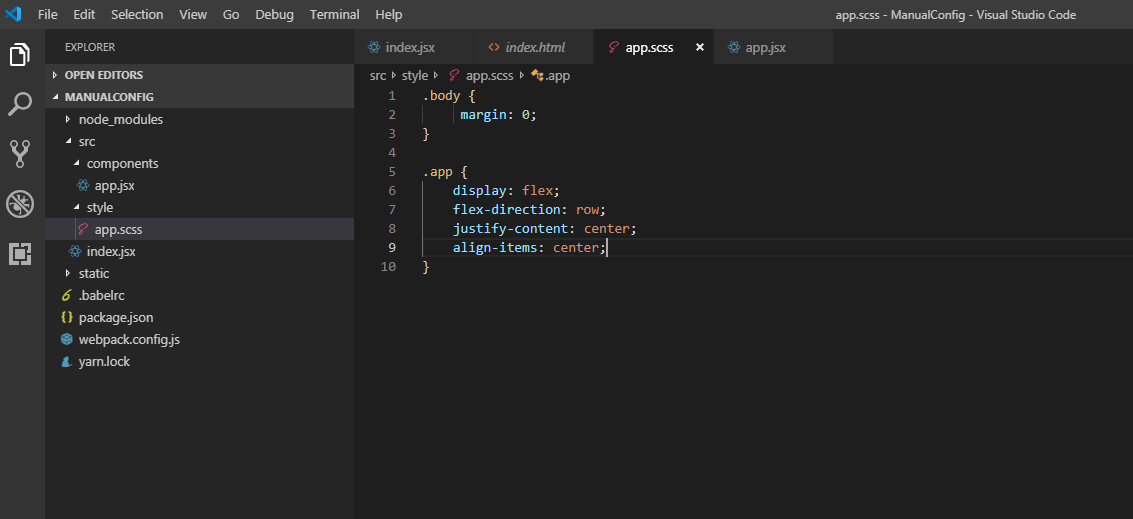
| |- index.jsx -- This is our entry point

|

| ------static

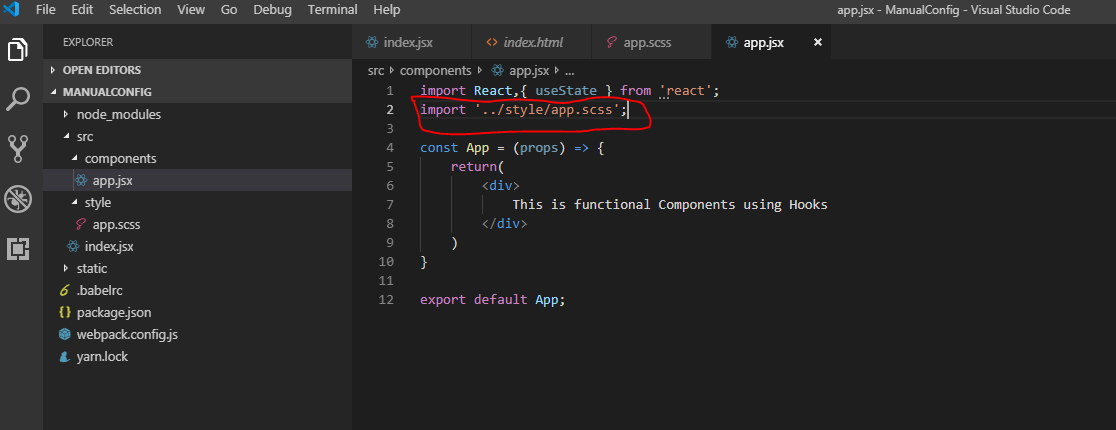
| -- index.html

* Add below code in app.scss file and save the file.

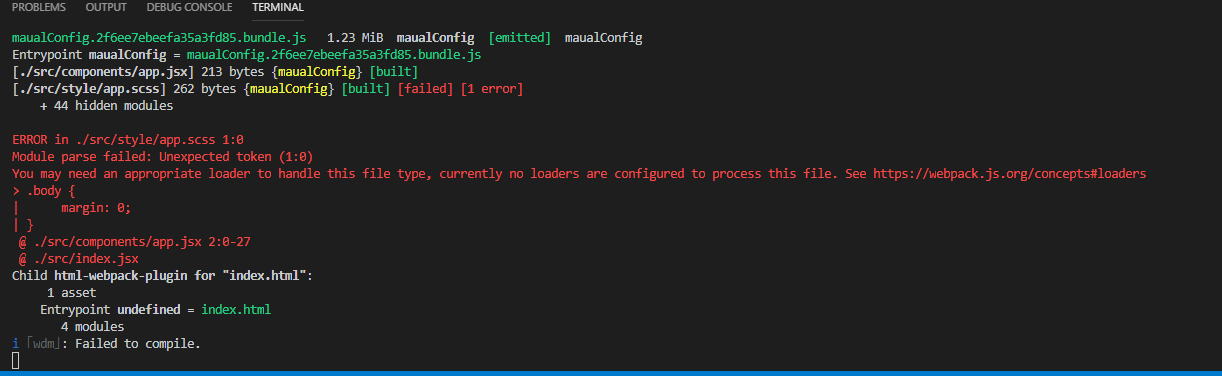


After saving your file nothing will happen that’s because “style/app.scss” file is not used in any component, so webpack won’t attempt to bundle it.

Go ahead and import app.scss file in app.jsx and save the file.



Your webpack will through below error when you import and save the file.



Webpack doesn’t know what to do with .sass/.scss/.css files. We have to add a loader to tell Webpack how to handle the stylesheets we’re using.

Let’s install below packages to handle styles within react application.

yarn add style-loader css-loader sass-loader node-sass

or

npm i style-loader css-loader sass-loader node-sass

**sass-loader:** Which compiles styles written in .scss file to traditional css (.css file).

**css-loader:** Which interprets .css file and result will be passed to “style-loader”.

**style-loader:** which adds css codes inside <style> tags in index.html

We are going to be implementing these loaders in different ways based on the environment.

### Setting up environment specific Webpack configurations

Before setting up the loaders, we have to split our configurations. When shipping out to production, we want bundles as light as possible. But we aren’t as concerned with this for development. So we would treat stylesheets differently for both modes. Let’s create the environment specific configurations.

Create **webpack.development.js** and **webpack.production.js** in the config folder. They will hold configurations specific to their mode.

**webpack.config.js** holds our generic configuration.

root

| ----- src

| |- - style

| | | - app.scss

| |- components

| | |-- app.jsx – a sample component which prints a message.

| |

| |- index.jsx -- This is our entry point

|

| ------static

| | -- index.html

|

| ------ config

**|-- webpack.development.js**

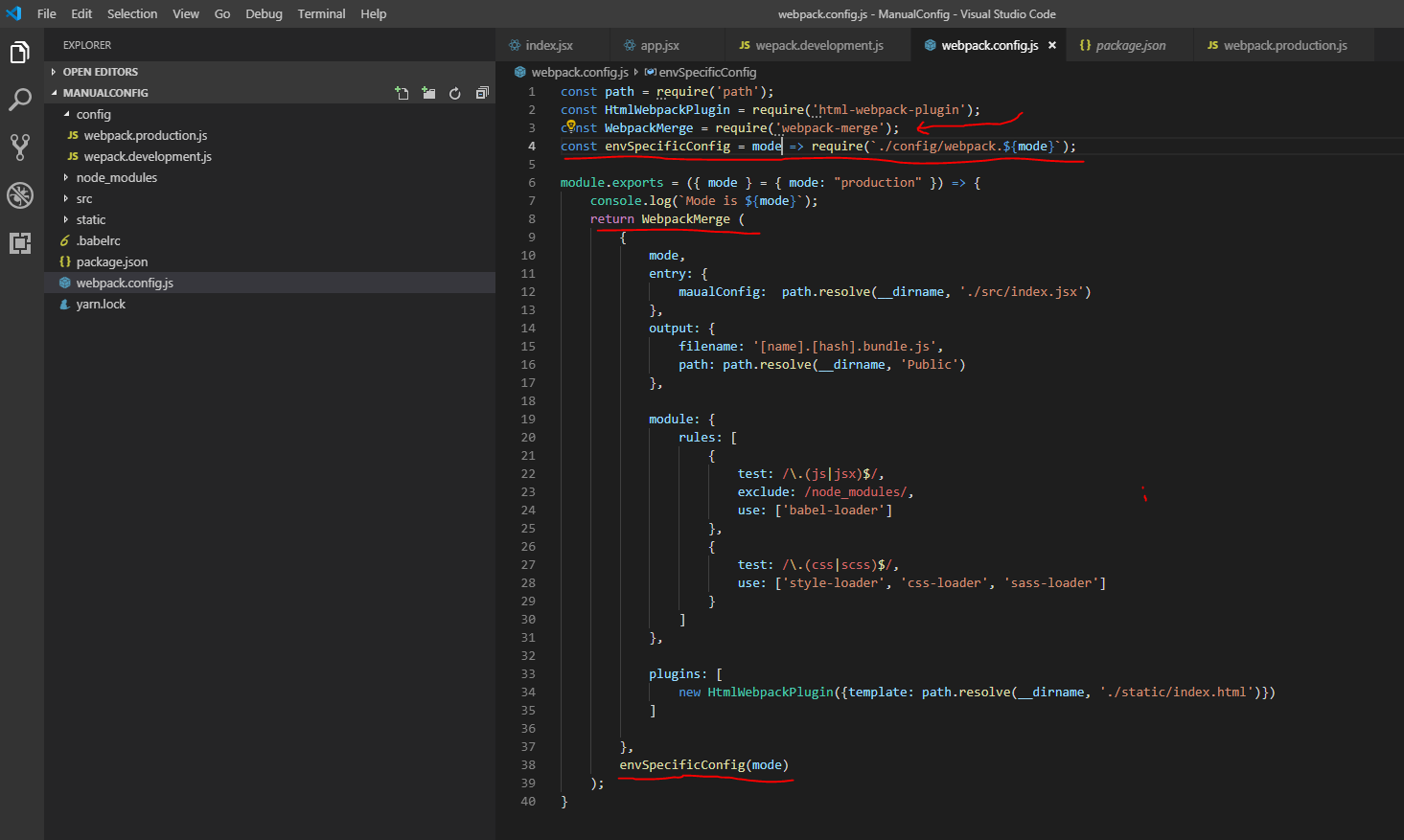
**|-- webpack.production.js**

To pass environment specific configurations, we need a utility package called **[webpack-merge](https://www.npmjs.com/package/webpack-merge" \t "_blank)**.

Let’s install

yarn add webpack-merge -D **or**  npm install webpack-merger -D

Go to webpack.config.js file and let’s add

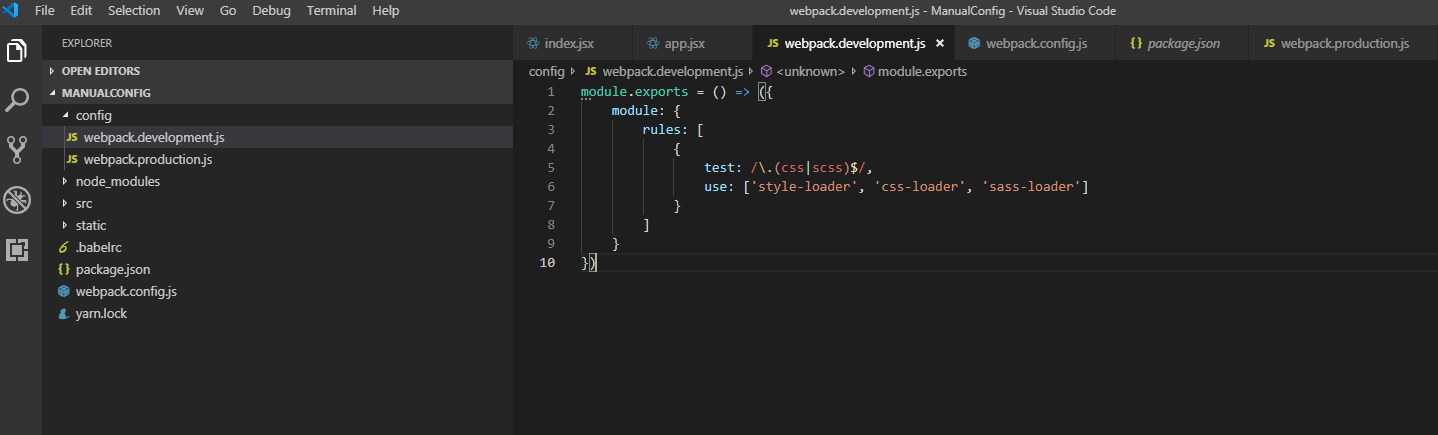


Here, we required the webpack-merge package, then we created a function **envSpecificConfig** that loads the configuration that matches the mode we’re in. We’ll pass **envSpecificConfig** as the second argument to WebpackMerge. WebpackMerge then adds configurations from it to the generic configuration.

Now that we have that flow setup, let’s define our environment specific configurations.

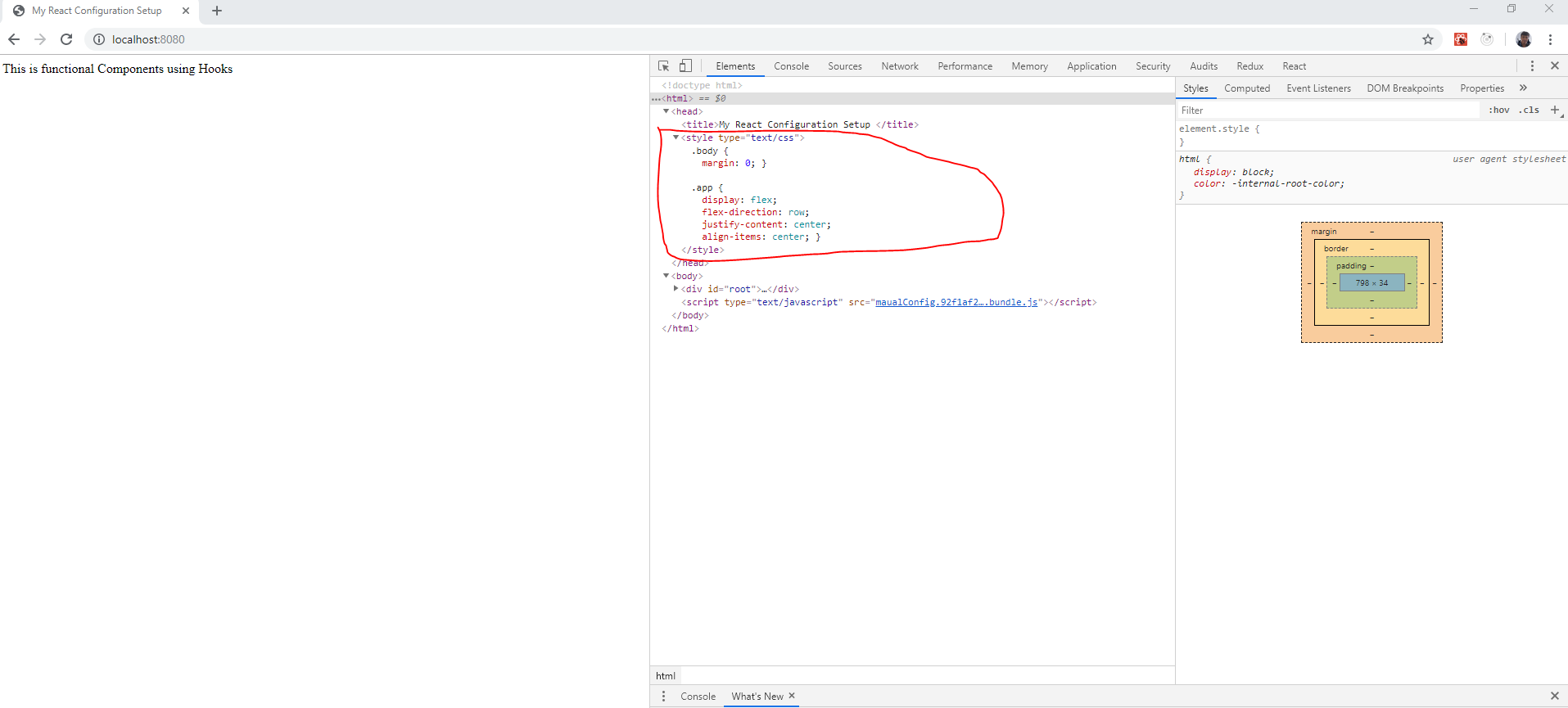
### Setting up development specific configurations

Let’s open **webpack.development.js** file and configure Style loaders



Using above configuration will work like below.

* Webpack will try to resolve the **“app.scss”** file
* The file name will match the **/\.(css|scss)$/** regular expression *(Which means any file having name matching with the regular expression i.e. any .scss or .css file, should be compiled with the chain of loaders specified in the use array)*
* The file will be interpreted by the “sass-loader” and that would help webpack to compile saas to css.
* **“css-loader”** will take this css output of *sass-loader* and will also process any other .css files we have in our application and pass on the .css to style-loader.
* **“style-loader”** which will then do the job of putting the css codes inside <style> tags in our index.html (Please refer below screen shot).



Note: A very important thing to note is that **the loaders chain is executed in reverse order**

### Setting up production specific configurations

We will be using below plugins one by one to optimize our production build

* mini-css-extract-plugin
* optimize-css-assets-webpack-plugin
* terser-webpack-plugin

**mini-css-extract-plugin:** This plugin extracts CSS into separate files. It creates a CSS file per JS file which contains CSS. It supports On-Demand-Loading of CSS and SourceMaps.

It builds on top of a new webpack v4 feature (module types) and requires webpack 4 to work

**optimize-css-assetz-webpack:** It will search for CSS assets during the Webpack build and will optimize \ minimize the CSS

**terser-webpack-plugin:** This plugin uses [terser](https://github.com/terser-js/terser) to minify your JavaScript

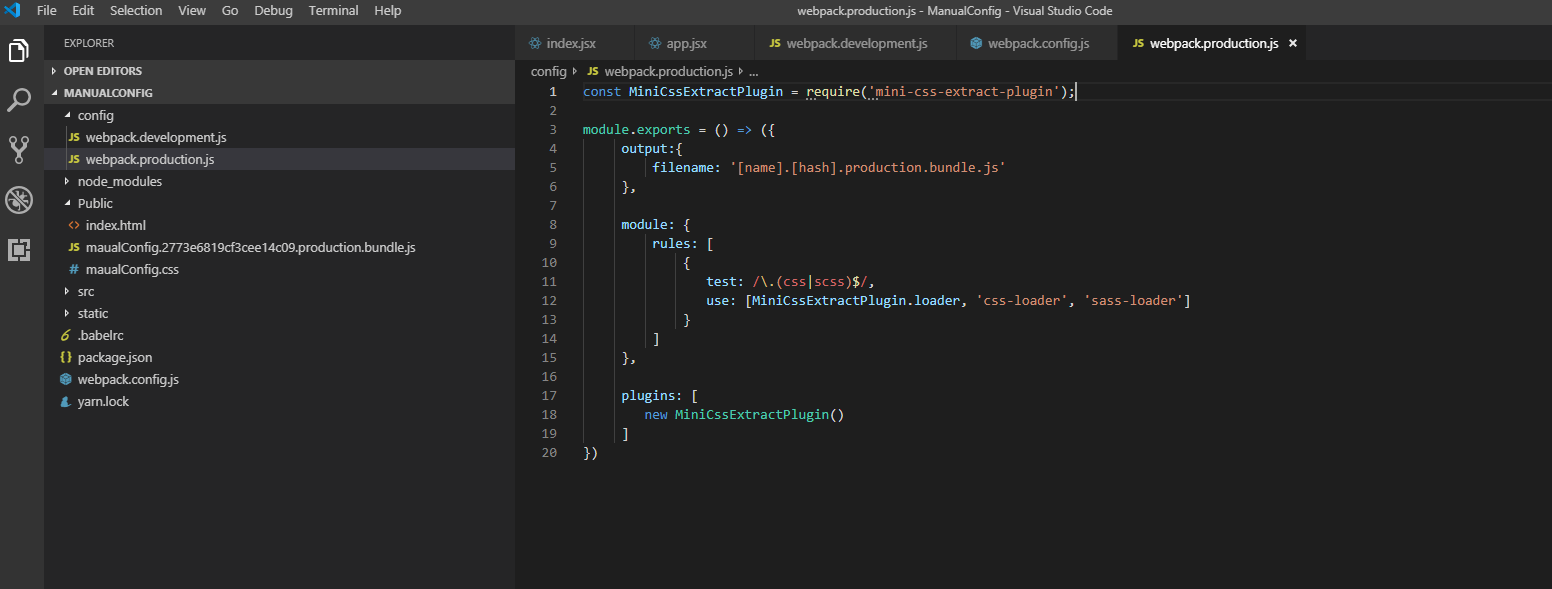
To begin, we will install above plugins:

yarn add mini-css-extract-plugin optimize-css-assets-webpack-plugin terser-webpack-plugin -D

OR

Npm install mini-css-extract-plugin optimize-css-assets-webpack-plugin terser-webpack-plugin -D

Let’s open **webpack.production.js** file and add below configurations.

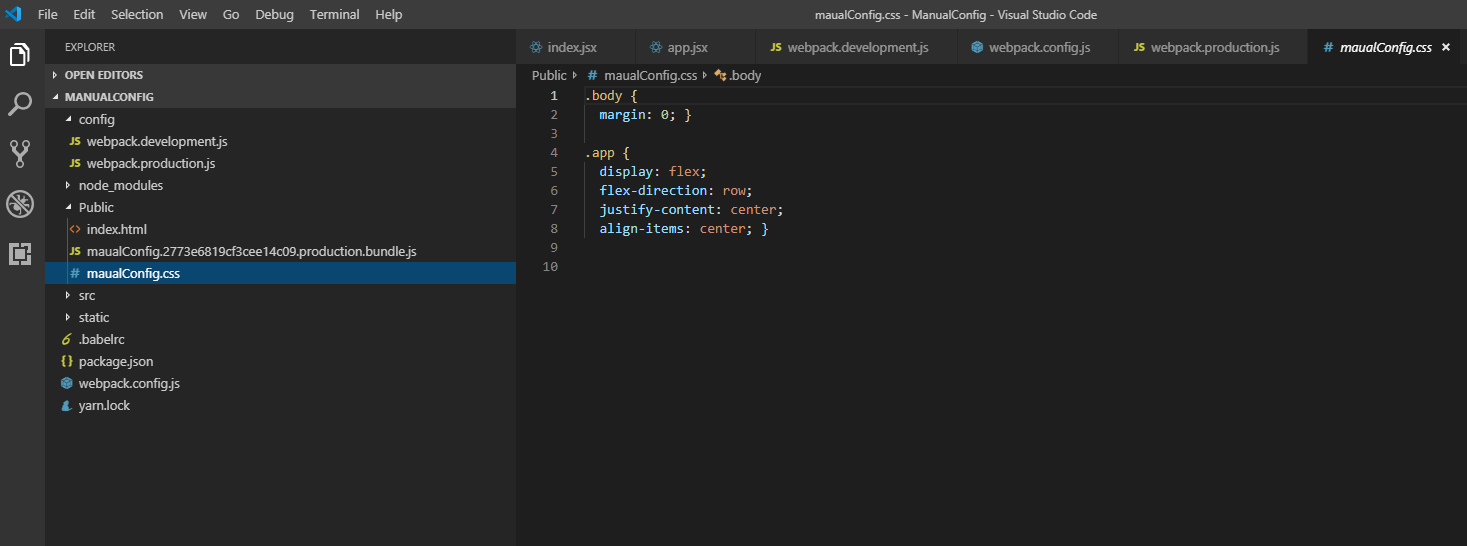


From the above configuration, we have imported **mini-css-extract-plugin** and configured in **module.rules** property, also in **plugins** property.

Now, let’s build the bundles. To build run below command

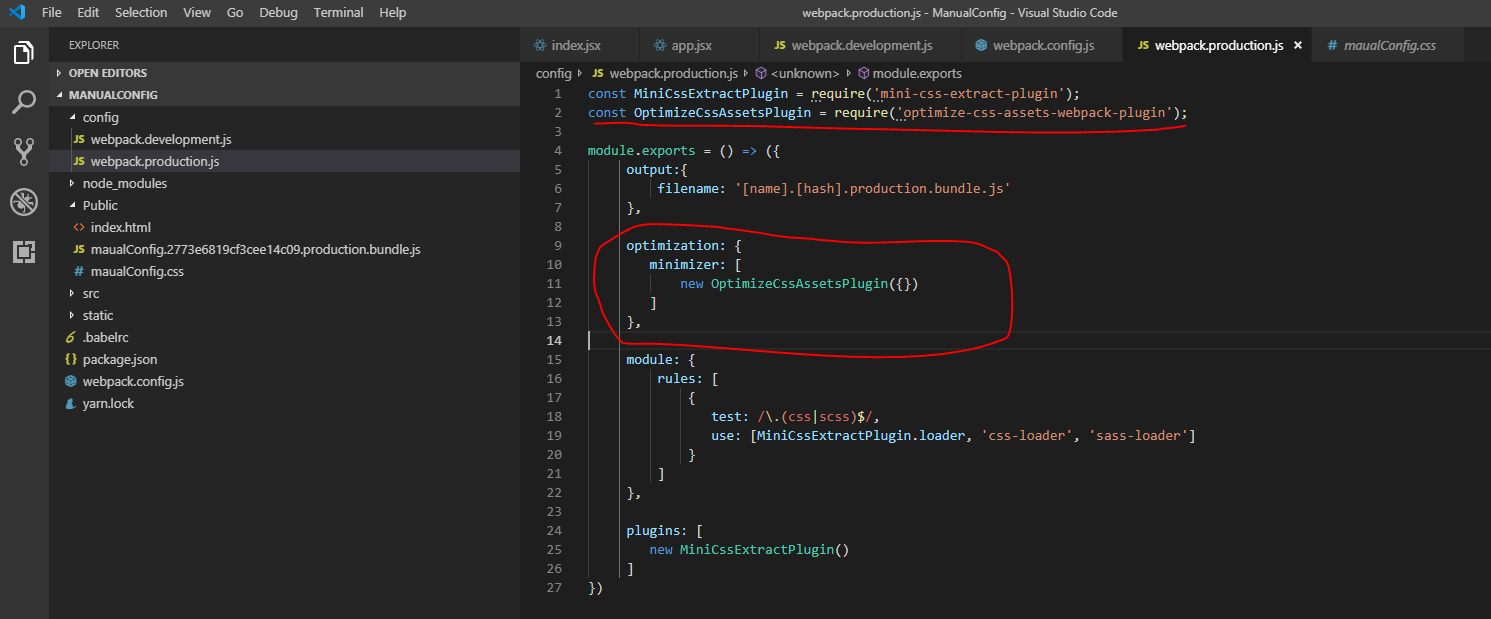
yarn run build

After running above command, the artifacts (bundles.js and bundle.css) will be generated and placed under **“Public”** folder. Let’s see what these bundles contains (Please refer below screen shot).

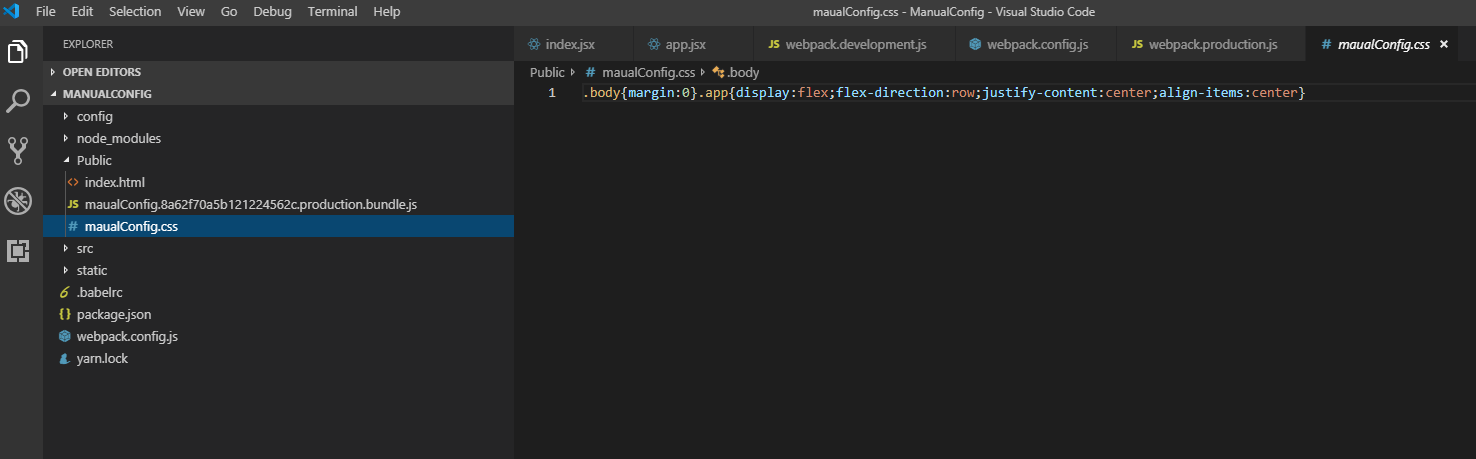


From the above screen shot, we have extracted css styles from JavaScript file and placed under separate .css file i.e manualConfig.css with help of mini-css-extract-plugin. One thing we should notice here contents of this file is not optimized that means it’s not minified. Let’s do it.

Go to **production.webpack.js** file and add below configurations

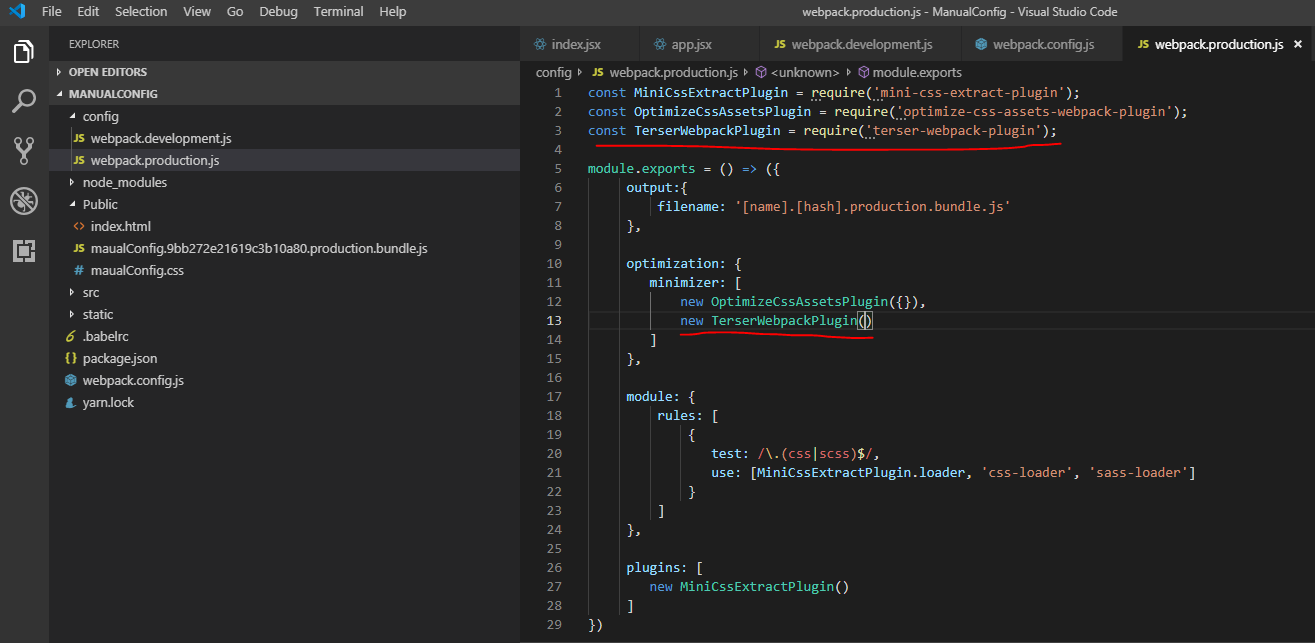


Now build the code (yarn run build)



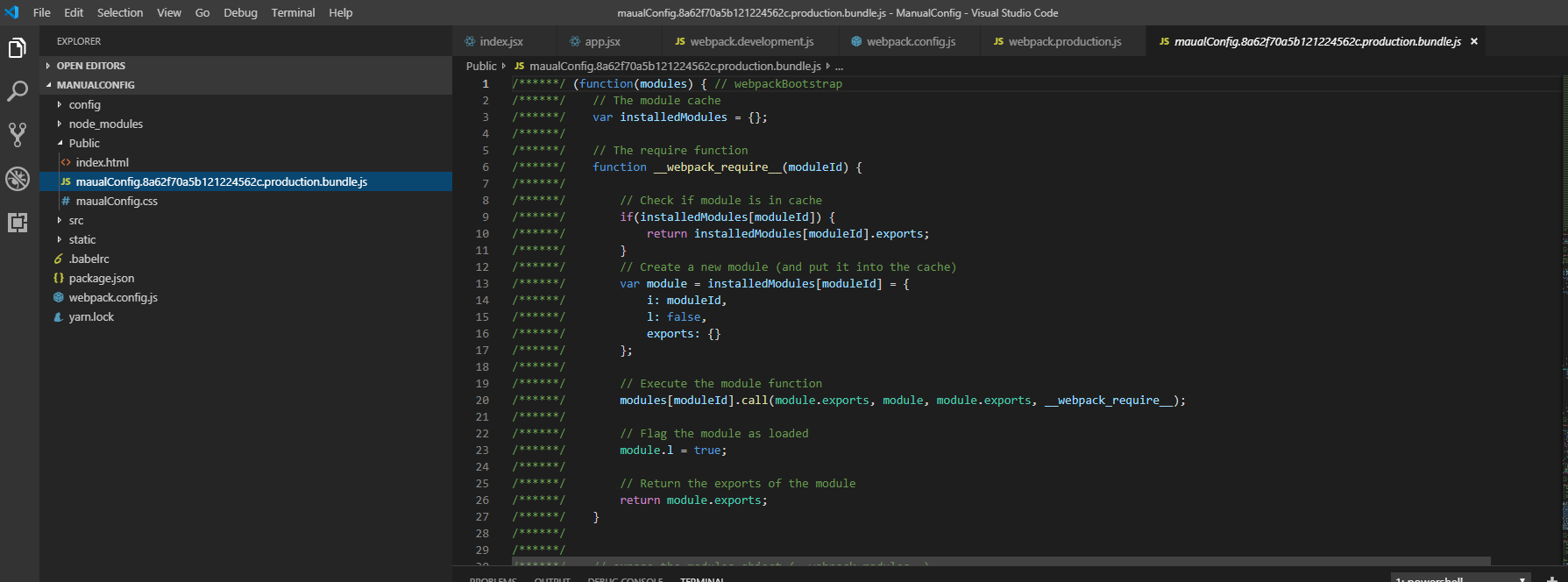
We can see .css code is minimized.

Let’s minimize JavaScript code using **“terser-webpack-plugin”** to do that go to webpack.production.js file and add below config.



File: webpack.production.js

Before adding TerserWebpack Plugin our production bundle.js looks like this (below screen shot)



After adding TerserWebpack Plugin, our production bundle.js looks like this (Please refer below screen shot).

