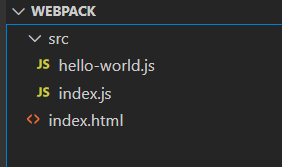
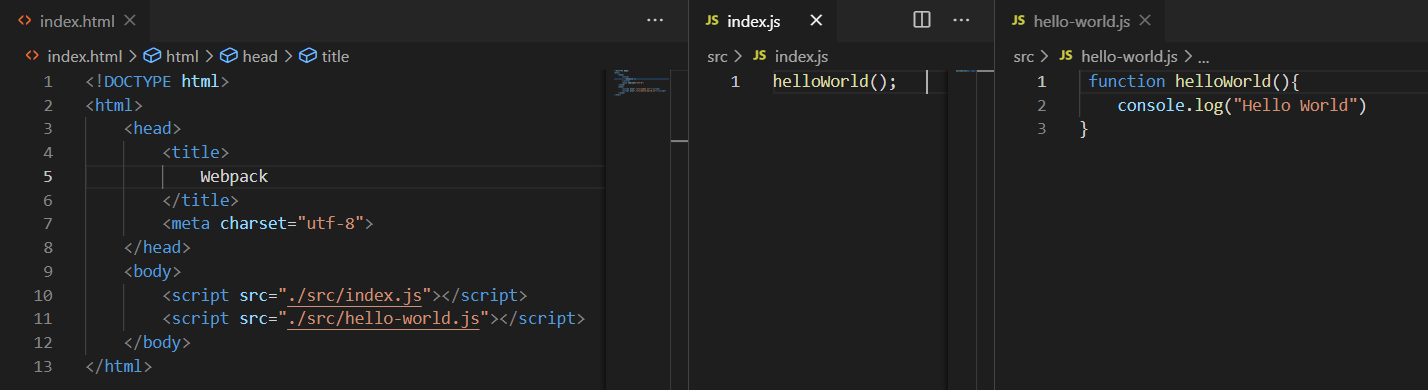
**Webpack Project**

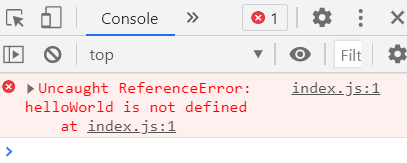
* Create project webpack with file structure.



* Content



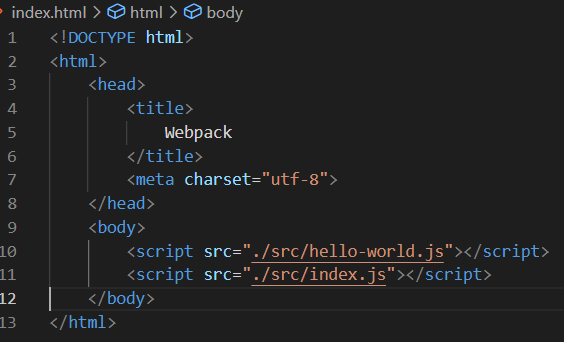
* Run index.html file in browser->



**Why error?** -> because the dependency needs to be specified first and then the dependents ->   
here index.js is dependent on hello-world.js

**Dependency**- hello-world.js

**Dependant**- index.js

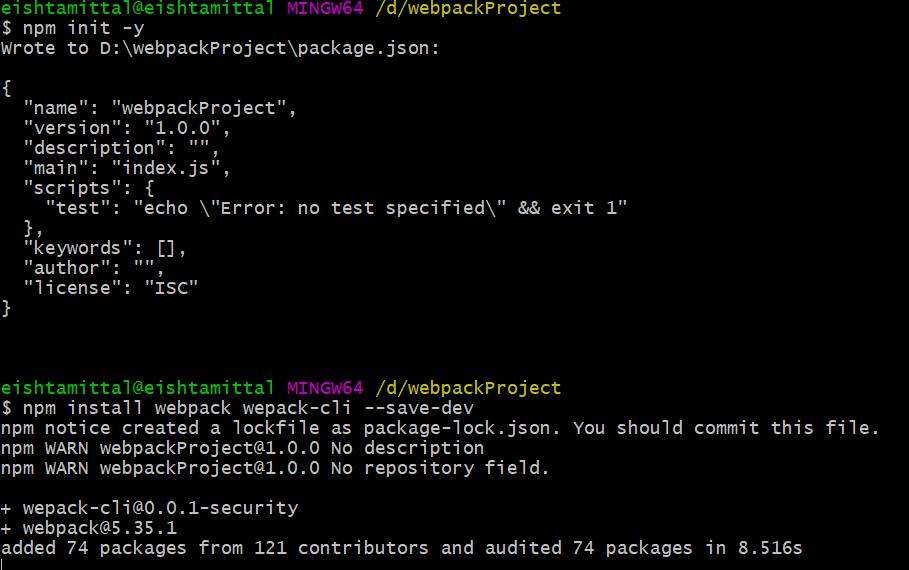


* Move <script> with src hello-world.js at the top.  
  Similarly when one file depends on other file, it should be scripted after its dependency is loaded.
* For multiple dependencies, it is difficult to order them in the scripts and include all of them in index.html.
* Here WEBPACK comes into play ->   
  it bundles all the dependencies into bundle.js file and then we can include this file in the index.html through scripts.

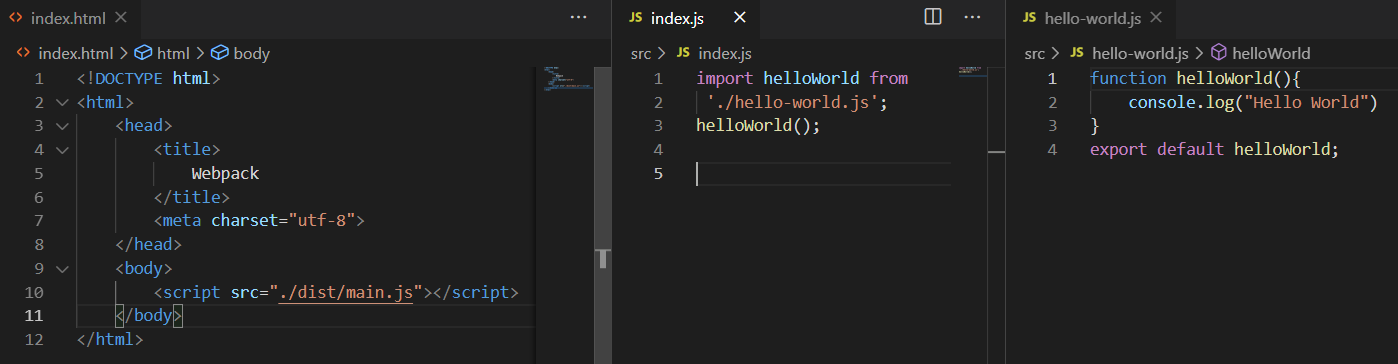
Steps->

1. npm init -y -> generates default package.json file
2. Install webpack

npm install webpack webpack-cli –save-dev

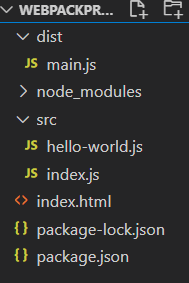


**Integrating webpack into our JS app**



Run -  
npx webpack

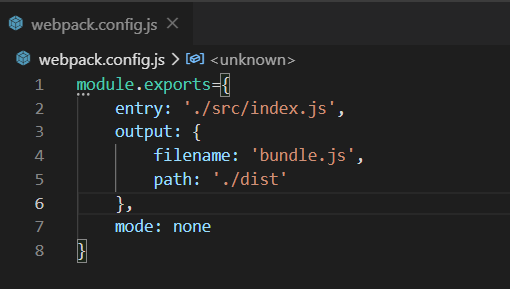
Folder structure:-



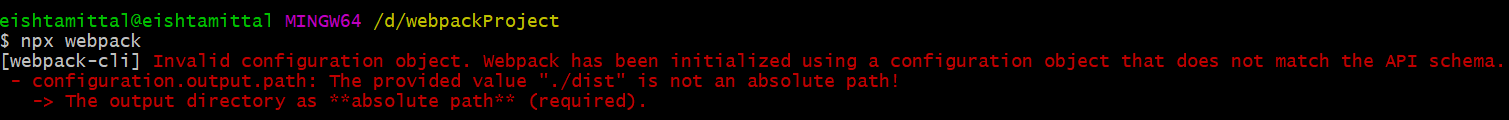
By default , output file is ‘./dist/main.js’

**Custom Webpack configurations**

* Webpack.config.js

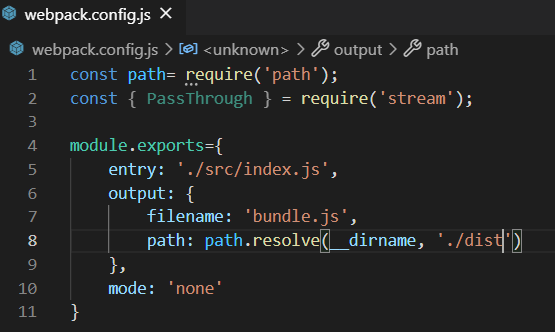


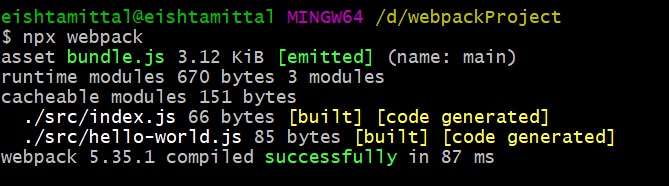
* npx webpack

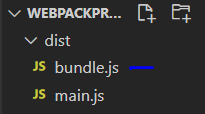


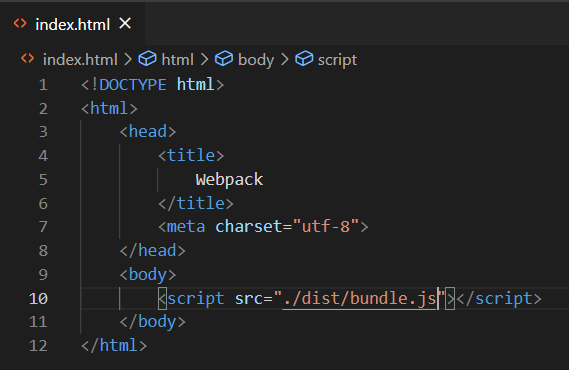
Resolve by using path in webpack.config.js

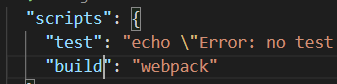
* in config file we cannot use ES modules, so we use the require.js
* using path, if dist folder doesn’t exist, it will create the folder and put bundle.js file in it
* you can delete the main.js as now the output comes in bundle.js.

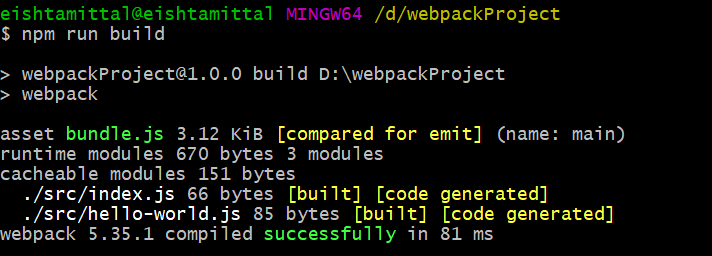
-





* Now refer to this bundle.js in the index.html
* We can also add the webpack command to the scripts in package.json to build the project





**Asset Modules**

Webpack allows to import lot of stuff in the JS file through: -

Asset modules and Loaders

* Asset modules is new feature in webpack 5.
* Allows to introduce assets in the JS file without installing additional dependencies .
* These include images, svgs, text files, fonts etc.

There are 4 types of Asset modules-

1. **Asset/ resource**-

* create a new file in the output.
* for large resources like images.

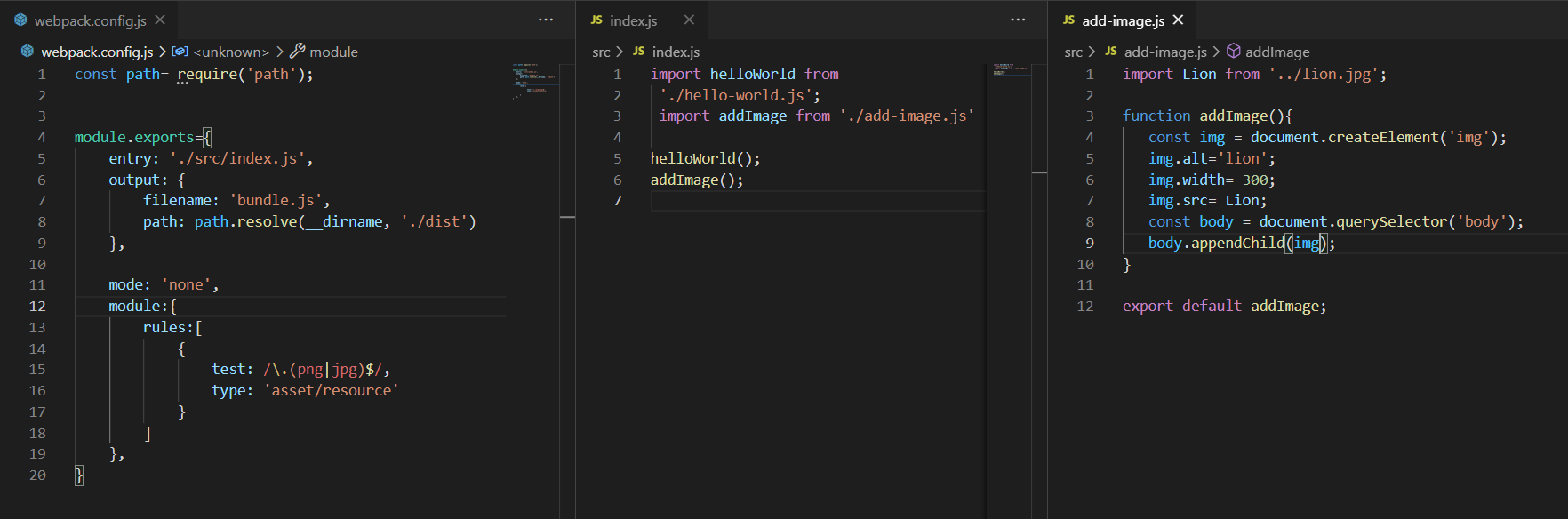
1. **Asset/inline**-
   * does not create a new file in the output.
   * for svgs and stuff.
2. **Asset** -> combination of first and second types  
   webpack decides itself   
   if the file is light than 8kb then asset/inline else asset/resource
3. **Asset/source**-> for files that contain data.

It imports the data as it is and put it in bundle as a string of text

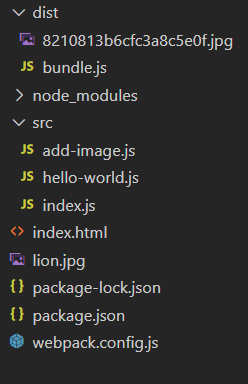
**Handling Images with Webpack**

Add an image to your project and add it to the body as given: -

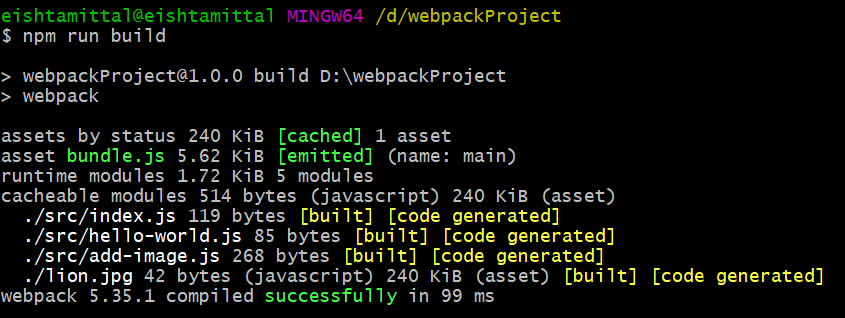
* Create a file add-image.js.
* Import it in index.js.
* Add a rule in webpack.config.js so that webpack can read it.

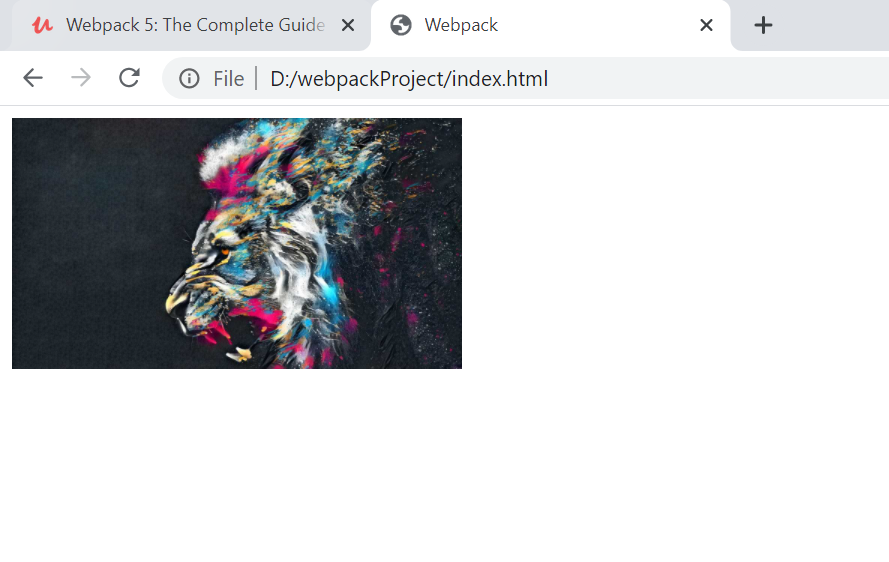


Folder structure



* An image is added to dist folder when the webpack build is run.





**Public Path variable in webpack.config.js**

* Tells where to import our files from
* Used when the files come from a cdn or express.js serves the resources

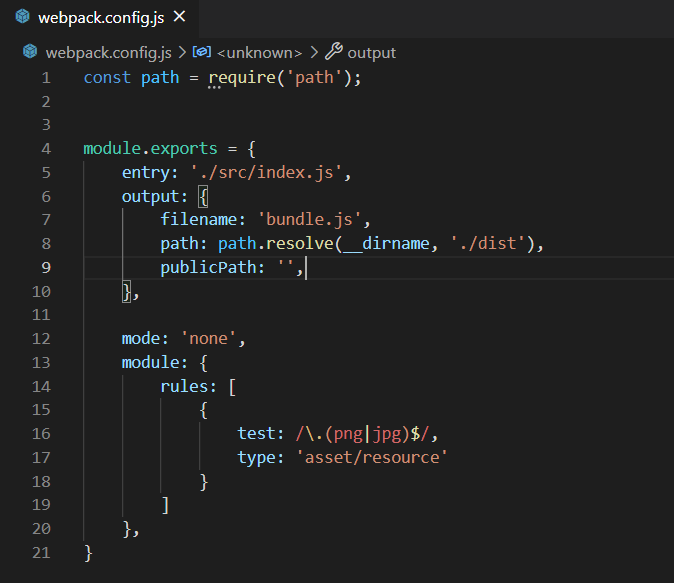
By default, value **after version 4** of webpack,   
publicPath: ‘auto’

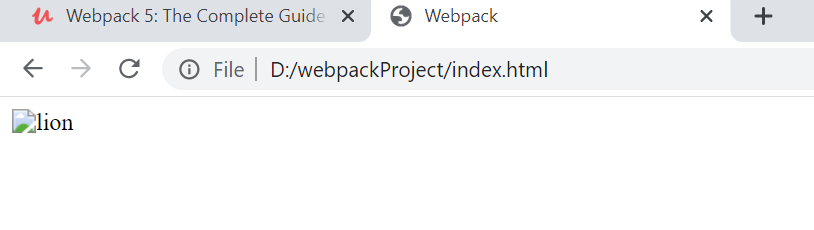
**Before version 4**,

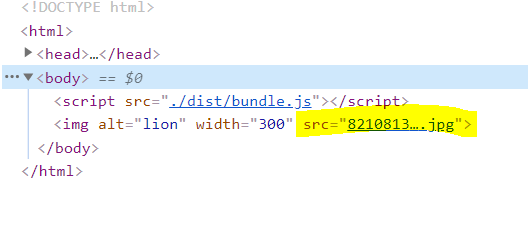
publicPath:’’ 🡪 empty string

example of both

* **publicPath**: ‘’ (empty string)

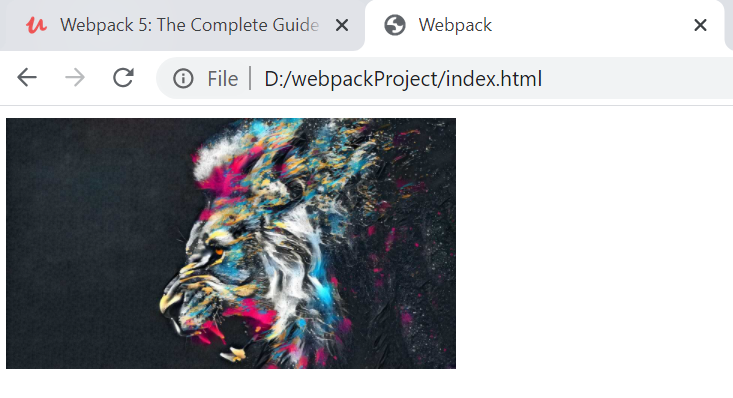


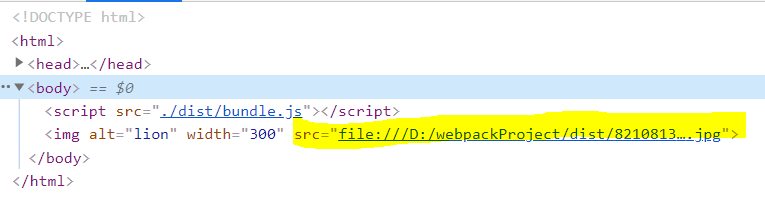




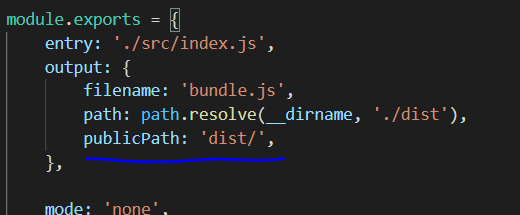
* **publicPath**: ‘auto’

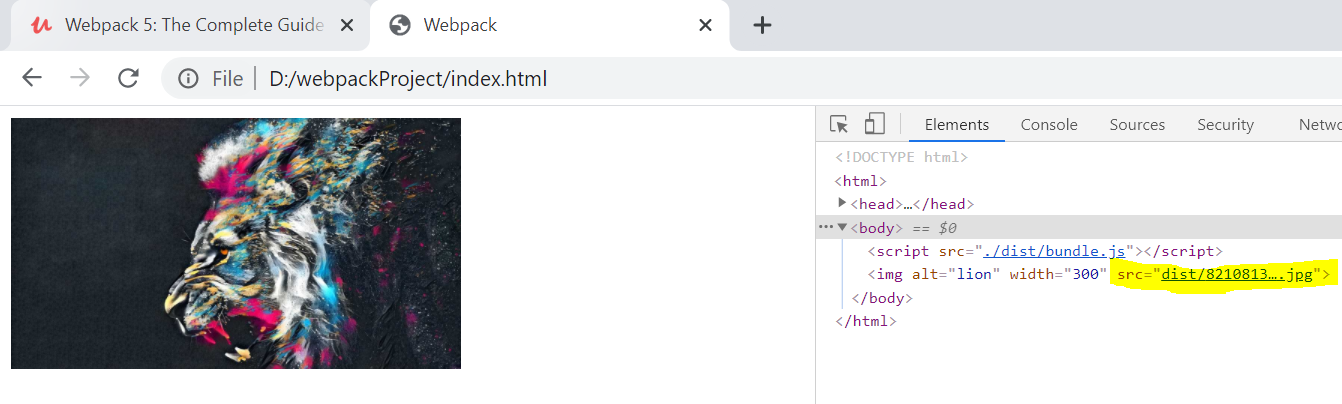






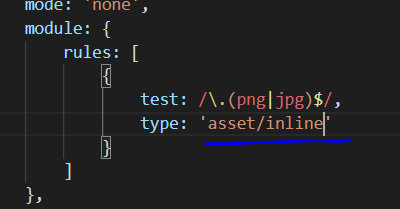
* **publicPath**: ‘dist/’

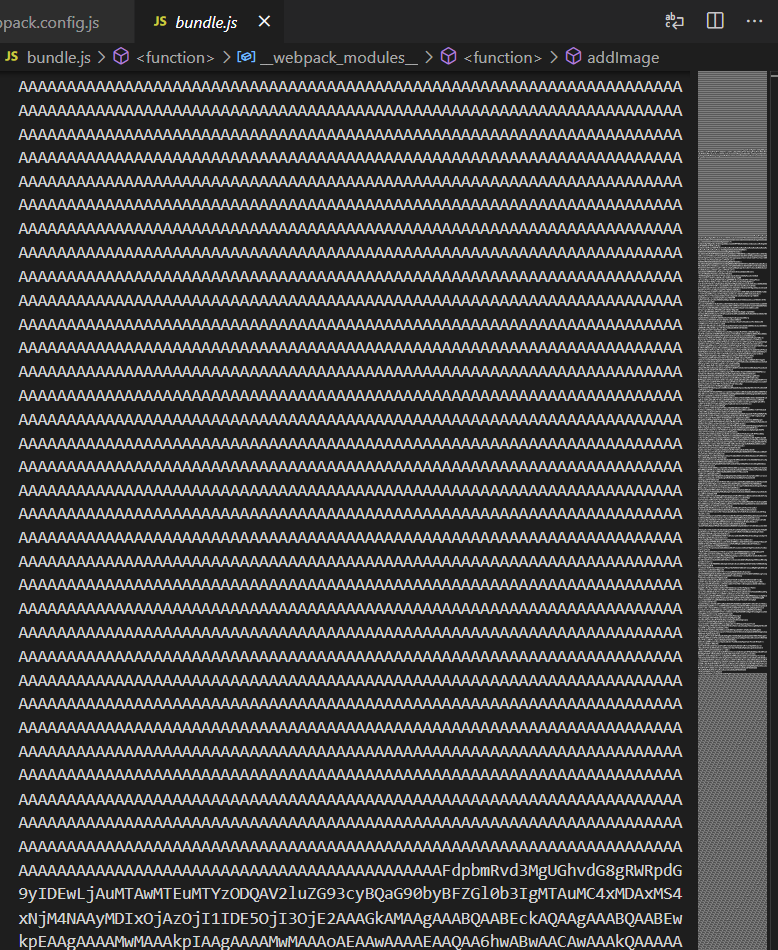




**Asset/inline**

* Does not make a different file in the output
* it compiles the images to base64 and includes them directly to the bundle.js file.
* Increases the size of bundle.js

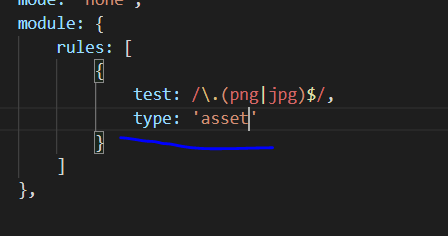


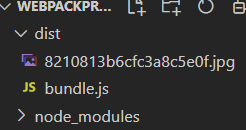


So, it is used only for small files like svgs and when we have a lot of files to import

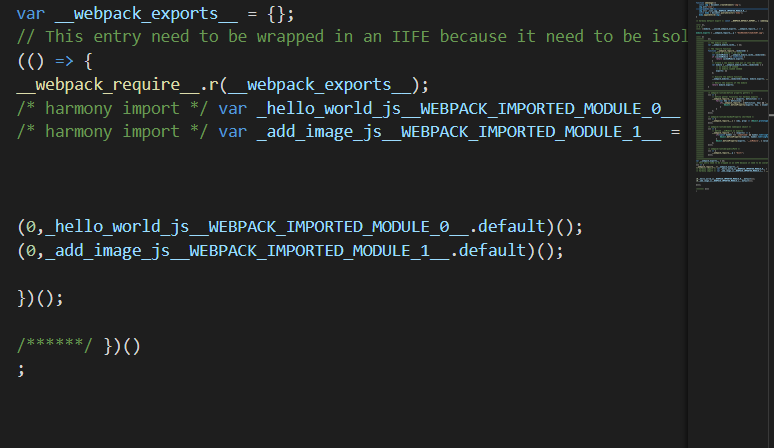
**General asset type**

* Combination of asset/resource and inline.
* Webpack chooses one of them based on the size of the file (by default 8 kb).
* We can also change this default size.



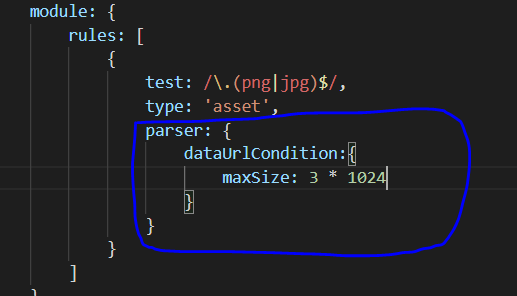


No base64 as webpack chooses the **asset / resource**



To change the default size we can mention one more property in the rule:- **parser**

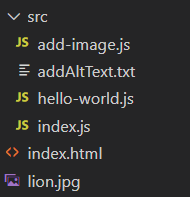
It has only one property: **dataUrlCondition**



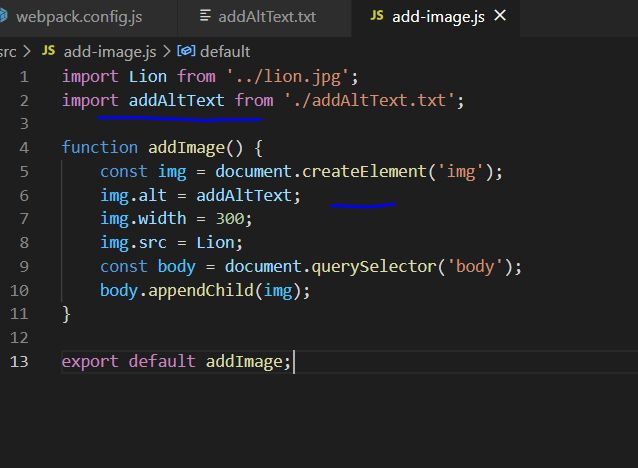
* npm run build.
* Here we change it to 3kbs.
* So, if the file size is less than 3kbs, it will be treated as inline else resource.

**Asset/source type**

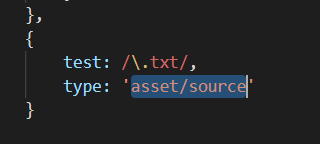
* Create a new text (.txt) file in ‘src’ folder and add a text there



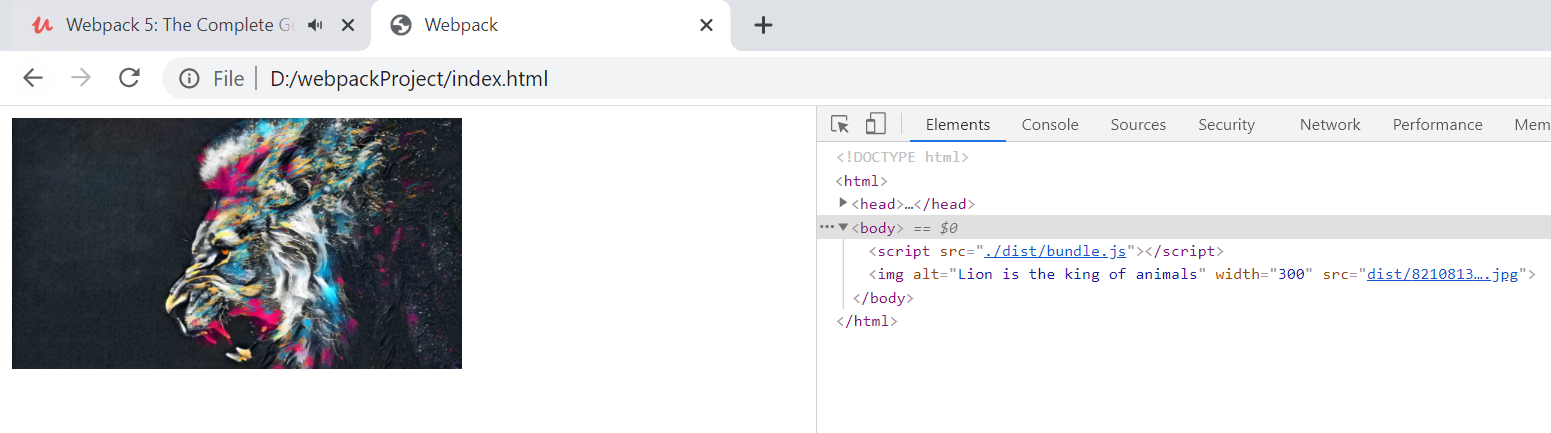
* Import this to the **add-image.js** file and use the text as image **alt**.



* Add the rule in webpack config file



* npm run build

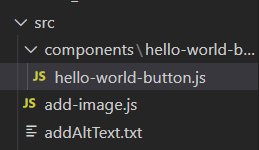


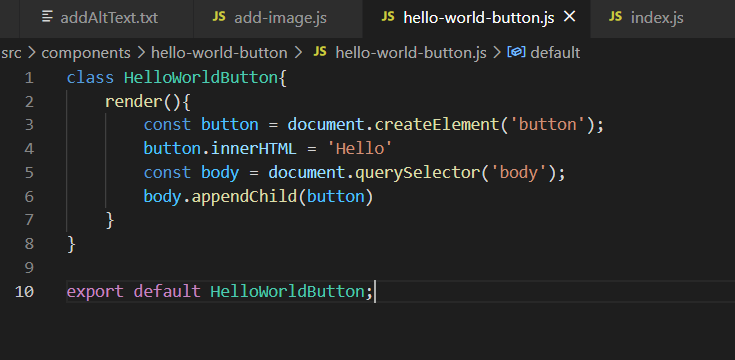
**Loaders**

Loaders allow to import all the kind of files we cannot import in asset modules like sass, css etc.

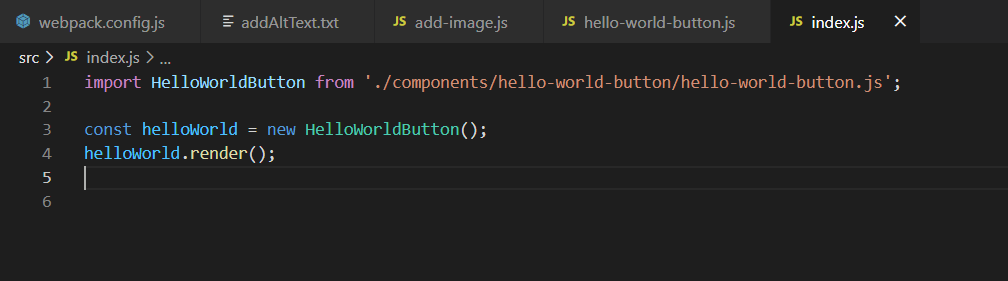
It takes property **use**  in the rules whereas asset modules take **type .**

**Handling css with Webpack**



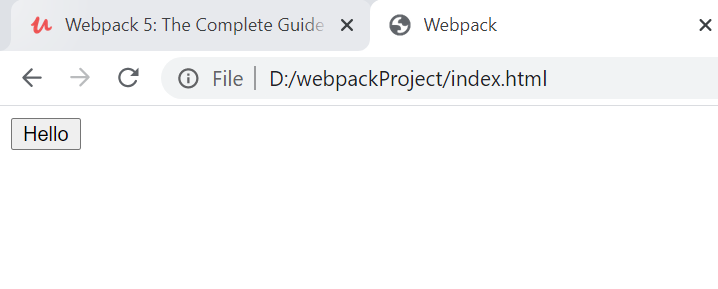


**Index.js**

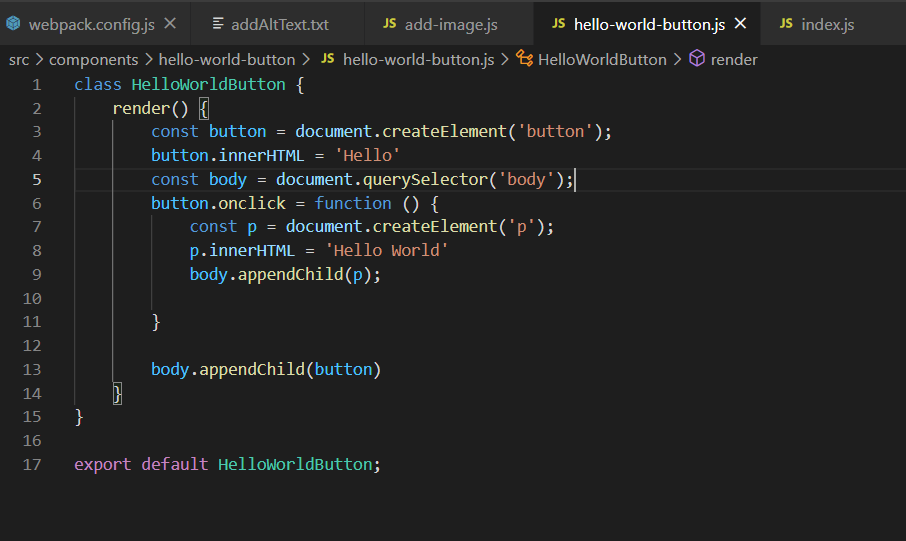


Output on

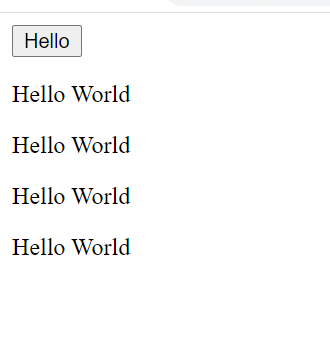
* npm run build



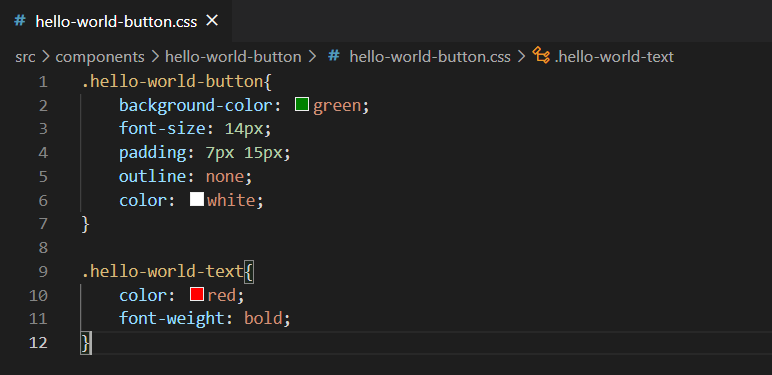
* Add onclick on the button.
* It adds a ‘p’ tag to the body on every click



* On each click on the button



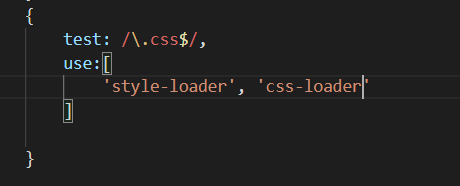
* Let’s add a CSS file to the hello world button folder



* Use the created CSS file in the JS and add the classes to the respective elements



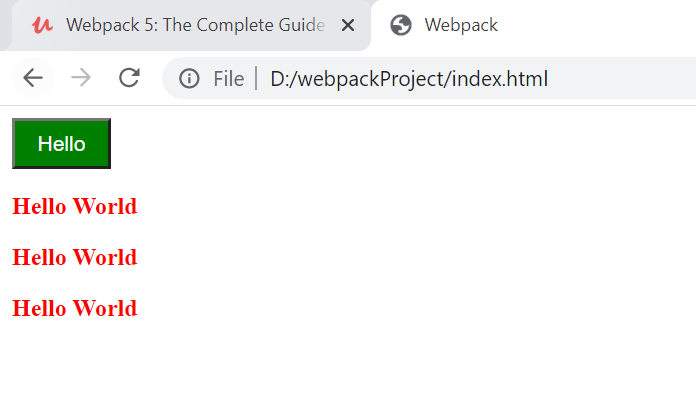
* Create a rule in webpack.config.js



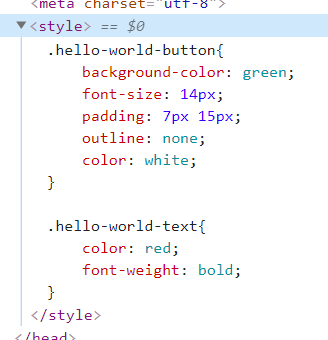
* In this we use ‘use’ property which is an array and takes as many loaders.
* We specify the loaders we need to use the CSS and then npm install them



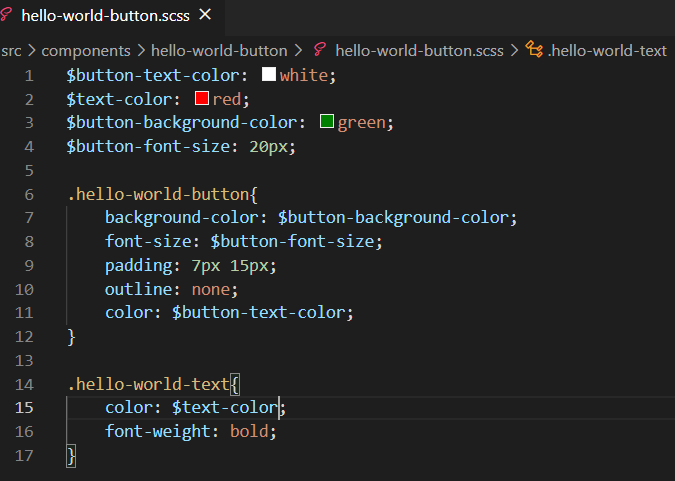
* npm run build

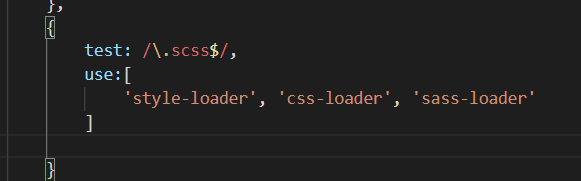


* And 'style' is added in the DOM.



**Handling SASS**

1. Modify the CSS. or add a new sass file and add variables  
   
2. Change the imports in the JS file.  
     
   
3. Add the rule to handle sass => add sass-loader



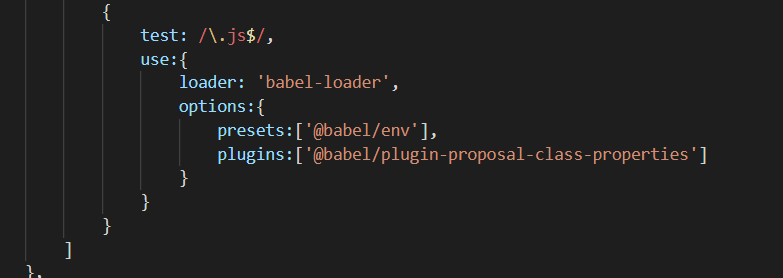
1. npm install sass-loader
2. npm run build

**Using latest JS features using babel**

* Add a class property(buttonClass) in JS class and reference it.



* Add the rule for the babel-loader to handle it as many browser don’t identify class properties



* Install the dependencies:-





* npm run build

**Plugins**

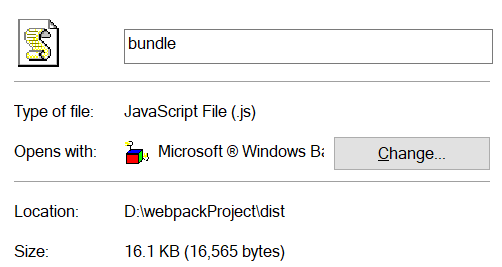
* Plugins are additional JS libraries that can do what loaders cannot..
* They can modify the way bundles are created.

Ex- uglifyJSPlugin decreases the size of the bundle.js by minimizing the content of the bundle.js.

**Minification of the Javascript bundle**

Why do we need to minify the JS bundle- if the application is to open on a mobile, it takes more time related to the desktop. It is better to minify the bundle to improve the load time.

Bundle size before minification



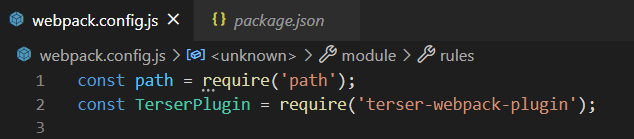
* Add the TerserPlugin in the webpack.config.js
* npm install->



**--save-dev**-> install as dev dependency

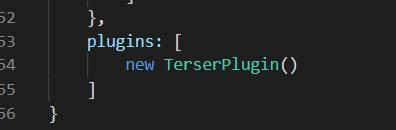
**--save** -> install as production dependency

**Import**->

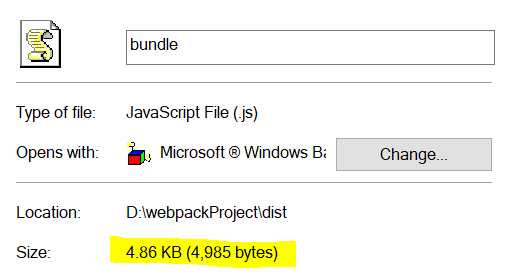


**Use**->

Add after the module ends:-



* **npm run build** and check the size of bundle-



**Extracting CSS to a different bundle**

* Plugin- **mini-css-extract-plugin**

As the CSS is added to the bundle.js in the style tag ,it becomes heavy to load at production time .

So, we want to have a different CSS file in the bundle.JS.

we achieve this by using a plugin called **mini-css-extract-plugin** .

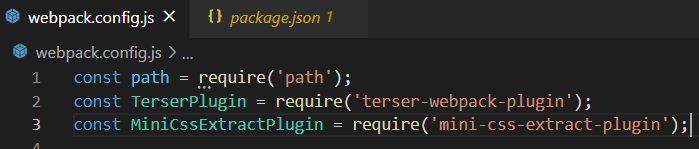
This way we have 2 bundles in the dist folder - one for the CSS and second one for the JS.

this allows us to load different files in parallel.

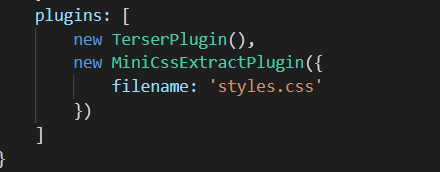
* npm install ->



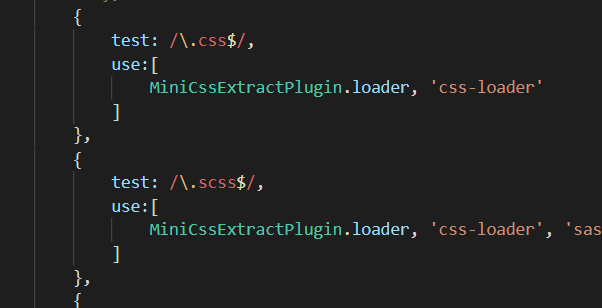
* **Import**->



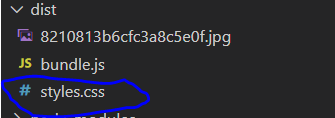
* **Use**->



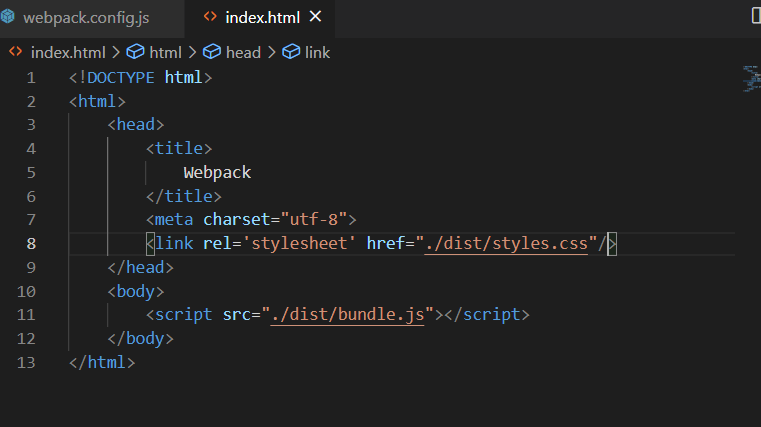
* Change the loader rules both in CSS and SCSS :-



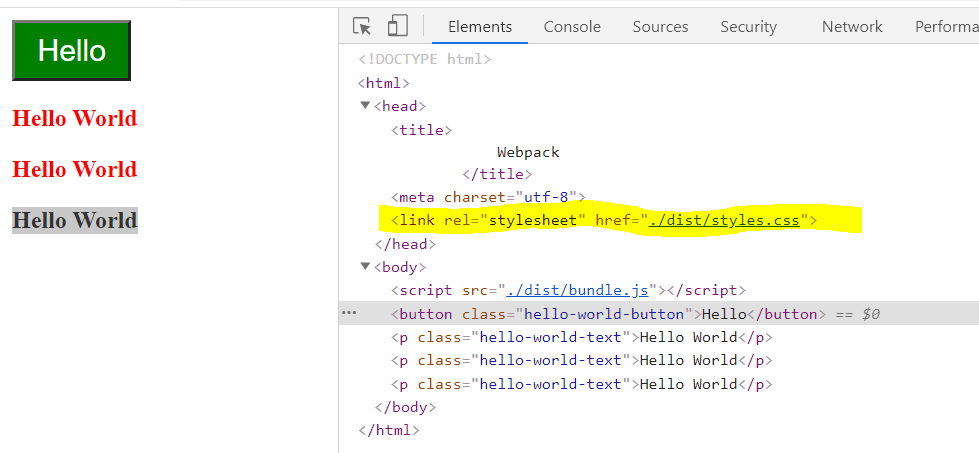
* npm run build-



* Use in index.html



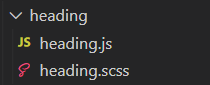
As you can see in the browser that the <style> tag is removed, and the file is loaded through link

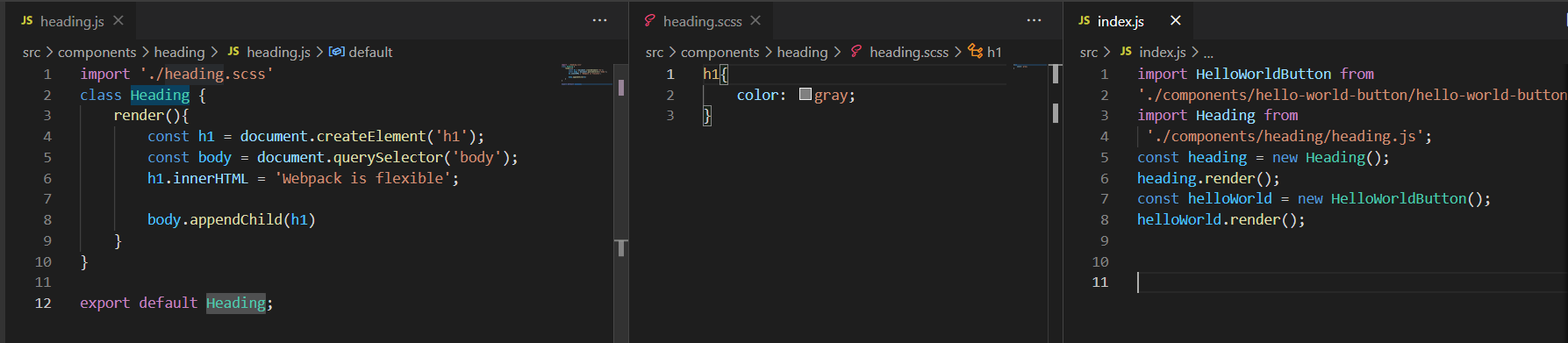


And it works fine.

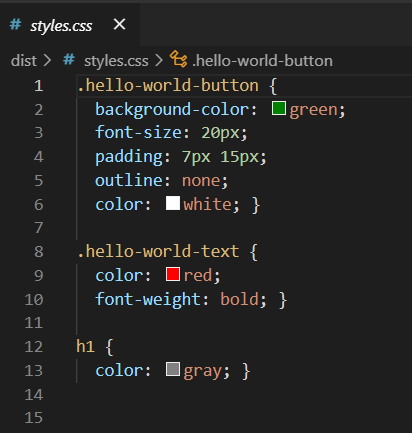
Right now, we have one component. let's see how it works with 2 components: -

**create a heading component** –





* npm run build => styles.css in dist



CSS from all the folders come to the styles.css

**Browser Caching**

Some websites load a lot of JS files whereas some websites load a lot of CSS files. The browser must load all these files before showing the website. So, every time the website is loaded the browsers must download these dependencies or the files .this takes a lot of time.

* the solution to this is browser caching.

The browser stores all the files that it needs while opening the website in its cache, so that it does not require to download these files every time the website is loaded.  
But what will happen when a file is changed or modified or deleted then also the browser will take the file in its cache.

Browsers remember the file by their **name**.

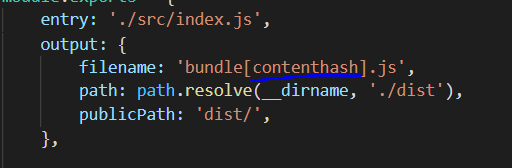
Whenever file is modified the browser will pick the same file again and again.

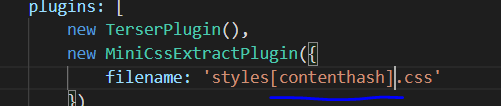
There should be a mechanism to check if the file is modified or not.

We can do this by changing the file’s name every time there is a modification in it.

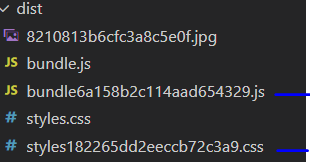
**We can do it with webpack plugin that attaches a hash-content with the file name and every time the file is changed, it updates the hash of the file and hence the browser downloads the new file again.**

You can do so by changing the output file’s name by adding [contenthash ] in it and same with the CSS file.



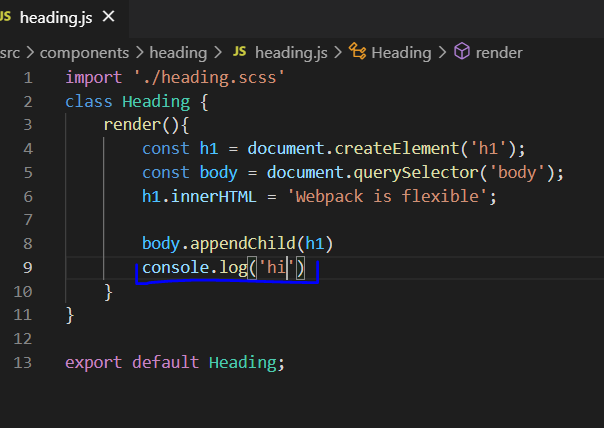


* npm run build without doing any change:-

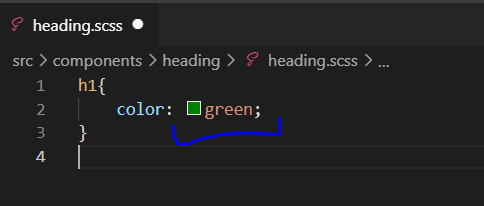


New files are created for the bundle.js and styles.css with hash-code in them.

* Adding change in JS.



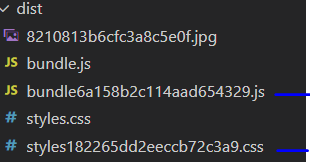
Change in CSS:-



**How to clean dist folder before generating new bundles**

Webpack generates new files in the dist folder every time we change any file. But it does not remove the older files. In this case there are a lot of files in the dist folder. So we make use of plugin to clean the dist folder before generating new files .

Before cleaning files in dist folder:-



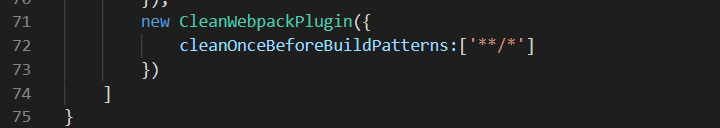
Plugin- **clean-webpack-plugin**

Install-> 

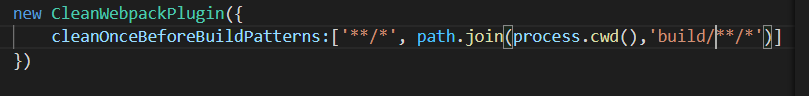
Import ->



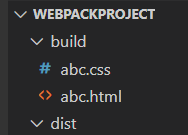
Use->



If you want to clean other folders as well you can share their path here :-



Here **process.cwd()** returns the current working directory and it is joined with the path of files in build folder.



where build is a folder manually created to show the cleaning of an extra folder that is at the same level of dist.

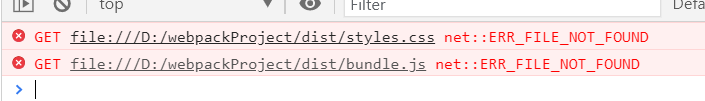
* npm run build:-

It cleaned all the files in build folder and the extra bundles created in the dist folder before the next output is created.

**Generating HTML files automatically during Webpack build**

Now as you go in the index.html file you will see that references to the files in the dist folder are yet not changed.

This will result in breaking of the application.



You can see that the file references are not same, as the name of the files in dist folder are not automatically updated every time the file name changes on content changes by webpack.

We will make use of plugin

**HtmlWebpackPlugin**

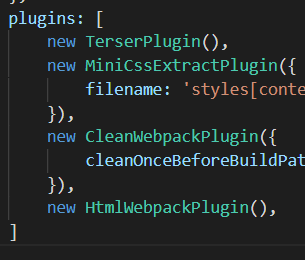
Import ->



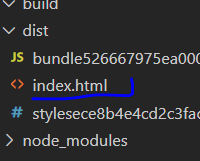
Install->



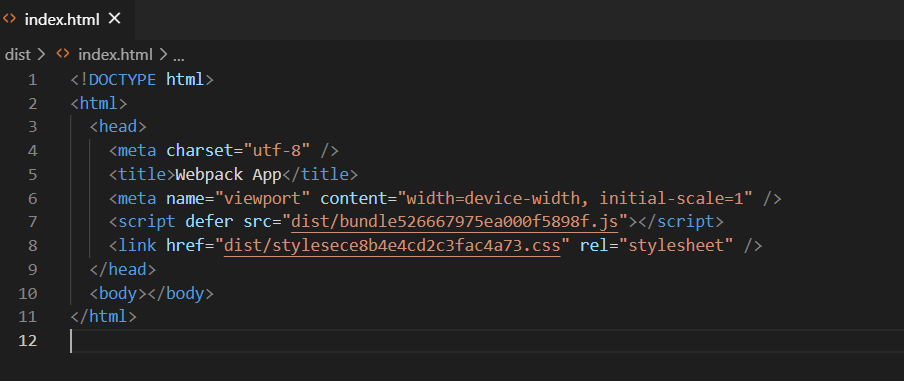
Use->



Npm run build:-

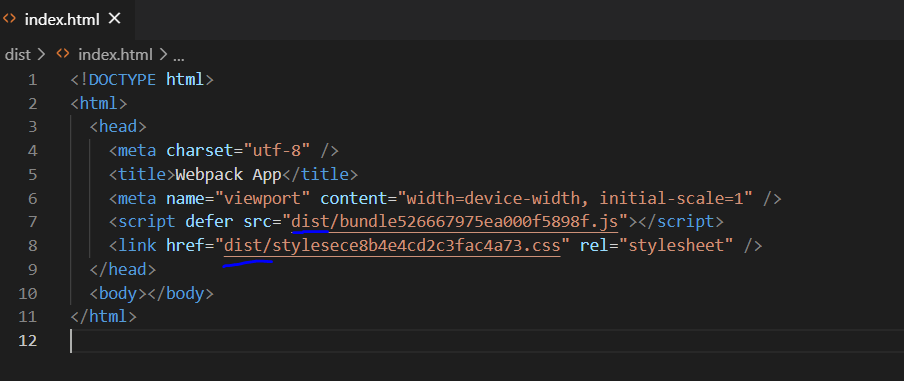


It creates a new index.html file in the dist folder that looks like: -

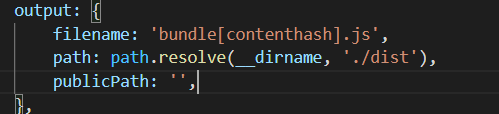


The index.html file created now is itself inside the dist folder.

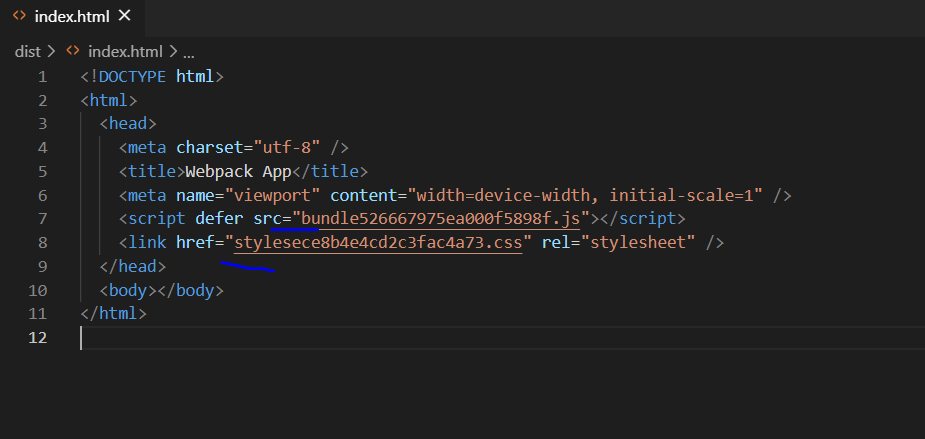
So, we don’t need to refer in the path the dist folder



So, we can remove it from the publicPath

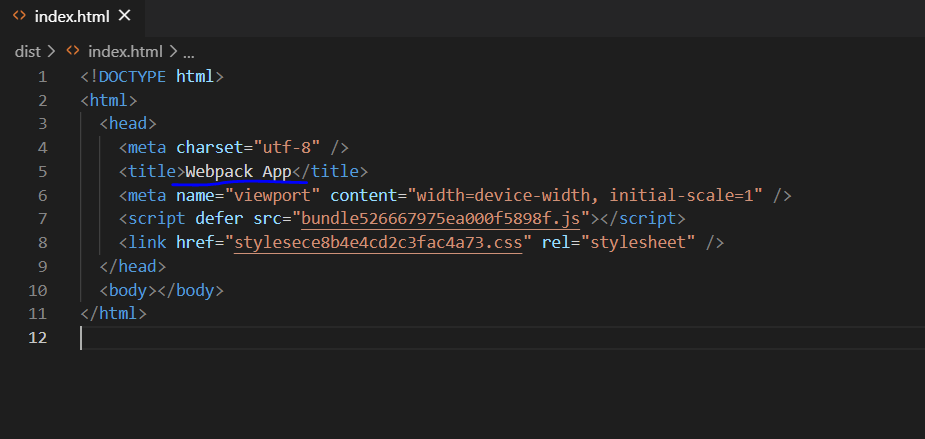


* npm run build

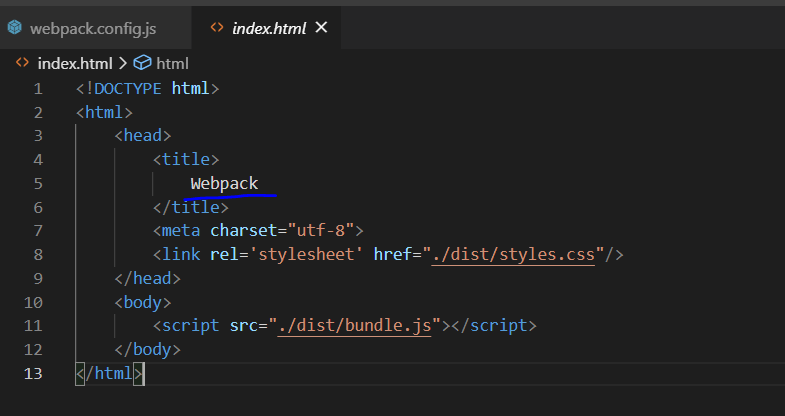


The ref to dist folder is removed.

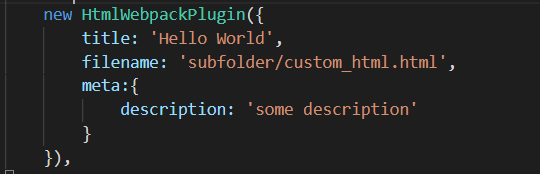
**Customising Generated HTML Files**



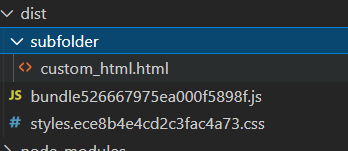
If you notice, the title of this index.html is different from ours.



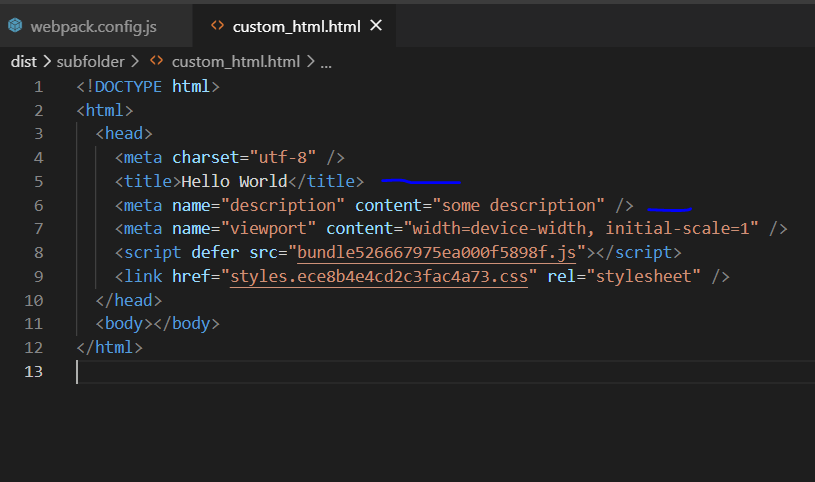
You can add custom data in the HTML Webpack Plugin



* npm run build



It creates the html file with a custom name and custom path

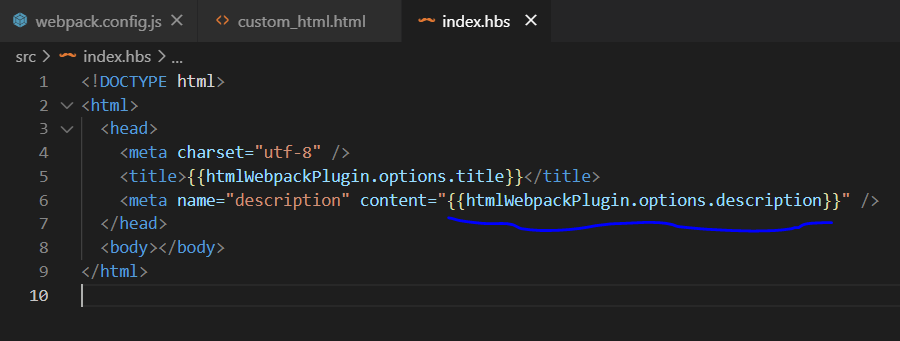


Also customized data meta and title.

**Integration with handlebars**

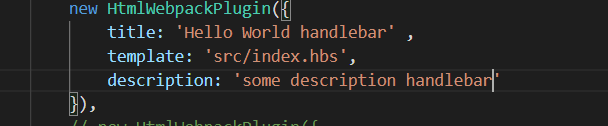
We can also use handlebars to create the index file

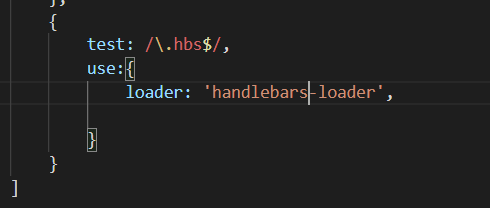
Create a new file index.hbs and remove the references as webpack will automatically add them.



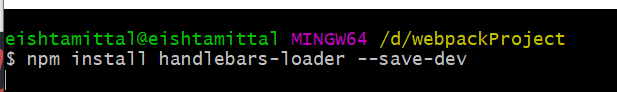
As we load a new type of file, we need to teach webpack how to handle it.

Add the description and title at the same level as template to be used as options.





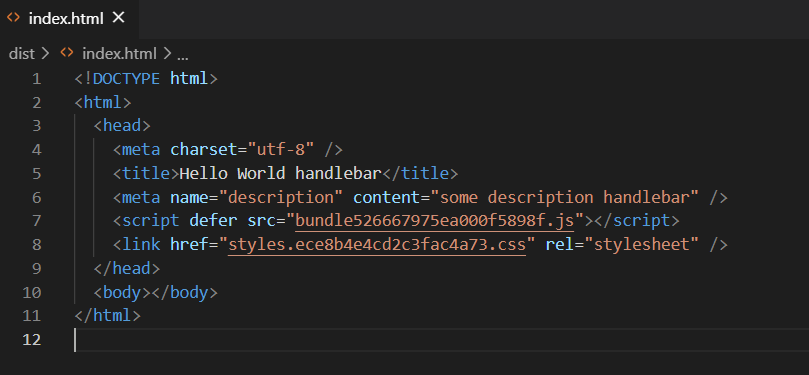
* Install the loader



* Install handlebars package also



* npm run build



You can see the changes in the title and the description

**Production VS Development**

**Intro**

We use mode option to specify the mode of development either development or production or none

**Mode**

**Mode: none** = means no preset configurations required of any environment.

**Mode: production**= configure the webpack file according to the production requirements.

Set process.env.NODE\_ENV on DefinePlugin to production

**Mode: development**= = configure the webpack file according to the development requirements

Set process.env.NODE\_ENV on DefinePlugin to development

**Managing webpack config for production and development configurations**

Split the webpack file into two:

Webpack.production.config.js and webpack.dev.config.js  
copy the content of the webpack.config file to these 2 files.

set the modes respectively   
  
**Production**: -

mode: production

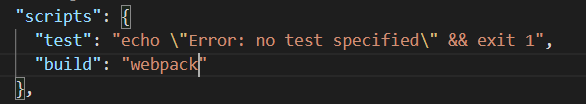
remove **TerserPlugin** as it is included by default here  
  
**Development**: -

* mode: development
* remove the contenthash from the files as we don’t need browser cashing in dev mode
* code minification is extra step which is required in production to reduce the load time and improve customer experience so remove Terser and minicssextract and use style-loader instead.

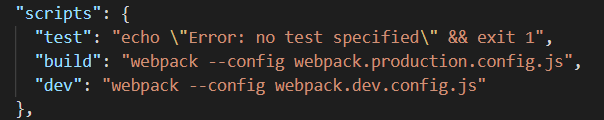
now how to decide which file to run.

npm scripts decide that: -

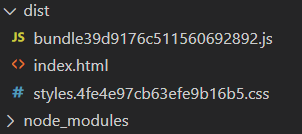
right now we have only 1 script in package.json:-



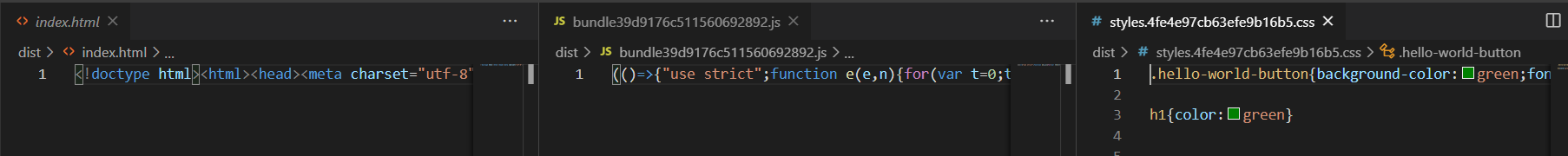
change it to :-



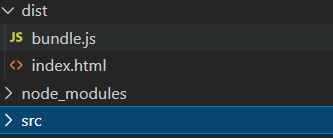
**Poduction**: npm run build



These files are minified.

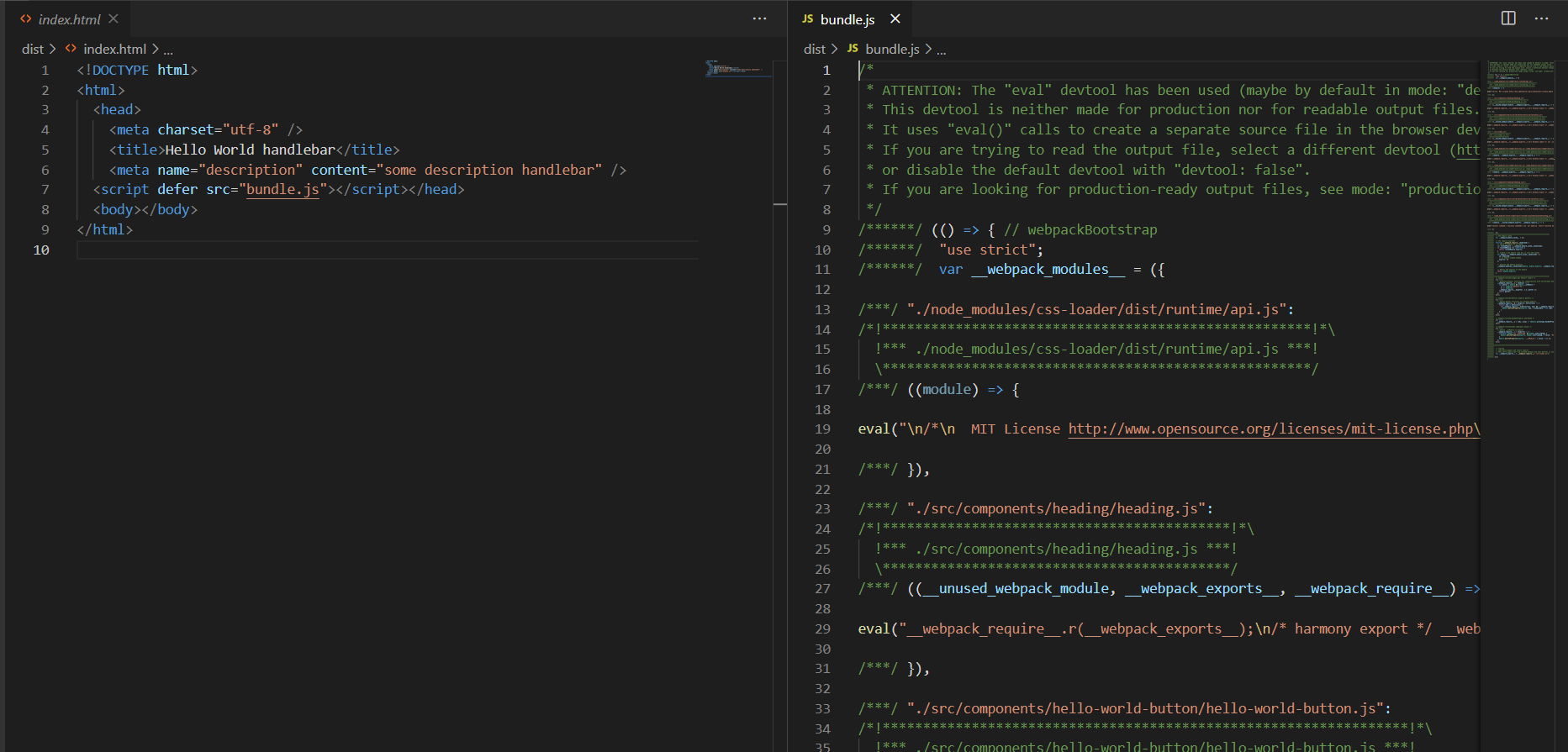


**Development**: npm run dev



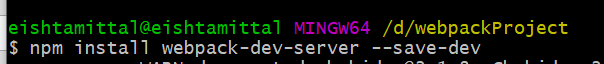
-> no hascontent in name

* Files are not minified

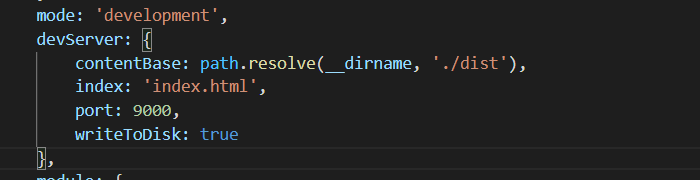


**Faster development with webpack dev server**

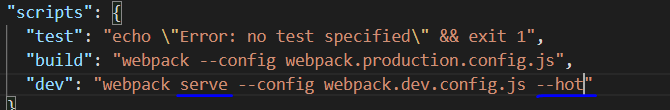
Npm install webpack-dev-server



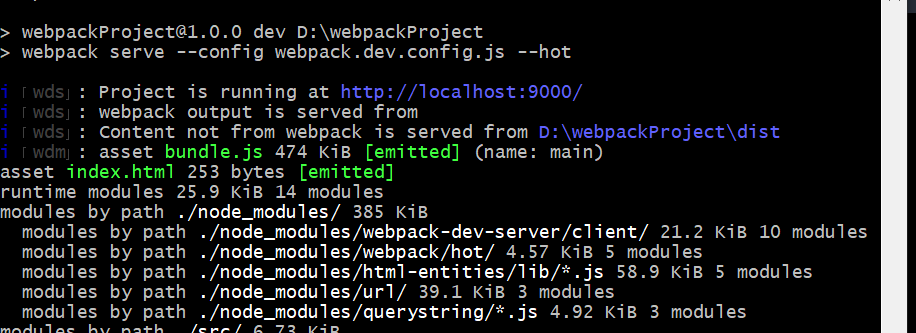
In the webpack.dev.config.js



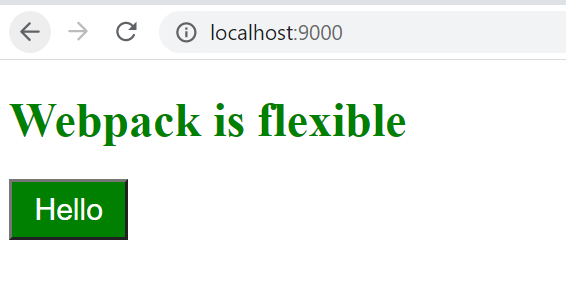
Change the dev script



Npm run dev



Now the application is served at <http://localhost:9000/>



Now no need to build again & again after the changes .  
every change done in the files is reflected to the DOM instantly without rebuild.

**Section 7: MultiPage Application**

**Introduction**

**Creating KiwiImage Component**

**Code splitting in webpack Multi Js and CSS bundles**

**How to generate multiple HTML files**

**Extracting common dependencies while code splitting**

**Setting custom options for code splitting**

**How to setup Development environment for multiple page application**

**Section 8: Github Repository**

**How to setup Github repository**

**Section:9 Webpack integration with Node and Express**

**Introduction**

**Getting code for single page Application**

**Integrating Express into our application**

**Serving HTML pages via Express  
Handling JS and CSS via Express**

**Getting Code for Multiple page application**

**Integrating Express into Multi page Application**