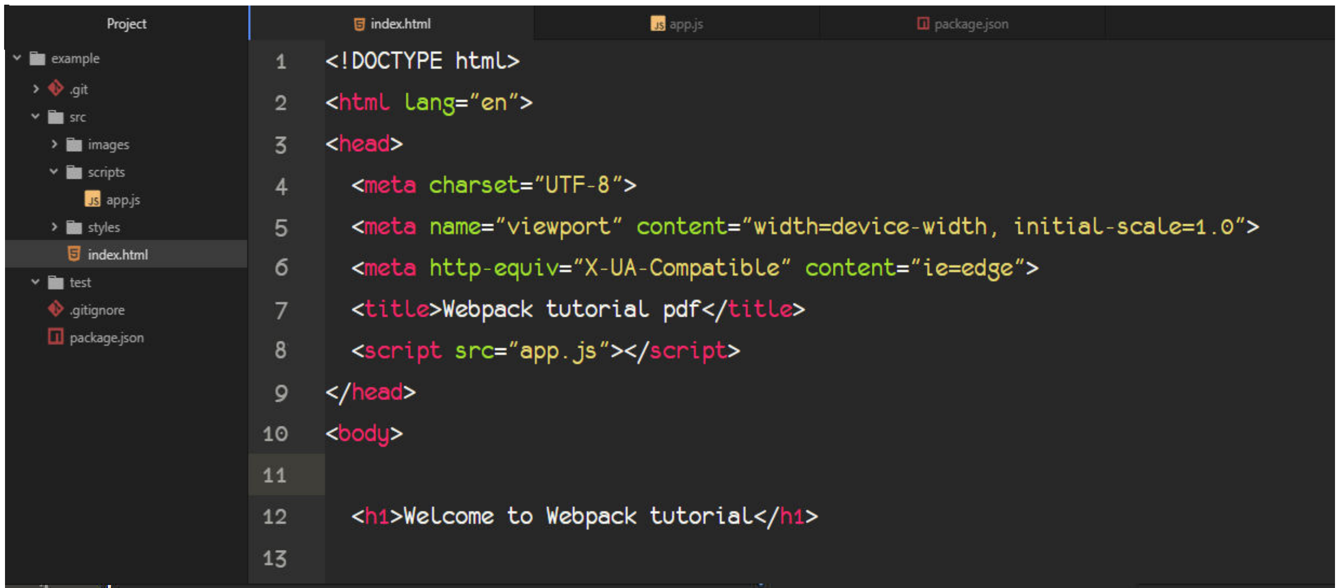
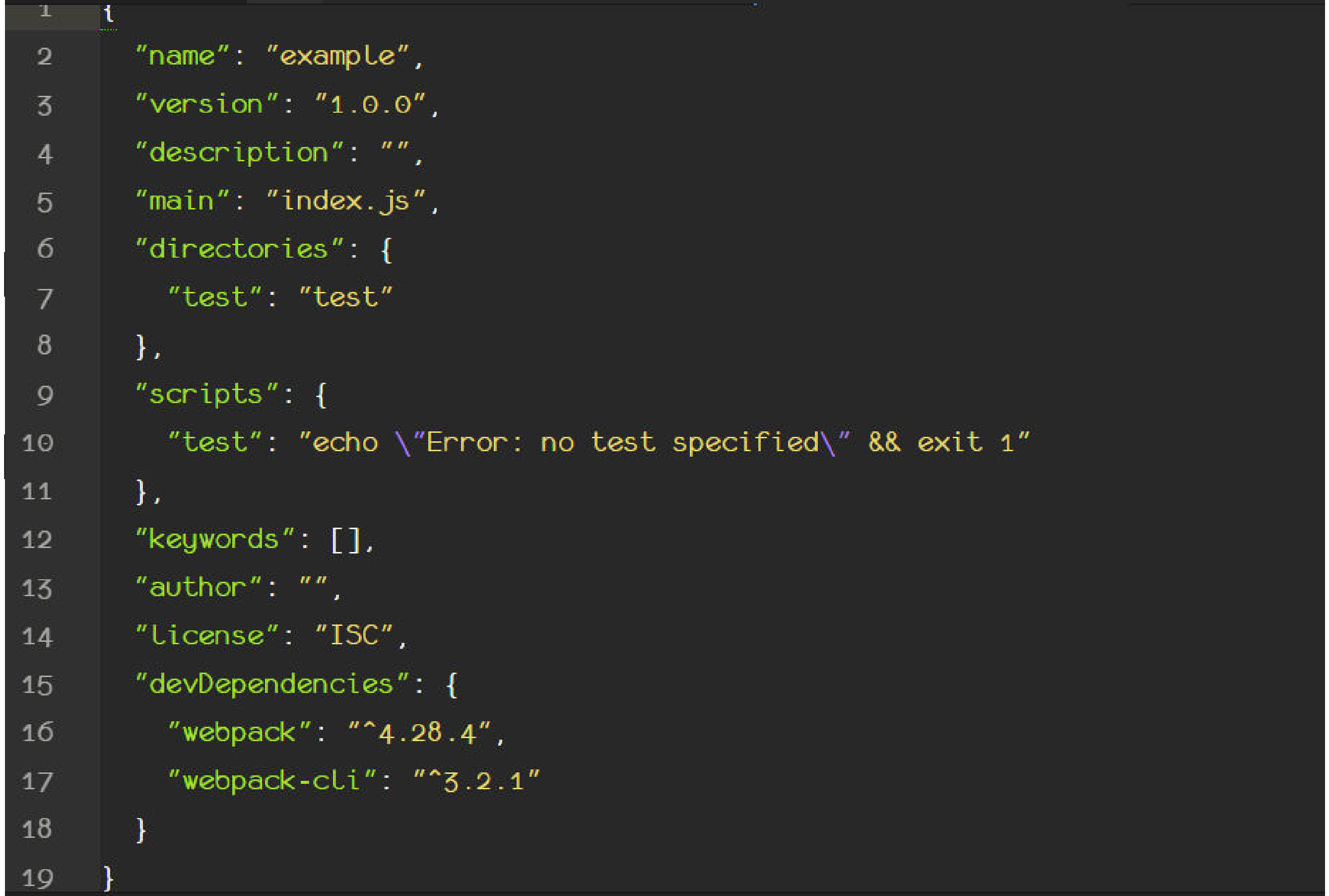
Webpack Tutorial Guide step by step

First of all setup your project folder with the following structure. Create an src folder on root folder of your project. Inside the src have your index.html, and the folder scripts, styles and images. Inside the scripts create an app.js script and connect it to your html file on head section.

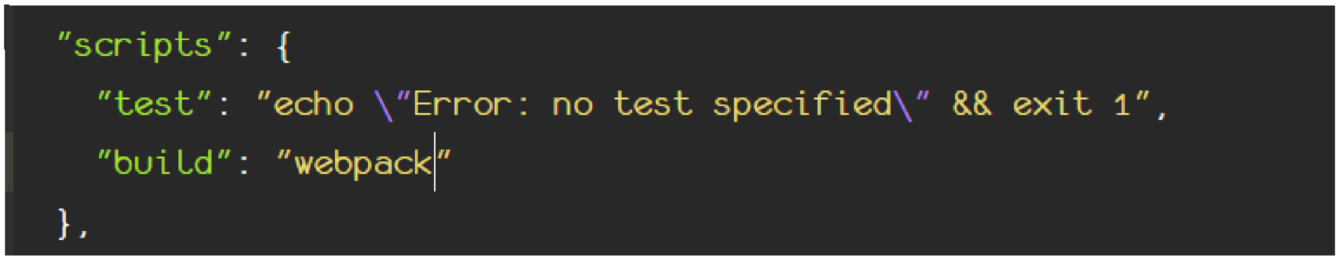
Then hit npm init -y on root folder always. (add .gitignore assuming you are going to have git and put *node_modules, images* etc.)



```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Webpack tutorial pdf</title>
  <script src="app.js"></script>
</head>
<body>

  <h1>Welcome to Webpack tutorial</h1>

```



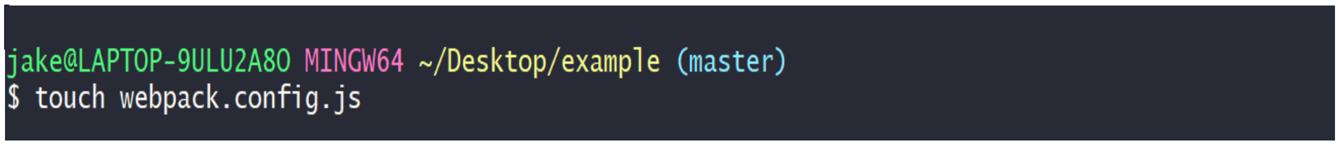
```json
{
  "name": "example",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^4.28.4",
    "webpack-cli": "^3.2.1"
  }
}
```



```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack"
},
```

Now, we must configure the webpack’s behavior and instruct what it should do when we hit ‘npm run build’.

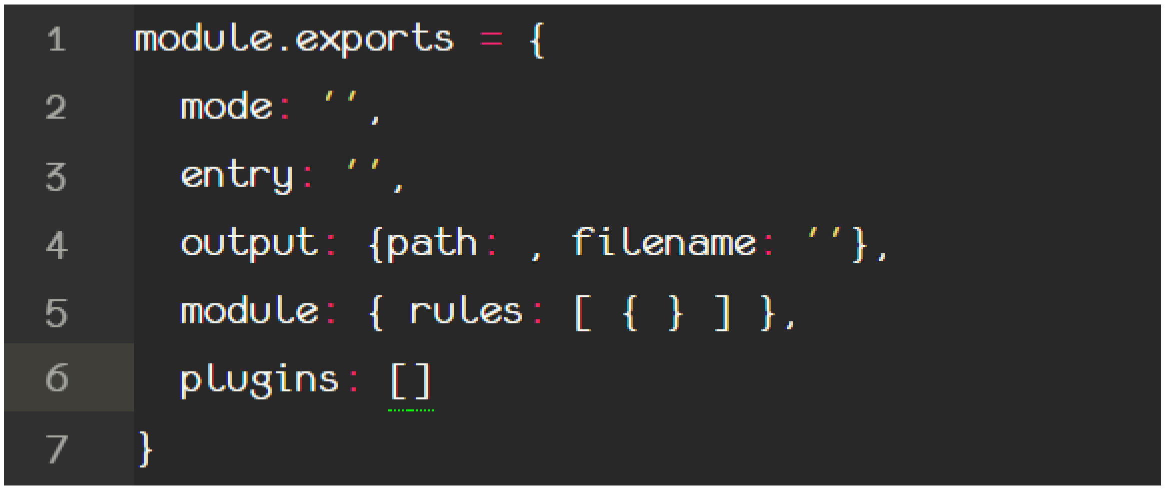
Create at the root folder of your project, a webpack.config.js file.



```
jake@LAPTOP-9ULU2A80 MINGW64 ~/Desktop/example (master)
$ touch webpack.config.js
```

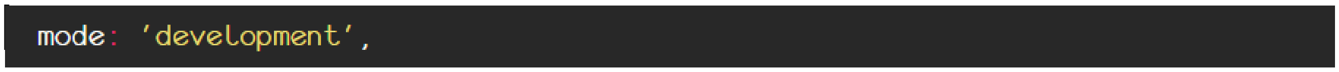
This is where you can define all the properties of webpack.

Bear in mind that webpack has a specific schema that must be put in the module.exports object. The abstracted structure of this schema can be shown below. Bear in mind that the schema must be like that and the properties are no free to vary or to change. Just follow the dogma!!!



```
module.exports = {
  mode: '',
  entry: '',
  output: {path: , filename: ''},
  module: { rules: [ { } ] },
  plugins: []
}
```

The first need we configure is the mode. You can set this either to ‘development’ or to ‘production’. If set to ‘production’ then the bundle.js code (the output source code will be minified and compressed, ready for shipping to production). For now we will set this to development mode.



```
mode: 'development',
```

file, that inside of it we can put all the code that webpack can start work with, translate, extract or analyze. This is the only source of truth, and of course it is the one that is connected with your html file. Specifiy a relative path, starting from where the webpack.config.js file is.

```
// The entry file for extracting everything
entry: './src/scripts/app.js',
```

Then we need to configure the output folder and the file. The destination-result. The output is an object that contains 2 properties. The destination folder and the name of the extracted file. To define the destination folder we need to specify an ABSOLUTE path to it. Thus we can use the path module that node.js provides. To do that we must include the path module at the top of our webpack.config.js file.

```
const path = require('path');
```

Then for the output we can use the path.resolve function in order to create an absolute path from the root folder of your OS to the current folder with the __dirname (the current folder) and a second argument, which is going to be the folder that webpack is going to create for us. In the filename we specifiy the name of the extracted file, by convention this is usually called bundle.js

```
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'bundle.js'
},
```

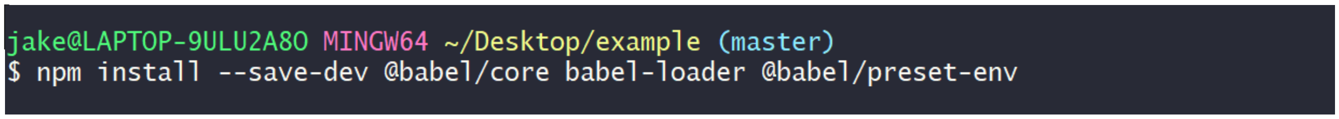
Great we have defined the source of webpack and the destination of results. Now we need to configure our tasks.

This takes place inside the module object. Create a rules array inside this object as a property. Inside the rules you can define objects, every object is a task. For now we want only to translate our code from ES6 to ES5 javascript.

That means that we need 3 things. 3 packages to install.

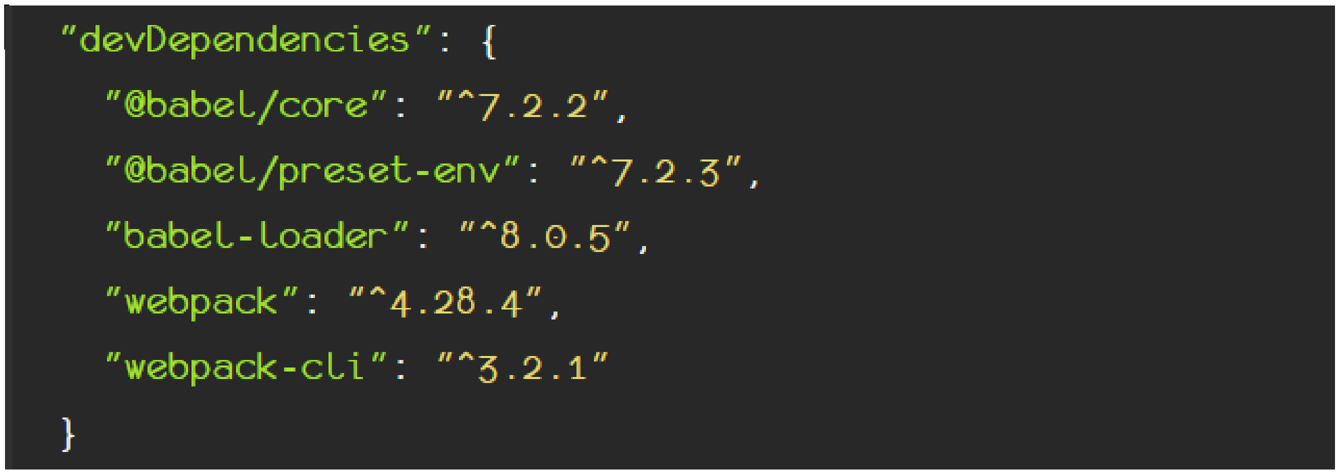
@babel/core: The actual translator that does the job.
babel-loader: this module enables babel inside every file that we define in test property (see below).
@babel/preset-env: To what do we want to translate? JS comes with many flavors (JSX, ES6, typescript, coffeescript etc). The preset-env is for ES6.

Go and install these packages through npm:



```
jake@LAPTOP-9ULU2A8O MINGW64 ~/Desktop/example (master)
$ npm install --save-dev @babel/core babel-loader @babel/preset-env
```

Check that these packages were installed. Your package.json file should look like that:



```
  "devDependencies": {
    "@babel/core": "^7.2.2",
    "@babel/preset-env": "^7.2.3",
    "babel-loader": "^8.0.5",
    "webpack": "^4.28.4",
    "webpack-cli": "^3.2.1"
  }
```

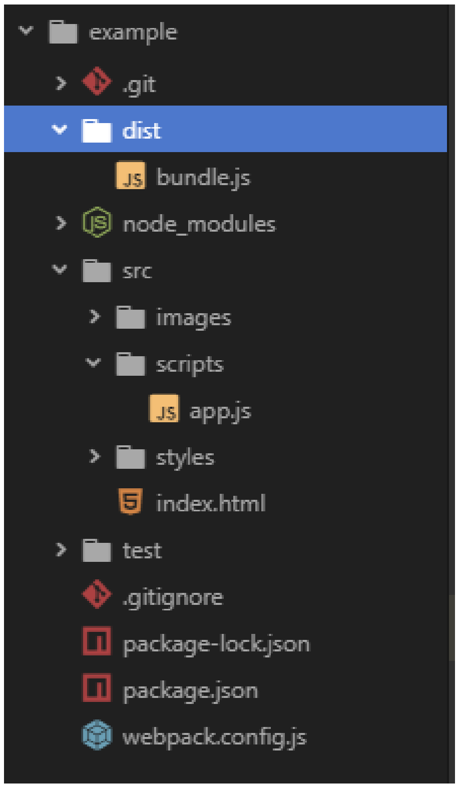
Time to define the task.

First define which types of files you want to be translated. This is done by the test property, that accepts a regular expression. The following code will translate all the files that end with .js but excluding the .js files inside the node_modules folder.

Then inside the use, the actual task is defined. We define the loader, what all the .js files that define   through the test property will have inside them loaded. Without that, babel has no access to the .js files. Then inside the options, we set the preset, the js flavor we want our files to be translated to. Both options and loaders are always inside the use object. In the end your complete task is going to look like that.

```
rules: [
  {
    // Transpile from ES6 to ES5 all files that end with .js
    test: /\.js?/i,
    // except the js files inside the node_modules folders
    exclude: /node_modules/i,
    use: {
      loader: 'babel-loader',
      options: {
        presets: ['@babel/preset-env']
      }
    }
  }
]
```

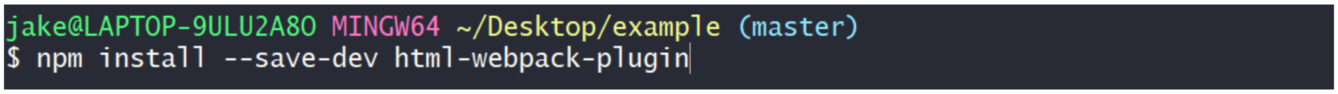
If you can see the dist folder inside your project's folder structure, and the bundle.js inside then everything has gone good! You should have something like that:



```
example
  .git
  dist
    bundle.js
  node_modules
  src
    images
    scripts
      app.js
    styles
    index.html
  test
  .gitignore
  package-lock.json
  package.json
  webpack.config.js
```

That’s good! But what we want to do is that we move all of our src files, into the dist folder. At the end the dist folder will be the one that is going to be served to the user. So let’s try to move the index.html file also there!

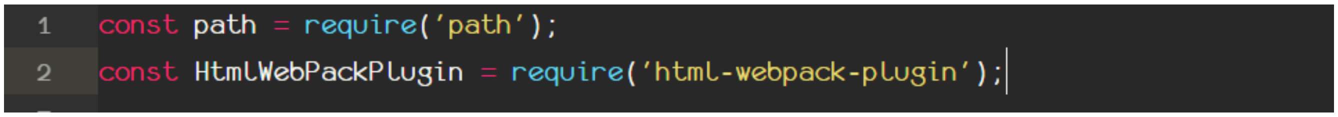
In order to do this we need to download and install a plugin. The html-webpack-plugin. Go and install it to your project.



```
jake@LAPTOP-9ULU2A80 MINGW64 ~/Desktop/example (master)
$ npm install --save-dev html-webpack-plugin
```

Check if installed as always to your package.json file.

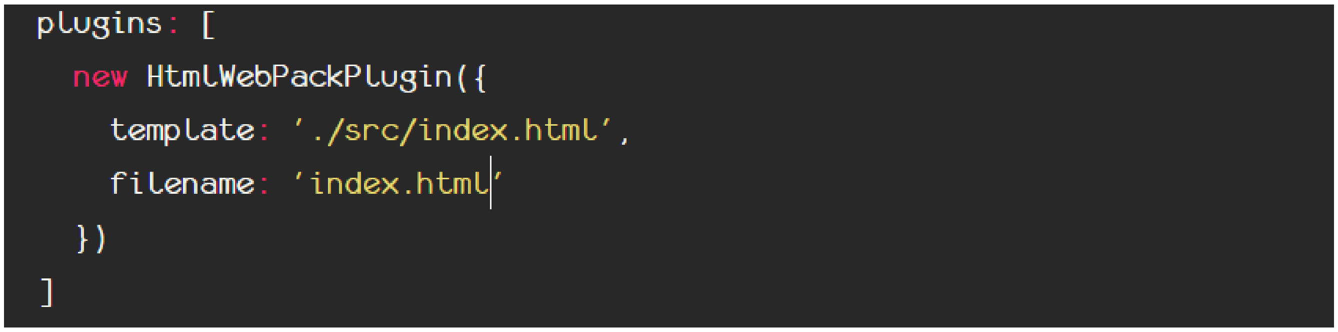
Then include it to your webpack.config.js at the top of your script in order to use it.



```
const path = require('path');
const HtmlWebPackPlugin = require('html-webpack-plugin');
```

This is a webpack plugin for html files of your project. Has nothing to do with .js files and it doesn’t have to be loaded into your js files. That means that there is no loader, and nothing inside the module object of your config file needs to change.

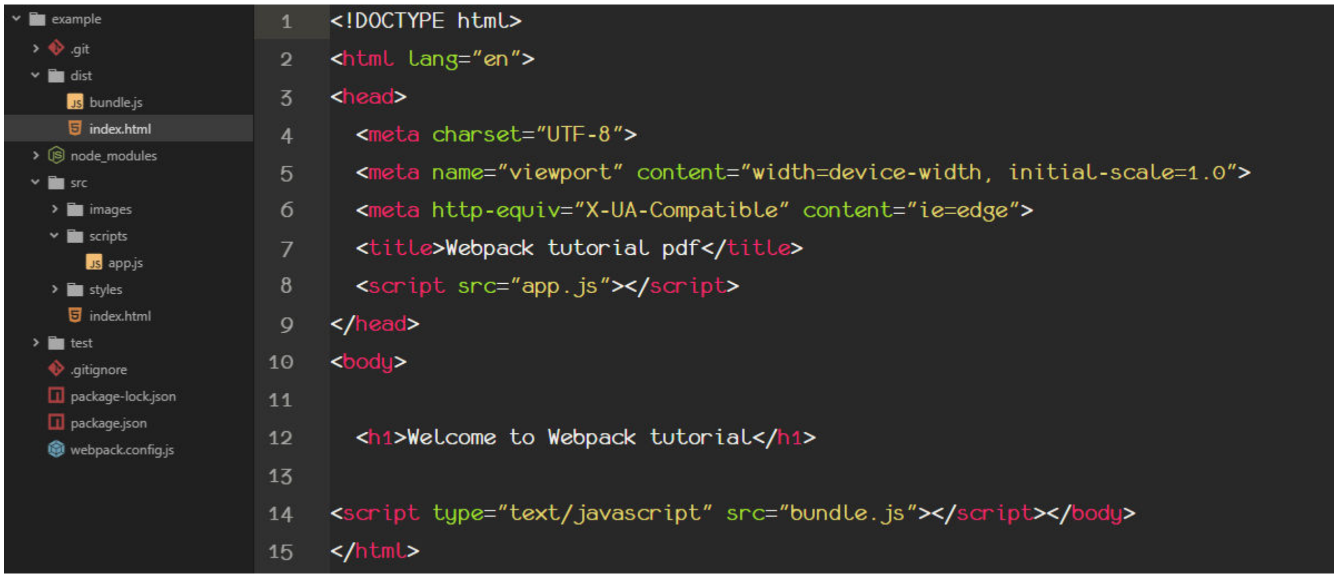
Instead this is a plugin, thus we need to configure it’s behavior by adding it as a plugin object. This object accepts 2 properties, the html file that has to be moved to the dist folder, and the name of the file that will be created there. By convention we keep the same name. The properties, template and filename must stay as they are.



```
plugins: [
  new HtmlWebPackPlugin({
    template: './src/index.html',
    filename: 'index.html'
  })
]
```

Hit npm run build again.

Hopefully what you have is a new index.html inside the ‘/dist’ folder and inside it you can see a connection to the bundle.js is already there! So you can remove the ‘app.js’ from before now. This is the power of webpack!



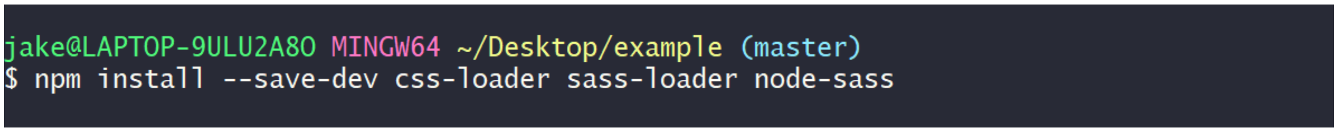
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Webpack tutorial pdf</title>
  <script src="app.js"></script>
</head>
<body>

  <h1>Welcome to Webpack tutorial</h1>

<script type="text/javascript" src="bundle.js"></script></body>
</html>
```

Fine! Let’s add CSS to it!

Deactivate all you scss to css compiler before you proceed!!!

Go ahead and install three packages.

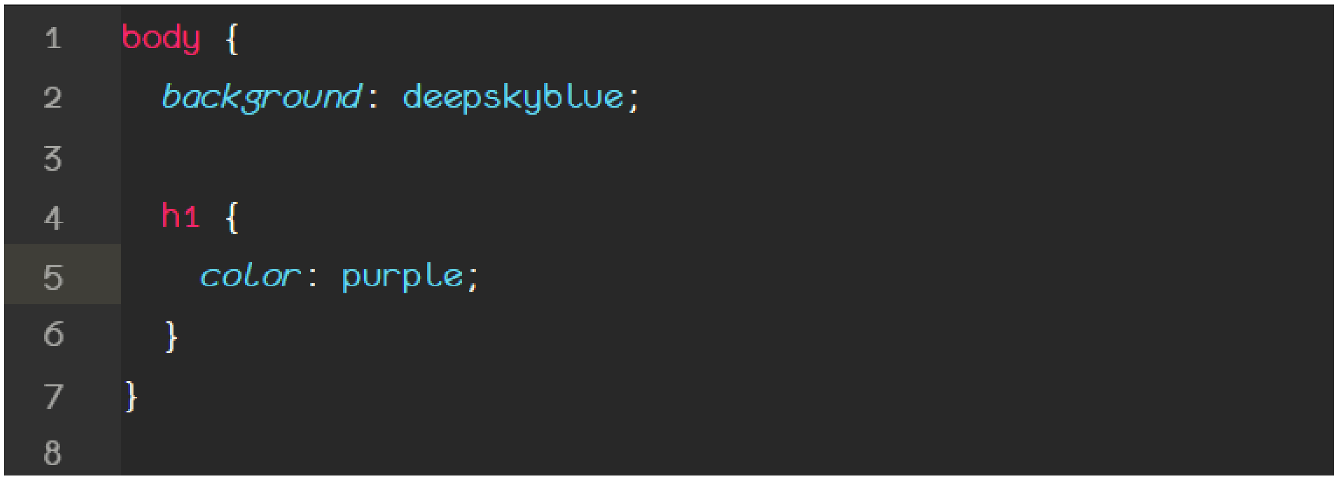
css-loader: this allows you to load css files INSIDE javaScript files!
sass-loader: This allows you to import sass, scss files inside your javaScript files
node-sass: this will transpile you scss to css.



```
jake@LAPTOP-9ULU2A80 MINGW64 ~/Desktop/example (master)
$ npm install --save-dev css-loader sass-loader node-sass
```

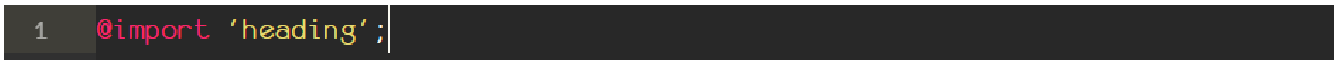
Let’s go inside the src/styles folder and create a main.scss file and a _heading.css partial.

Inside the _heading.css partial write some code regardig the h1 file.



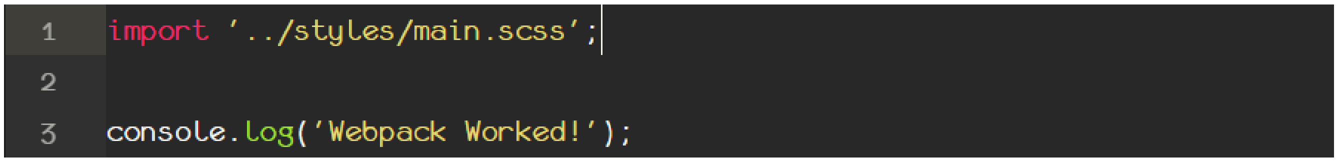
```scss
body {
  background: deepskyblue;

  h1 {
    color: purple;
  }
}

```

Now import this partial to your main.scss file as usual.



```scss
@import 'heading';
```

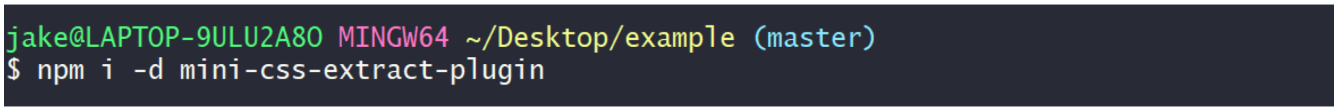
And now let’s import the main.scss file inside the app.js!!



```js
import '../styles/main.scss';

console.log('Webpack Worked!');
```

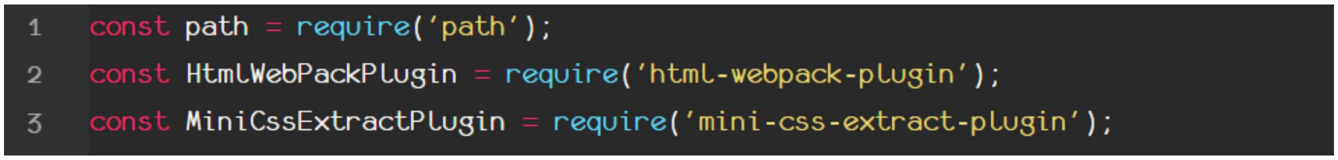
This will be possible from now on, since we are going to configure the installed css-loader and sass-loader packages.

Now, there is also another package left. By default what we have done will load the main.scss file but will NOT create a new file in the dist folder. In order for this to be done, we need a new plugin, named mini-css-extract-plugin. Go to terminal and hit:



```
jake@LAPTOP-9ULU2A80 MINGW64 ~/Desktop/example (master)
$ npm i -d mini-css-extract-plugin
```

Now we need to go to the webpack.config.js file and import this plugin.



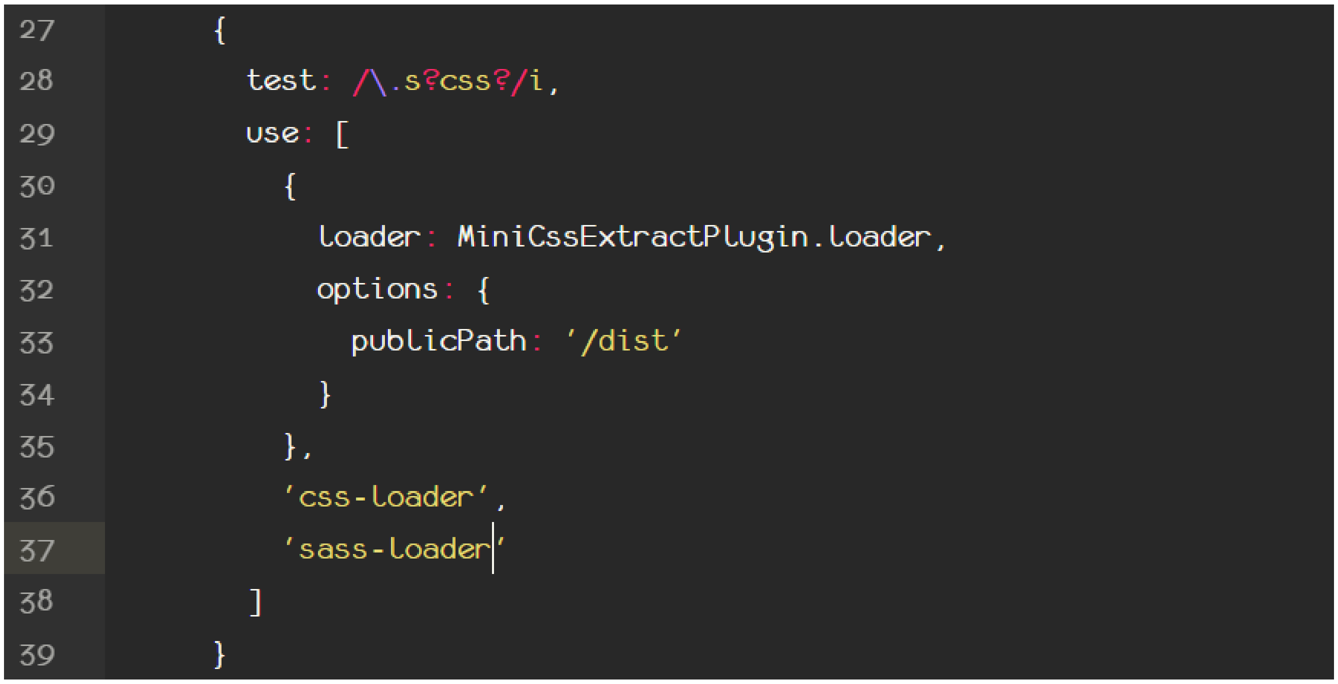
```js
const path = require('path');
const HtmlWebPackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
```

Then it’s time to define our task, that we want to add .css and .scss file into our entry app.js.

Inside the module object, inside the rules array, we create a new object task, and we configure it. We define in the test property all the css or scss file by making the s? Optional at the beginning.

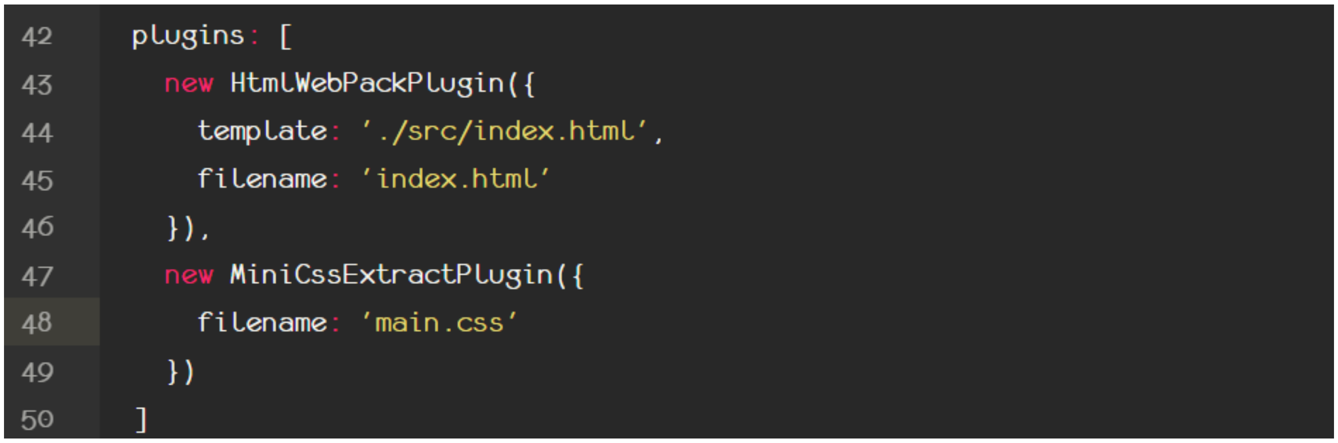
Then inside the use is an array now because we are going to add more than one loaders. The order has to be reverse because webpack executes loaders in reverse order. Thus we need to put sass-loader last, css-loader before and the MiniCssExtractPlugin.loader at first. For the latter we need also to specify the destined folder that the newly generated css file will go.

At the end your new task in rules will look like that!



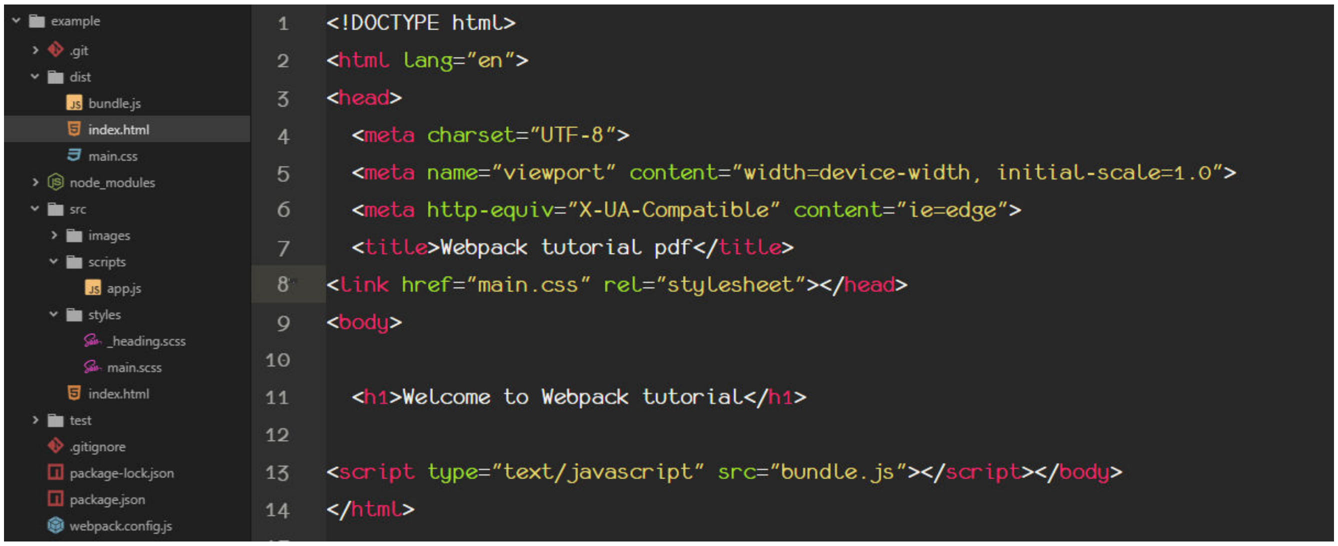
```
      {
        test: /\.s?css?/i,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
            options: {
              publicPath: '/dist'
            }
          },
          'css-loader',
          'sass-loader'
        ]
      }
```

Then the last thing is to specify the filename of the newly created css file. In plugins section we create a new MiniCssExtractPlugin, to do that.



```
plugins: [
  new HtmlWebPackPlugin({
    template: './src/index.html',
    filename: 'index.html'
  }),
  new MiniCssExtractPlugin({
    filename: 'main.css'
  })
]
```

Hit npm run build and hopefully you have something like that:



```
example
  .git
  dist
    bundle.js
    index.html
    main.css
  node_modules
  src
    images
    scripts
      app.js
    styles
      _heading.scss
      main.scss
    index.html
  test
  .gitignore
  package-lock.json
  package.json
  webpack.config.js
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Webpack tutorial pdf</title>
<link href="main.css" rel="stylesheet"></head>
<body>

  <h1>Welcome to Webpack tutorial</h1>

<script type="text/javascript" src="bundle.js"></script></body>
</html>
```

Check that a main.css file was created to your /dist folder and it was automatically added to your index.html file!

Thanks webpack!! :-)

Images Optimization

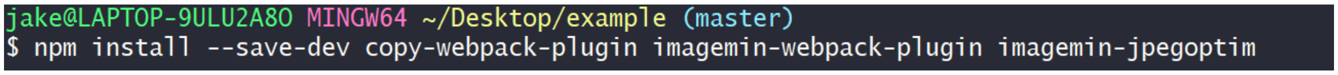
Let's see now how webpack can fix the performance of our application. A big portion of an application's size is taken by images, and due to their large size. We can reduce programmatically their size through webpack.

For the next task we are going to need 3 plugins to install

imagemin-webpack-plugin
copy-webpack-plugin

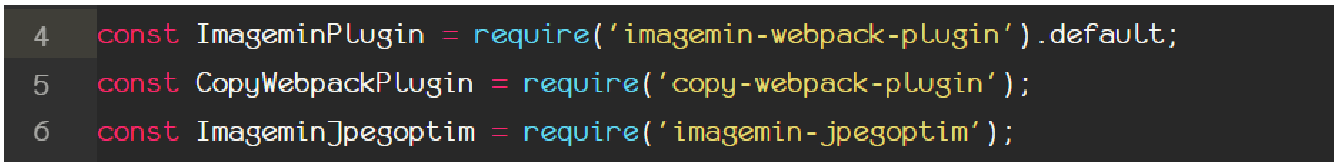
imagemin-jpegoptim

Try and install them



```
jake@LAPTOP-9ULU2A80 MINGW64 ~/Desktop/example (master)
$ npm install --save-dev copy-webpack-plugin imagemin-webpack-plugin imagemin-jpegoptim
```

Then we must include them on our project through the require method.

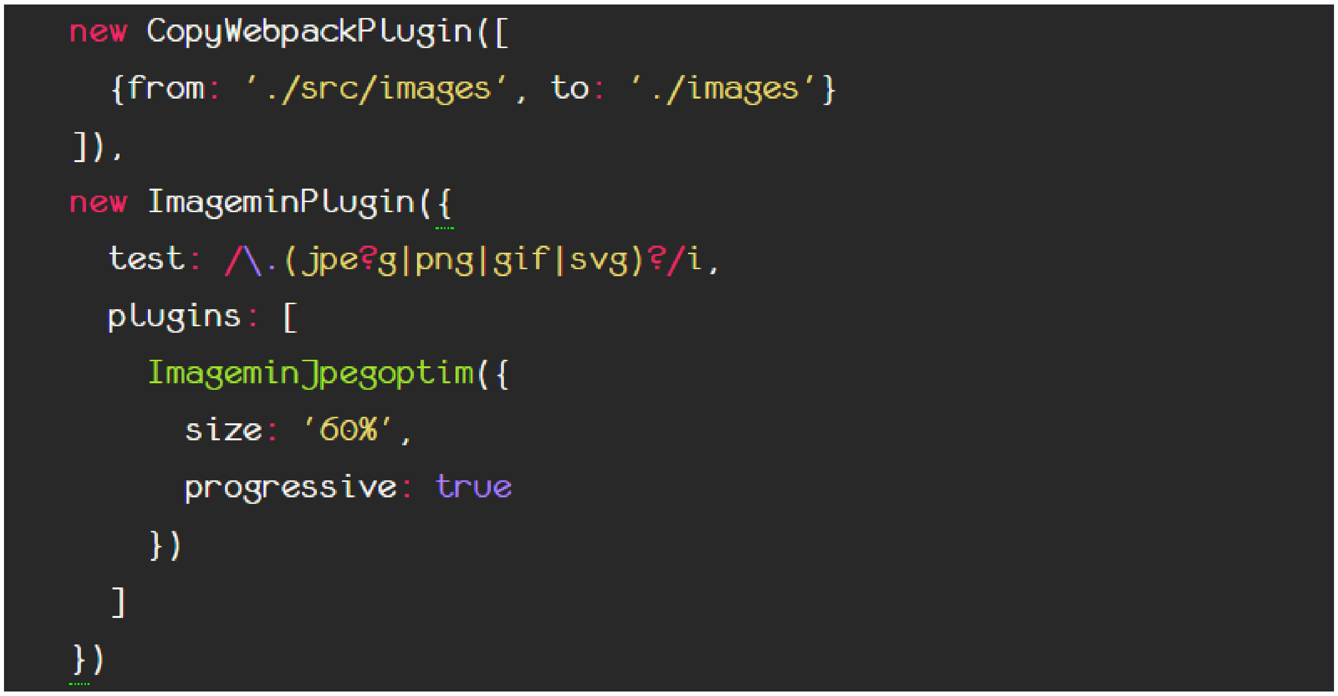


```
4   const ImageminPlugin = require('imagemin-webpack-plugin').default;
5   const CopyWebpackPlugin = require('copy-webpack-plugin');
6   const ImageminJpegoptim = require('imagemin-jpegoptim');
```

Pay attention to line number 4. This ‘default’ property is how it should be. Imagemin-webpack-plugin works that way. Otherwise you get an error!

Now is the correct time to put some images into your ./src/images folder. It doesn’t matter how many, but put more than one for the fun of it.

Then let’s go to plugins section and configure this behavior like that.



```
new CopyWebpackPlugin([
  {from: './src/images', to: './images'}
]),
new ImageminPlugin({
  test: /\.(jpe?g|png|gif|svg)?/i,
  plugins: [
    ImageminJpegoptim({
      size: '60%',
      progressive: true
    })
  ]
})
```

First the copy-webpack-plugin allows us to transport the images from the src folder to the dist folder. Then the optimization will happen to them (to the dist folder, and we can keep the original images in the src folder).

After that, we specify which type of images we want to optimize, inside the test (jpg, jpeg, png, gif, svg etc), and then we use the   jpegoptim plugin to configure the extent of the compression. For now we set this to 60% (very aggressive, but ok for now).

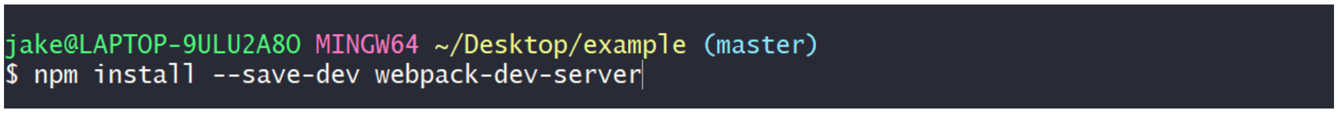
Hit npm run build. If everything went ok, then you have an images folder inside your dist folder. Check the original sizes and compare!

Create a   server instance

Let’s try to run our application now in the browser. Normally you have built-in servers inside your text-editors for that, but webpack also provides a development, or a production server runtime (is a node.js server actually).

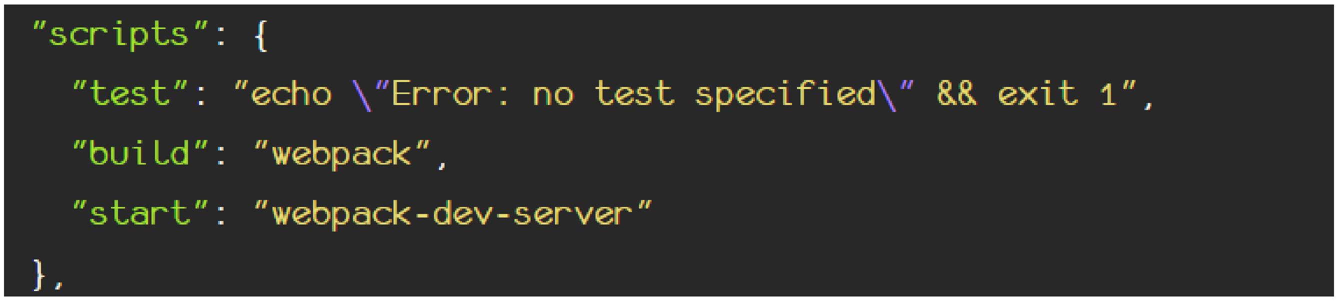
We are going to need one more package, the webpack-dev-server package.

Go ahead and install it.



```
jake@LAPTOP-9ULU2A8O MINGW64 ~/Desktop/example (master)
$ npm install --save-dev webpack-dev-server
```

Once installed, we want to instantiate a new server with a single terminal command. Thus we go to the package.json file and create a new script with the name of start.



```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack",
  "start": "webpack-dev-server"
},
```

Once done, hit npm start. If everything went good, open your browser to the url of localhost:8080.