# **Webpack**



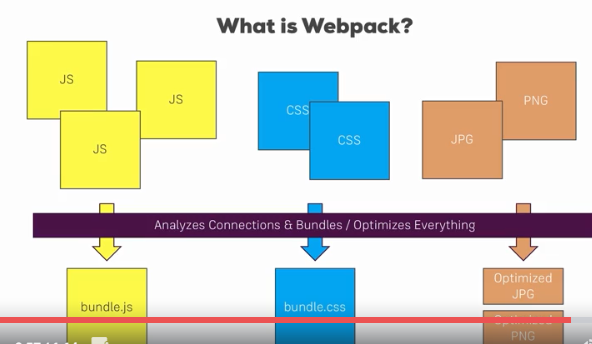
## **Introducing Webpack**

**Webpack is a bundler but it actually is more than that, a bundler alone would just concatenate files, webpack does that but it also allows you to optimize your files and you hook in various plugins and so-called loaders to also transform your files and for example transpile next generation javascript to current generation javascript.**

in its core,

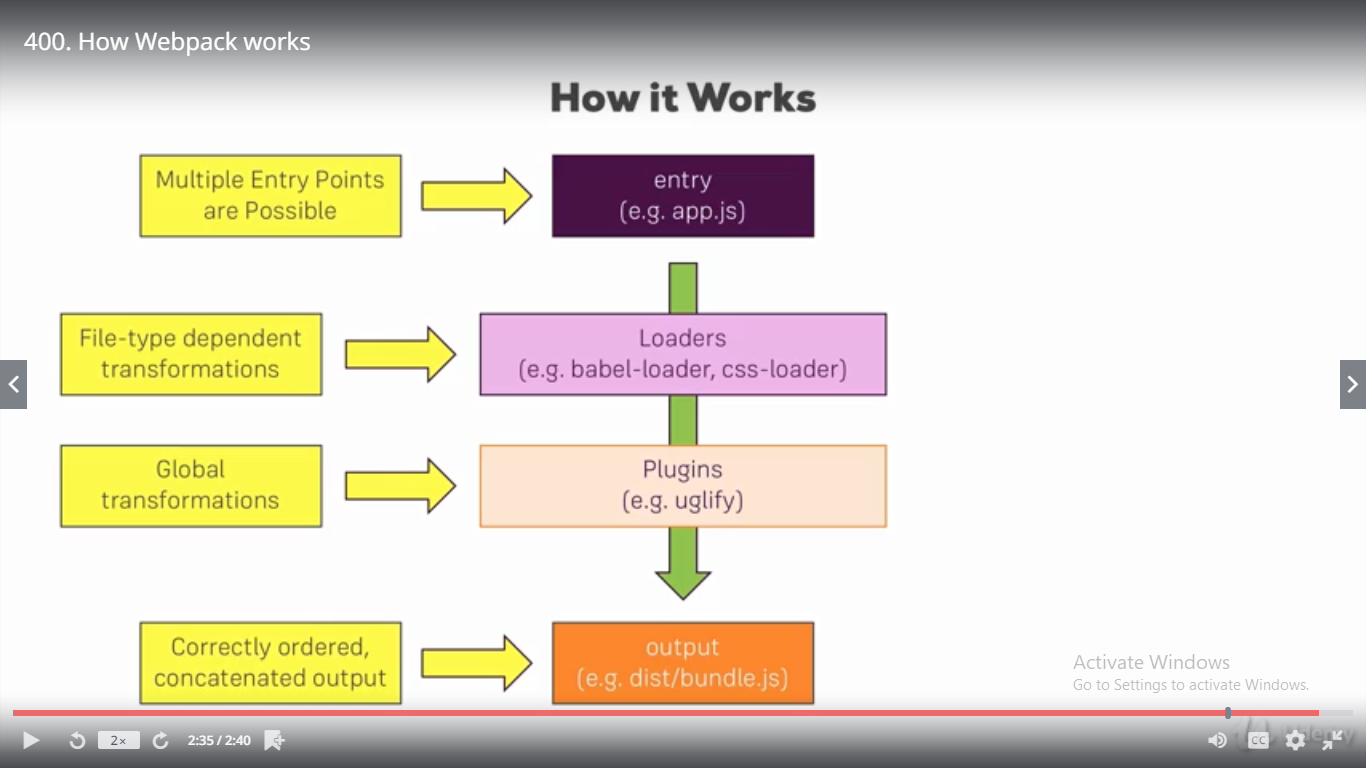
**the idea behind webapck is to have multiple javascript, css, image, whatever files and bundle them together. It analyzes connections between these files like imports and then bundles everything together, allows you to optimize it.**

webpack is the de-facto standard for setting up projects these days. That's why we'll also **use it in this module for setting up our own react workflow and project.**



## **How Webpack works**

let's understand how it works behind the scenes since it is such a crucial tool handling our whole build workflow.



Behind the scenes, webpack has **four important things**, four important features.

### **entry**

**it always needs at least one entry point, you can have multiple ones.**

**This could be our app.js file, our root javascript file which mounts our react application to the dom, which calls react dom render for example, it needs this file since it then analyzes the dependencies of this file and the root entry file will have a dependency to another file which then in turn has more dependencies, so webpack can build up a dependency graph starting with that root entry file so it can understand which files make up our application**

### **output**

**if we give it our entry file. It then analyzes all the dependencies and bundles them together into an output we specify it like a bundle.js file in a dist folder, we specify the file name and where it should go. And there it will put all these dependencies into that file, correctly ordered and in one concatenated output file.**

This is the core functionality but as I said it's more than that,

**in-between, there are two other important features we can utilize.**

### **Loaders**

**loaders are applied on a per file level so we can for example say all javascript files should get handled by loaderX, all css files should get handled by loaderY, babel-loader and css-loader are two popular examples which get used in a lot of projects, so loaders are file dependent or apply file dependent transformations.**

### **Plugins**

**We also then have plugins where loaders are applied on a per file basis, plug instead take the concatenated files, so the bundle but before it's written to the output. Here we can apply some general transformations or optimizations like uglify, so this is on a global level and happens after the loaders did their job.**

This is how webpack works, what it does behind the scenes and **this is all set up in a webpack configuration file.**

## **Basic Workflow requirement**

let's first analyze what the requirements of our workflow are.



We certainly want to support next generation javascript features so we need to be able to compile that next generation javascript code to current gen javascript code which runs in all major browsers these days. Additionally we want to be able to handle jsx so that we can use jsx in our script files, in our react project. We also want to add css auto-prefixing so that we can write simple css rules and get automatic prefixing so that it supports the best syntax in all browsers our applications supports. And we also want to support images of course, we should be able to import images as we did it in our create react app and then use it to our template. Finally we want to optimize the code, before we built that bundler or we spit it out, it should be optimized to shrink our javascript code to as small of a size as possible.

**These are the requirements, that's basically what creates react gives us though that also gives us more like linting and so on**

**these are the core requirements I also want to build in this custom workflow.**

## **Project and NPM setup**

Now the first thing is that we want to use npm to manage our dependencies, so to manage all the files and the tools we need to set up a working workflow. For that we need to put this project under control of npm which we do by running npm init

you'll see that the package.json file was added, this is the file which we'll use to set up our build scripts and to also the manage the dependencies of our project.

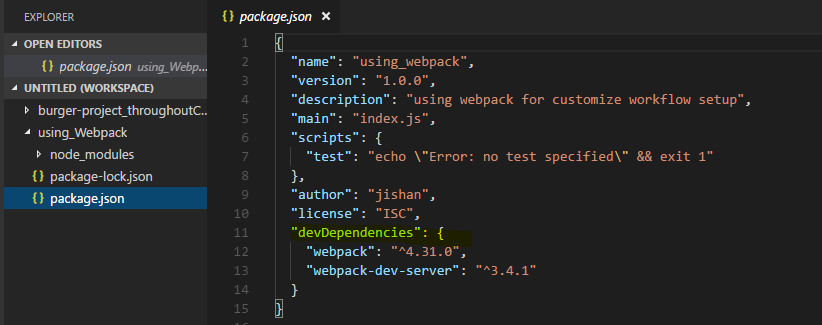
let's install import dependency, with npm install --save-dev,

**we install new dependencies and we mark them as development only dependencies ,it makes it clear which dependencies we use only for setting up the build workflow and which dependencies really have an impact on the running application,**

**you will see that they will be grouped differently the package.json file later. Both --save and --save-dev add an entry to package.json so that we can easily share our project without having to share the big node modules folder too.**

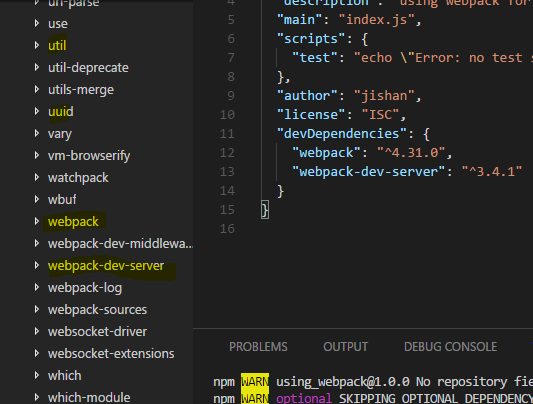


**Webpack-dev-server is this development server we want to use so that we can test our application locally on the machine and webpack is well basically the build tool itself, webpack-dev-server wraps this build tool, they're both from the same team.**



you see that we have the new dev dependencies node in the package.json file.

**we can now share this project without node modules which has all the source code (dependency) downloads in this course and everyone taking this project can just run npm install so that all the dependencies listed in dev dependencies or dependencies, the normal dependencies get installed and the node modules folder gets recreated.**



**In the node modules folder, we see a bunch of dependencies not just webpack and webpack-dev-server, that simply is because web pack and webpack-dev-server have their own dependencies which they also installed but here you can see webpack and webpack-dev-server. So with that, we can use these dependencies.**

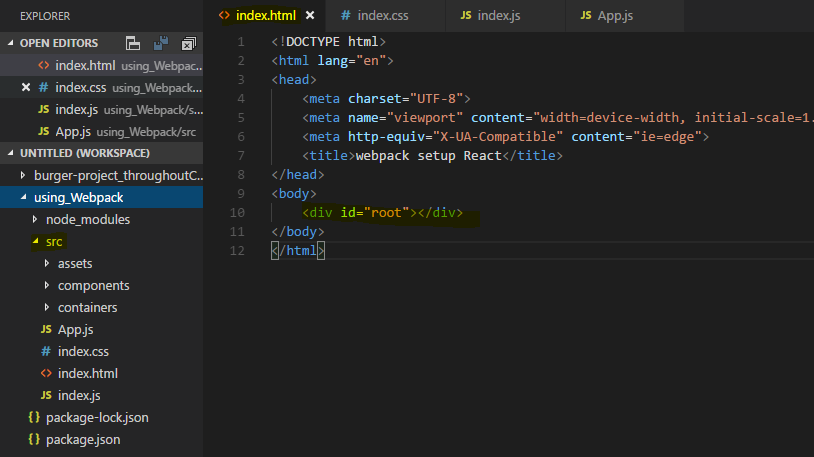
## **Creating a basic folder and file structure**

we now want to set up our project and first of all I'll set up the project structure I want to work with.

Now I won't recreate the exact structure of create react app but a fairly similar one.

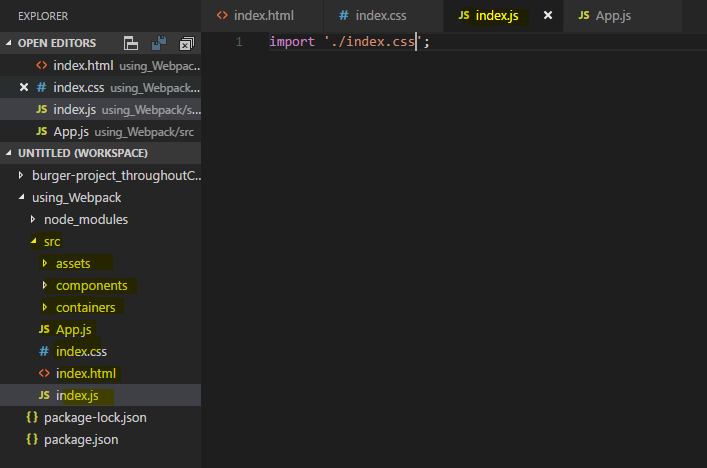
**I'll create a source folder in our root folder and in that source folder, I'll add my index.html file so this single page which gets served in the end. if you got any imports from CDNs of fonts or styles you can add that in the head section, the scripts and any other imports generated by the workflow will be injected into this file automatically though, we'll add a configuration which does this to our set up.**

**With that I'll now add a div to this body which should get a specific ID root, this will be the div where we'll later mount our react application to.**



now let's set up the other parts of this project. In the source folder, I want to have the assets folder where we could store images or anything like that we use, I also want to set up a components folder and a container folder so basically the same setup I used in my other project in the project in this course.

Now **I also want to create a react application, for that I'll create an index.js file in the source folder and the index.css file which I want to use for some global styling**, so this will be imported in the index.js file, there I'll import that index.css file

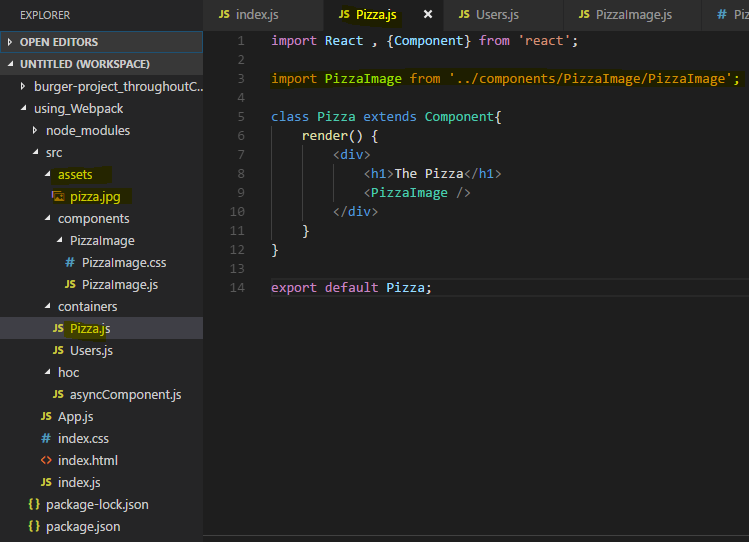


**now I also want to add an app.js file which should be my root component** and then so that we can really see that all of that works,

I want to set up a basic react project which isn't super amazing but which uses routing and lazy loading so that we can see if that works and which just simulates a lot of features we might use in a typical react application, for example it should also incorporate an image. Now I'll set up this demo react project in the next lecture

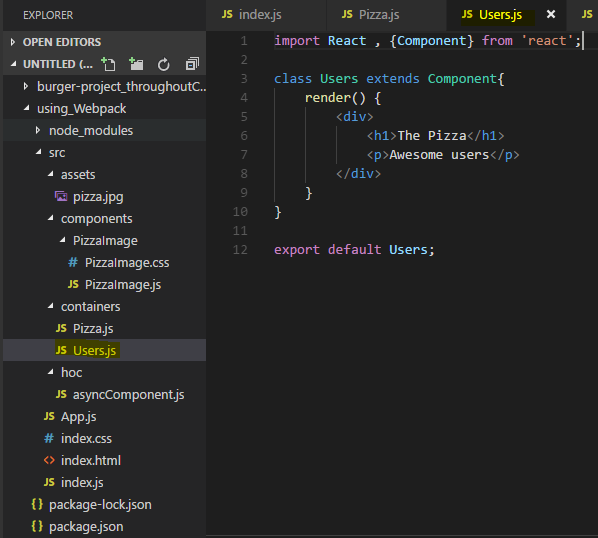
## **Imp - Creating a basic React Application**

**Let's set up a basic react application we can work with**



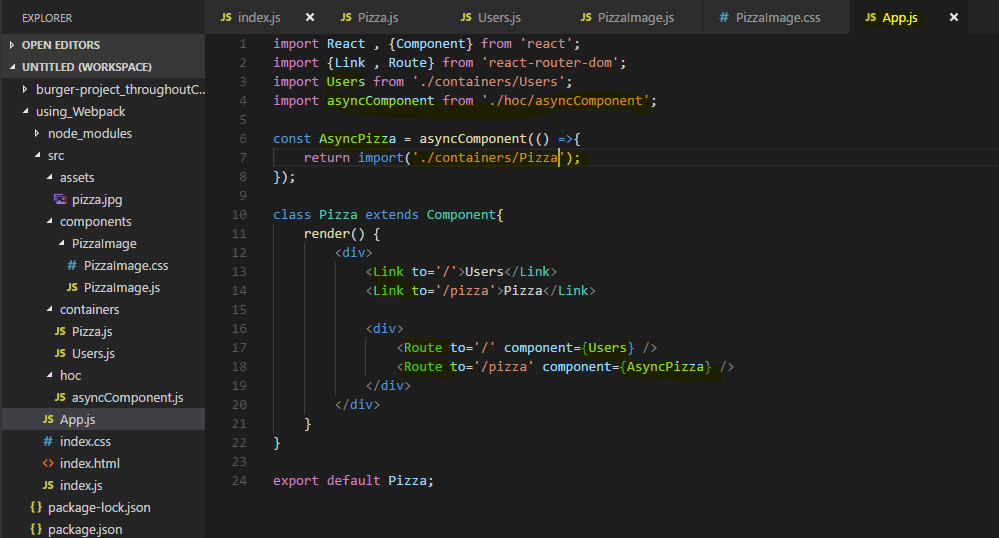
So inside my pizzaImage div, I then want to have an image, self-closing where I want to import that image and output it here as a source.

**Note : my build workflow should essentially copy that file to a new destination when you build the application then and give me a link to that file in this constant PizzaImage automatically.**



Now I've got two containers,

**now I also want to load them via routing. So in my app.js file, my root component**



**Here in App.js file , I want to Lazy load my Pizza component hence I have created a hoc component which import the Pizza Component asynchronously whenever needed(we can also Lazy load by React.Lazy – react 16+ version)**



This is my app set up thus far

**now of course we need to mount this app to the dom, for that I need to go to the index.js file so I'll add react-dom render here at the bottom to then render app in the place of this root element, so this div with the ID root in the index.html file.**

So for that, I'll now reach out to it with document\_getElementById and then pick the root element here.



Now with all of that, our react application is mounted

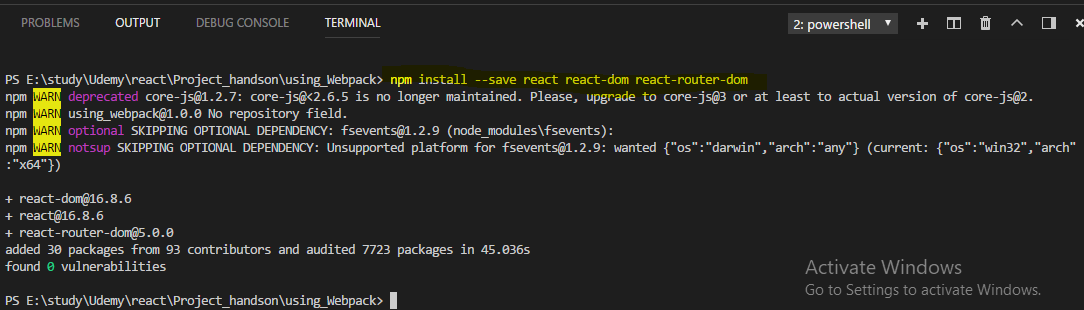
## **Installing Production Dependencies**

now it's time to install some dependencies,

for example the react application obviously uses react or react-dom and the react router

so we should install all these production dependencies,

we do this with npm installed **--save,** **not --save-dev because we want to indicate that these are production dependencies**



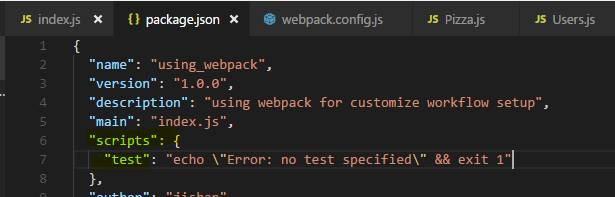
 installed and therefore placed into the node module folder, appropriate entries with the latest versions are also added to the package.json file

we did install webpack a while ago already,

we now need to install a couple of other dependencies to build a workflow and I will add them step by step whenever we need them. So what I want to do is I now want to start with setting up the webpack configuration such that it is able to handle next generation javascript, css, images and all that fun stuff.

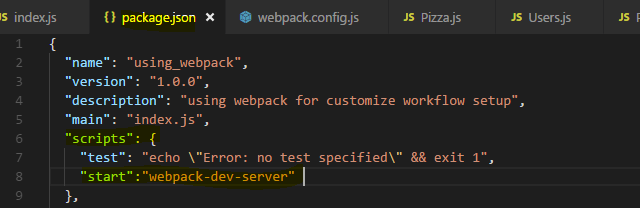
## **Setting up the basic webpack config**

We already installed the webpack tool, the dependency in the package.json file and I said that I will manage the workflow through the scripts we have in package.json file



which we then **can run with npm run and then the script name** or in the case of the special script, **npm start we can just type npm start, and the start script should be the one launching the development server.**

So I want to add a new script to the scripts section which is called start and on the right side of the colon, in the double quotation marks which you need to use since this file uses the json format, here I want to simply execute webpack-dev-server



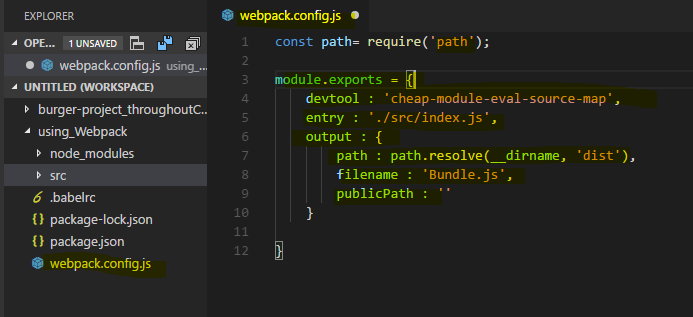
**make sure to not mistype, this has to be the name of this package which simply exposes a function you execute with this call. So this will spin up the dev server,**

however the dev server out of the box won't work, we need to configure it and to configure it, I'll add a new file next to the package.json file, I'll name it webpack.config.js **this is a special file name, make sure to name it exactly like this webpack.config.js. Webpack will automatically look for this file when executing either webpack or webpack-dev-server as we do here, so it will automatically take this file into account, therefore you have to get the naming right.**

in this file, I'll set up my development workflow. We'll later create a separate file for the production workflow because there, we will differ in some aspects, we will apply some extra optimizations and so on.

n the webpack.config.js file, I'll now set up my webpack workflow and for that, we need to export something here which webpack will then use, we do this with the node export format by writing **module.exports and we then set this equal to a javascript object.**

**So this is the syntax webpack understands to read in the config we now specify in this javascript object** and I did mention the four important aspects of that configuration



**let's start with the entry point. Here we define where the journey starts for webpack and we want to point to our index.js file here, this tells webpack that it should look at this file first and then analyze the dependencies of that file which are all the imports and then it is able to do something with these imports.**

**However to be able to handle css or any other non javascript file, we need to give webpack some extra set up, we need to do this for javascript files too though since we use jsx which is not native javascript and since we use next generation javascript features too,**

**we also import images and stuff like that so that's all the stuff we have to handle with loaders later.**

**Before we do that, I'll add one additional configuration, it's called devtool and here we can define which kind of source maps, if any webpack should generate and I'll use cheap-module-eval-source-map here. Now that's super confusing long term, in the end it is the best kind of source maps you can generate for the development, very detailed, good performance, so this is what you should use here to get source maps which allow you to easily debug your original code in the browser.**

So with that we get the entry

let's now also define the output

**so the output key and here I want to create a new folder with a file, output is not just a string, it's an object and in that object we configured the path, so where this should be stored, the file name, so where our or how our file should be named and the public path, that's important for webpack to know where our files are put to and if that then is the root folder of the server in the end or if it's a nested folder.**

**So here it should be the root folder so public path is just an empty string which means you store the files in a specific folder and you don't need to adjust any imports or adjust for anything, the file structure, the folder structure will be the folder structure as we deployed in the end. More important is the file name, here that's a string too, we define what the file should be named and I want to name it bundle.js, it's a convention to use bundle.js just because it is a bundled javascript in the end.**

**Now you might wonder what happens to css images, this will be handled differently by certain loaders we add and the loaders will define where the css or image code is put,**

so this only affects javascript therefore it's bundle.js.

**Now the path is also important, where should the assets be stored? And there I will actually import a node module which is called path which I do with the node import style where I require the path module. Now we didn't install this with npm install but it is a default node module, so since we have node installed we can also pull this in. The path object here has a resolve method which allows us to generate an absolute path in the end, here we can pass a special variable which we have available in node systems and webpack will use node behind the scenes, so it will have access to that variable, it's \_\_dirname and it refers to the directory**

this is running and then the second argument to resolve is the folder where we want to create it and resolve will then create an absolute path taking the full path to the current folder on our operating system and appending dist,

**so it will in the end output a file in the dist folder which doesn't exist yet which it will create instead which is named bundle.js, this is the output of this operation.**

**Now that is the base set up here, this doesn't transform any files or anything like that though** so these are all things I'll take care about in the next lectures.

## **Adding File Rules**

One important thing we have to add upfront is to make sure that webpack automatically appends .js at the end of these imports here so that it is able to import that correctly.

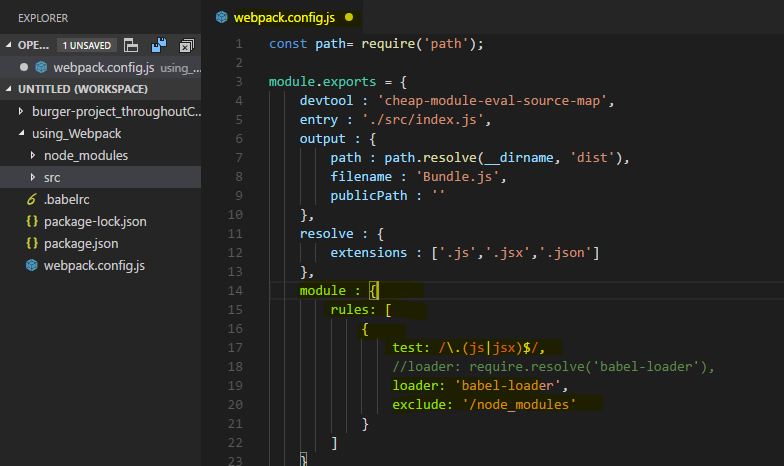
Now to get it to behave like this, we have to configure it in a certain way, in our webpack config on the same level as entry output and so on, **we add the resolve configuration**.

**That takes a javascript object and there we can resolve some extensions, so we can basically tell webpack that it should be aware of certain extensions and if it encounters an import without an extension, it should try these extensions and see if it finds a file of one of these. So that's an array of possible extensions and I want to support .js , .jsx and .json here so that it will see if it finds a file with that extension if no extension is defined.**



**Now the next and most important thing are our loaders though where I want to make sure that we do correctly handle different file types, this is done with a special configuration on the same level as output and resolve which is named module because I import a dependency is referred to as a module, so you could kind of translate module with file.**

**So here we set up what webpack should do with the individual files and we control this with a rules array.**



**Now each rule is a javascript object and each rule now needs to first of all test, which means test if a given file, a file webpack identified fulfills a certain criteria, this criteria is the file name and is checked with a regular expression, the regular expression commonly checks the file extension.**

**So we could say anything ending with .js or .jsx and I have to escape the dot here since we're in a regular expression and otherwise it would refer to any character.**

**If we're in a .js or .jsx file, so if it ends with .js or .jsx, then I'll want to apply a certain loader, now a loader is like a third party plugin which does something to the file.**

**Now for js or jsx files, I also want to add an additional configuration which is exclude which allows me to exclude certain patterns. Also a regular expression, I only want to check for node modules in the file path, so it shouldn't try to transform anything in node modules since these are third party libraries we already do import, we don't want to transform them again, they are already optimized, it should only touch our own files.**

Now what about this loader though?

**Well for javascript, we will use the babel loader, babel is the defacto standard for transpiling next generation javascript to current gen javascript, so let's install and configure babel in the next lecture.**

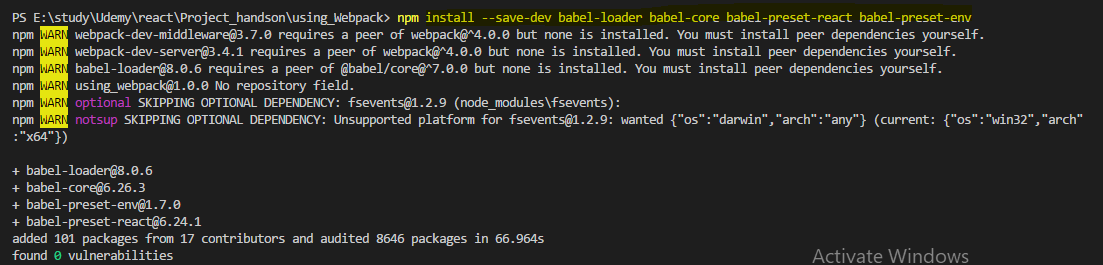
## **Introducing Babel**

for javascript I mentioned that I want to use babel, the defacto standard for transpiling my javascript.

**We will install it with npm install --save-dev again because it's a development only dependency**

**we need to install the babel-loader package, that however does just provide the hook for webpack. Babel itself is a library which kind of has the knowledge about how to transpile your javascript, so we also need to install babel-core which holds that knowledge.**

**We now also need some configurations to basically tell babel which kind of next generation javascript or which special javascript syntax we want to support and here, these are so-called presets, the packages we need are babel-preset-react to support jsx and babel-preset-env for an environment adjusting preset which basically has a look at the features we use and automatically transpiles everything correctly.**

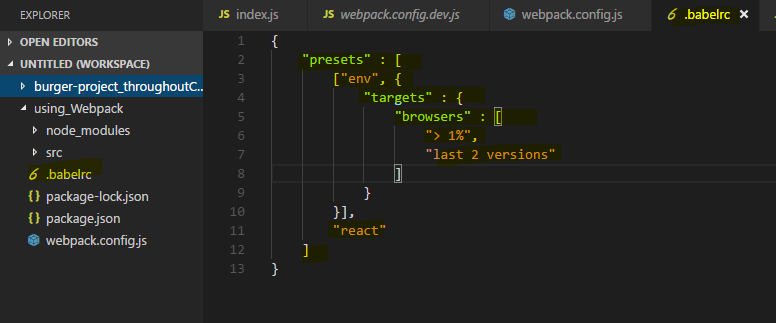


**now we need to configure babel because just because we installed the presets doesn't make babel use them, it doesn't analyze all other packages we installed and then see if there is a preset inside of them.**

**So instead we add a new file, dot and the dot at the beginning is important, .babelrc**

**that's a configuration file babel will automatically read if it's in our root folder, so in the same folder where we will later run our webpack build workflow.**

**The babelrc file takes adjacent configuration and there we should add a presets node, now we can make babel aware of all the presets we want to apply**



and here**,** I want to apply two of thepresets,the first one is env and the second is react.

So these are the two presets we installed without the babel-preset- thing at the beginning, just the path at the end so react and env, these are the two presets.

**Now the env preset will receive some special configuration from my side, I do this by enclosing it into an array with square brackets and then the second argument after the name of the preset is a javascript object where I can configure it and here I want to set up the targets of that preset which means which browser versions do I want to support,**

**that's of course important. It automatically identifies which next generation features I use but to know what next generation actually means, it needs to know what my current gen is,**

**so which browsers do I target with my app?**

**So targets is another javascript object where we can then set up browsers and browsers then is an array where we can use the browser list type of configuration, you can google for browser list to find this repository here to learn more about the syntax you can use for configuring browsers. However the syntax I'll show you here is a pretty common set up.**

**So what we want to do here is we want to target greater 1% of market share or last two versions so this is will be combined with or in between.**

**So now this is the target environment we want to support and this is what babel will then take into account when compiling our next generation javascript to current gen, it will compile it to javascript which runs in browsers which fulfill this criteria.**

So with that, we get the babel configuration finished, now we can add the babel loader in our webpack config

**I'll therefore add a loader in single quotation marks, the babel-loader. So here I take the full name of the package we installed, babel-loader.**

**So now it will be applied to javascript files compiling the jsx and next generation javascript features,**

**it would still fail though because we also import css which webpack wouldn't understand, it only understands javascript natively and in the pizzaImage component, it would import css and even an image, so we need to support that too.**

### **Note :**

**If you have latest webpack installation , then babel installation and file also changes**

**npm install --save-dev @babel/preset-react**

**npm install --save-dev @babel/preset-env**

**npm install --save-dev @babel/core**

Let's add support for these types of files in the next lecture.

## **Adding CSS file support (css-loader/style-loader)**

let's now also support css files and images. For that, I'll add a new rule and I want to start with css,

so we test for files ending with .css, that's our simple test.

I also want to exclude anything in node modules here just to be safe

and then we can define which type of loader we want to use here.

**Now for css, I actually need a more complex setup, for js we also have a quite complex set up but that's all handled through babel and our babel config,**

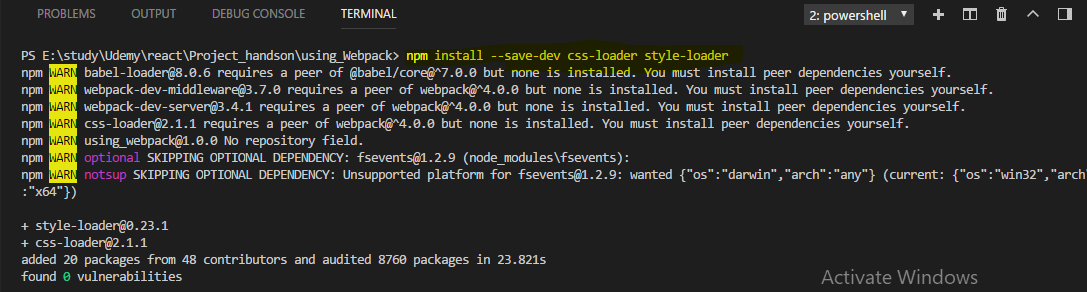
for css we'll configure it here in the webpack config file.

**So instead of loader I can use use here, that's the long form, loader is the very short form if we just want to set up a loader without any config, if we want to set up multiple loaders or a loader with config, we should use use here.**

It takes an array of the loaders we want to apply and here, I now want to install two loaders

I'll do that with npm install --save-dev css-loader style-loader

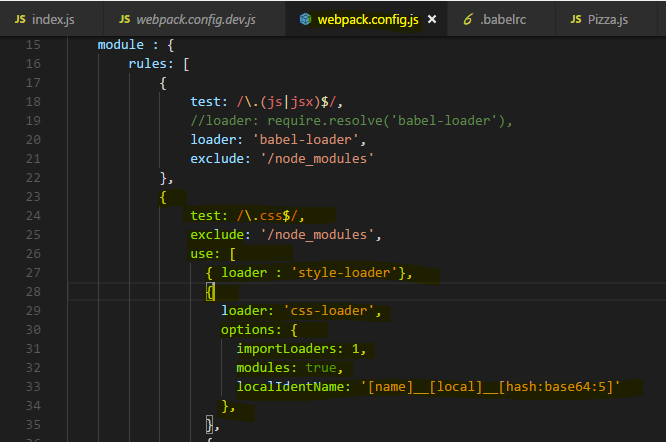
**Css-loader basically tells webpack what to do with these .css imports and the style loader. That's a loader which will then extract the css code from the css files and inject it at the top of our html file hence reducing the amount of file downloads we have to make.**



**in my use array, I'll add a javascript object which has a loader property again where I define the style loader, it's followed and the order does matter here by another object with a loader, the css loader.**

Now important and this is easy to mess up,

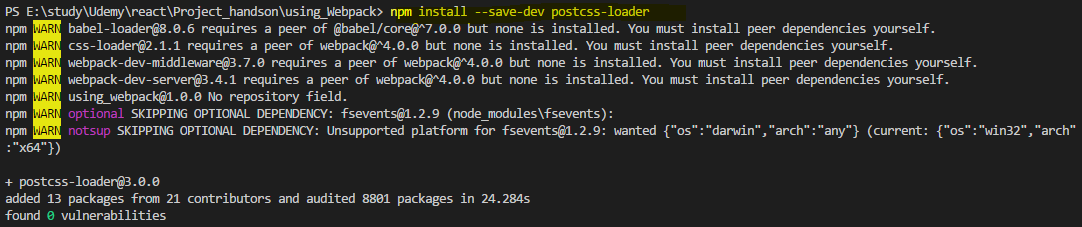
**webpack parses loaders in this use array and applies them from right to left, so from bottom to top if we write it like this. So it first takes the css loader which it should because that makes it understand the css imports and don't throw an error and then it applies the style loader on the extracted css code and that's exactly the order we need here.**



**Now we can also give the css loader which is responsible for parsing and handling the css another property which is called options. Now options is a javascript object where we can configure this loader.**

**now here I want to set modules to true to enable css modules and I'll set up the localIdentname to define how the generated css classes due to css modules should look like. And here I'll take the name of the class we defined, two underscores then the module name, so the component name and then a hash to really have unique names, this is the same set up we added in the styling module to do the configuration create react app gave us.**

We're not done with css though,

**I also want to add auto-prefixing and for this I'll install another package with npm install --save-dev postcss-loader** 

**now postcss sounds like it does something to css after we parse that, here it'll actually run before the css loader, dive into the css file and adjust our code before css loader pulls it out and adjusts the class names and so on. The post css loader simply is a loader which allows us to transform the css,**

**for example we could also use postcss loader to handle sass and stuff like this, here**

**however I will simply add it as another loader and therefore in third position because it should run before that css loader**.

Here I will set up my loader which is postcss loader and I need to set up my options on that loader with the options key where I now want to set the ident option, that's important internally here

**but more important are the plugins, that actually takes a function which returns an array and these are now steps we should apply or we want to apply to transform everything.**

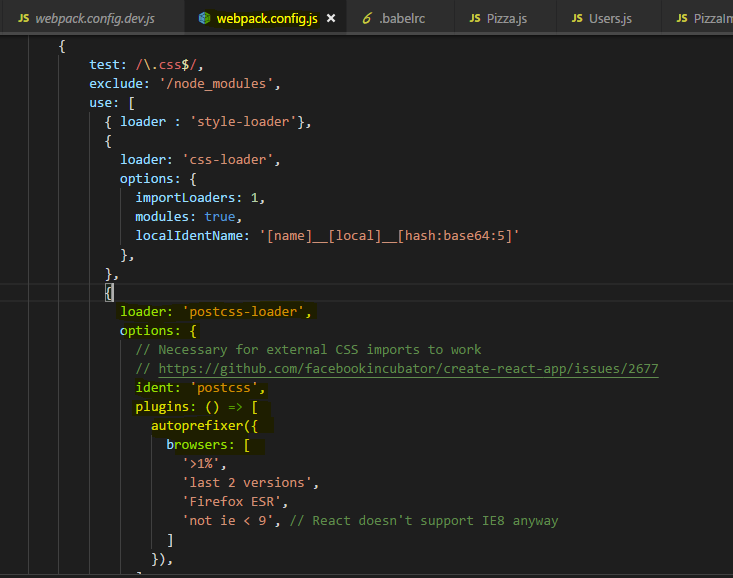
And here I want to run **autoprefixer**, now autoprefixer needs to be installed.

It's a separate tool. So I'll run npm install --save-dev autoprefixer,



**that's a third party library which is able to auto prefix our css properties, we need to import it into the config file** so I'll create a new const, autoPrefixer which uses the required syntax to import autoPrefixer.

Now with that import added, we can add this to the plugins array here, so here I'll add autoPrefixer and this is actually a function we need to execute and we pass a javascript object to it to configure it and there again we pass a list of browsers



That's the bare minimum set up I want to use.

**Now since we run one additional loader before css loader, we actually need to inform css loader about that, that's just a special set up css loader needs.**

**So there I'll add importLoaders : 1 because we run one loader before css loader is applied.** And that's our css handling setup

let's continue and let's make sure that we now also can handle images since we also import these.

## **Creating rules for Images**

I'll add another rule and that rule should test for images. I want to use a loader we have to install with npm install

**npm install –save-dev url-loader**

**URL loader is a loader which will take our images and if they are below a certain limit we define, it will actually convert them into data 64 URLs which it can inline into our documents, so we don't have to download extra file. But for bigger files, it would be inefficient so files above that limit we specify will simply be copied to our output folder and it will then generate a link to these files and put that into our import we use in our components.**



What we do here is we use the URL loader here but now we configure it with query params and we do it with query params so that the configuration here, it gets automatically passed onto a fallback we will use if the limit is exceeded.

Now that fallback is another loader we also need to install though

**run npm install --save-dev file-loader**

**file-loader is a loader which in the end simply copies the file you could say, because it copies it into a new direction and gives us a link to it and that is the fallback we'll use automatically here if we exceed the limit**

we now define here. As a query param, limit could be let's say 8000kb, so anything above that will not be inlined but will be copied. And we need to pass some configuration to the file loader, the name and that's also the path of where we want to store that file, I'll use **images/[name].[ext], these are placeholders taking the original name and extension. So the image is going to get copied into that folder, in our dist folder because that output set up is taken into account even though we're not creating a bundle.js file but the path still is taken into account, so that is where our images will end up at if they're exceeding this limit.**

So this is our loader for images and with that, the image files we import in our javascript files shouldn't pose a problem anymore.

**Now you can add more rules for different file types, for fonts for example, but I'll keep it to it for now**

let's now turn our heads towards plugins we might want to use and then most importantly, have a look at how this actually works and if it does work.

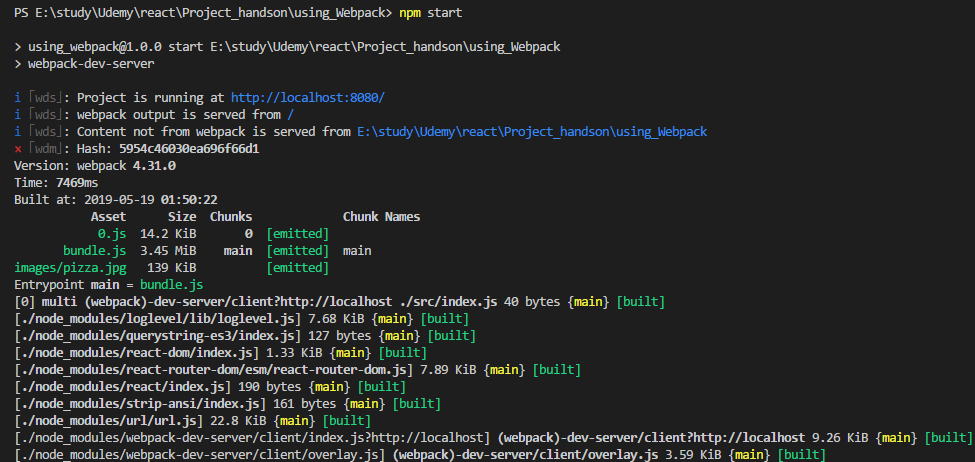
## **Lazy Loading support in webpack**

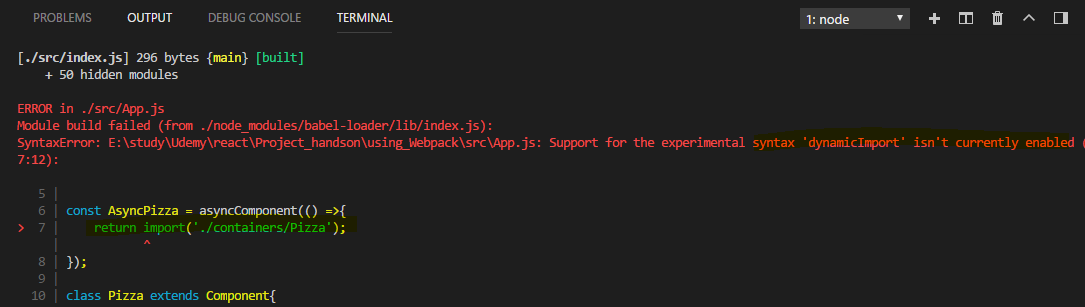
now I want to have a look at plugins and then see if the setup actually works.

Now regarding plugins, it's actually super simple for us here, we don't need any for development workflow, we will add one later when we set this up for production, there I want to add some plugin but for now that is all, I won't add a plugin here.

Let see if this works

if we run our start script which runs the dev server which I said would automatically take this config file.





**So now at least it doesn't crash when starting the server, it does give us an error though that it doesn't understand this import syntax here, the dynamic import syntax, it doesn't understand this.**

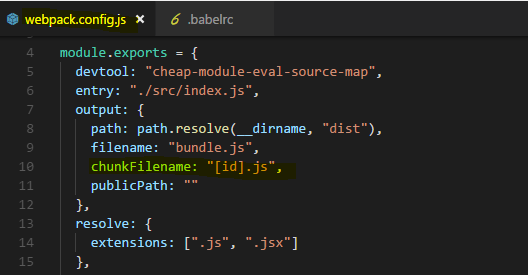
So how do we make it to understand this dynamic import?

**We need to adjust our setup here to be able to create dynamically generated extra chunks of code, lazy loading means that it's an extra bundle and not part of the main bundle which is downloaded initially, to support code splitting in webpack and code splitting is just a different name for lazy loading,**

I have to add **chunkFilename** here to my output config.

**This simply determines what these other files, the separate files which are generated for the lazy loaded code which is of course not included in the bundle**, that would defeat the purpose of code splitting, what these file name should be named.

They have to have a variable part in their name because they are generated by webpack dynamically, I'll use “[id].js”



here, IDs are dynamically generated ID 0, 1, 2 and so on, generated by webpack **so that feature is outsourced into separate bundles received their own javascript files.**

Additionally we need to make babel aware of this dynamic import syntax which it otherwise doesn't understand I'll install a plugin

it's called babel-plugin-syntax-dynamic-import and this will make sure that babel understands this dynamic import syntax.

npm install --save-dev babel-plugin-syntax-dynamic-import



So once this was installed, we can go to the babelrc file and we already get presets, now I'll also add a plugins node here which also is an array and here I'll add my syntax dynamic import plugin, so that's the name of the plugin.



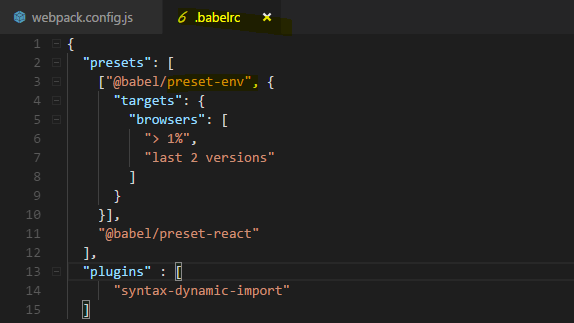
With that added, let's save all these files and let's retry npm start and see if this now works,

now this compiles but we get a different error,



we get an error that this syntax here where we assign our properties like this isn't supported.

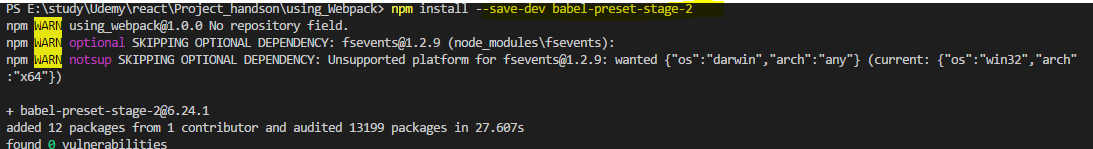
**The reason for this is that the env preset which we're using, this one here only supports latest browser features that are part of the specification. this special property syntax is only in draft state though and it's not included in the env preset**

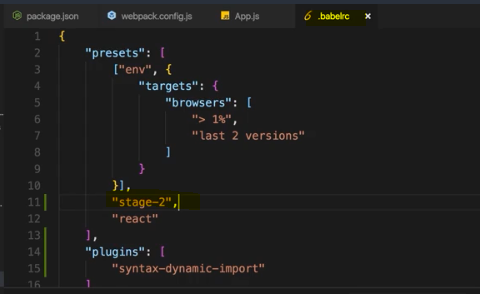


It isn't a problem though we just have to add another preset.

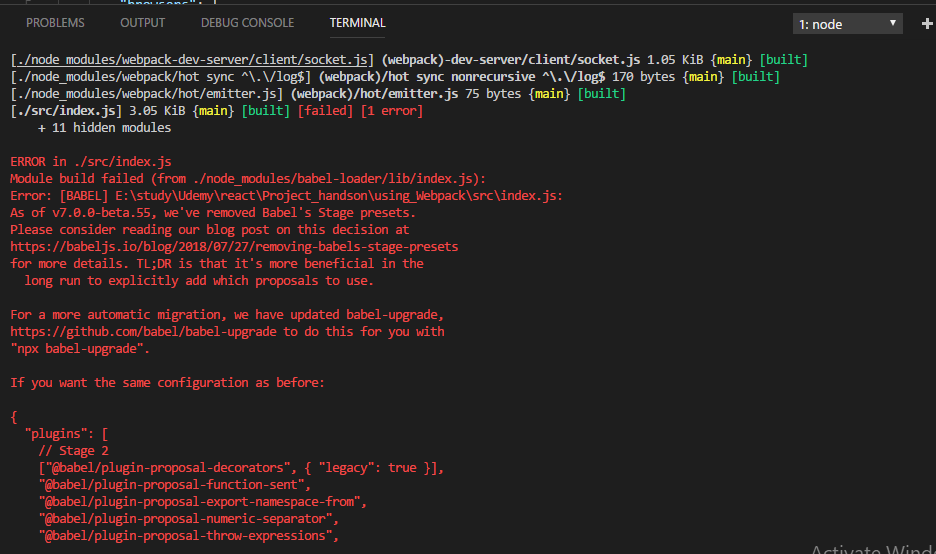
we now install another preset which is named babel-preset-stage-2,

**stage 2 presets simply are draft features which may eventually get supported but which we already want to use.**

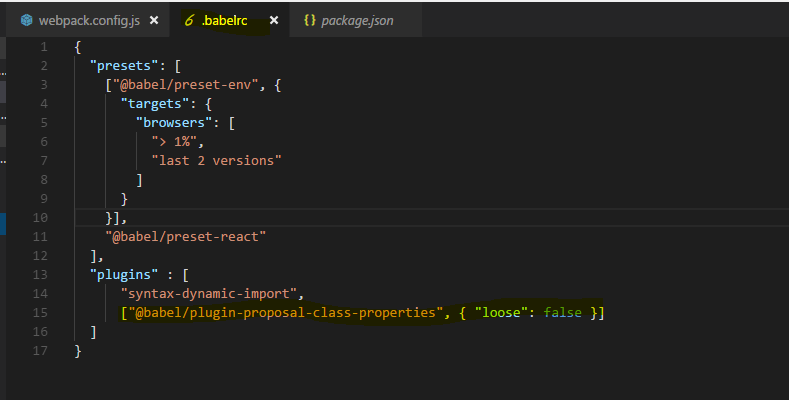


Now we have to add this to our babel setup 

### **Note : Error**

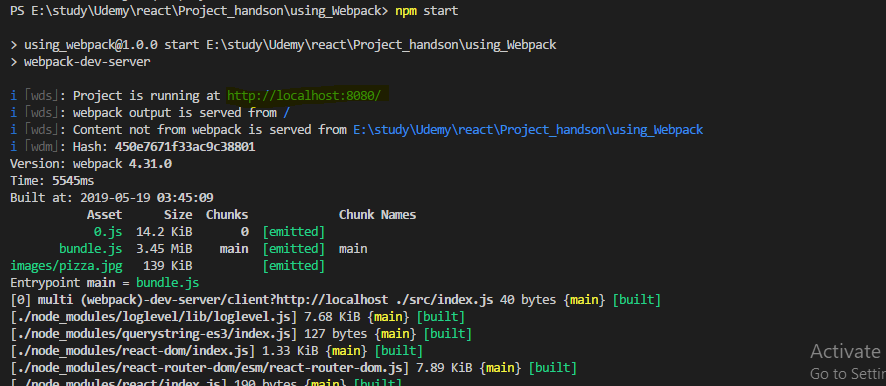
If you get this Error - 



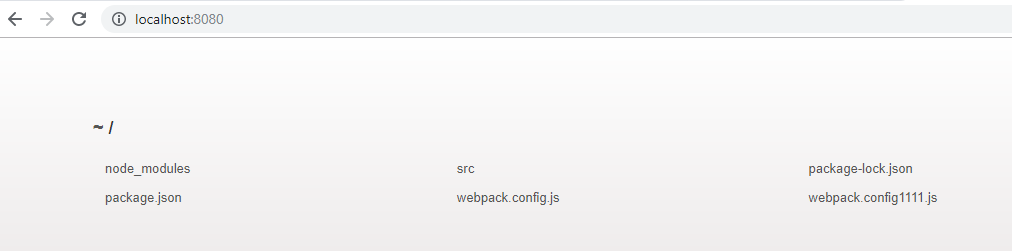
**Try this -** 

Let's now try this again with npm start and now this looks better, at least we don't get an error,

we'll face one problem though. The application is now served and by default, it's doing that on localhost8080,



so let's open that, there however we won't see app,



**the reason is that we didn't connect our build output which by the way is not stored in the dist folder when we use webpack dev-server but in memory, that we didn't connect that output with the index.html file, we have no imports in that file,**

how would we add the script to it?

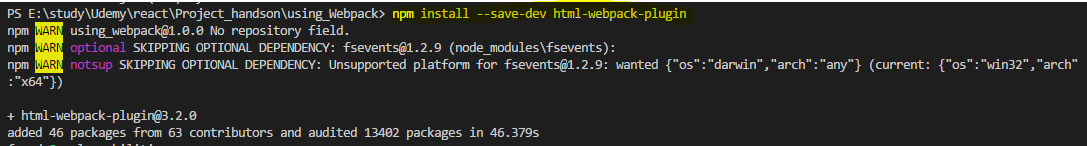
That is something we have to do on our own and we will do that in the next lecture.

## **Plugins – Injecting the script into index.html file**

**To make our project work and servable by the dev server or later in production, we need to connect our index.html file to the output files which are generated, when using the dev server they are only stored in memory but they're still generated.**

We need to install a webpack plugin for that,

**npm install --save-dev html-webpack-plugin**



**this is a special plugin which allows to connect our html file with our output and webpack will do that automatically and inject our bundled script and so on into that html file.**

In the webpack config, we now add plugins, it's a new node on the root config level, plugins and it takes an array as an argument or as a value, I should say.

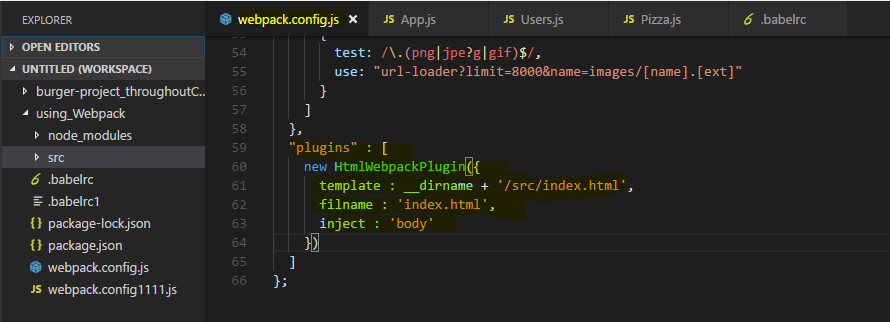
And now first of all we need to install or import I should say the html webpack plugin, so I'll create a constant which I'll name htmlWebpackPlugin but the name is up to you and here I will require html-webpack-plugin, the package we just installed.

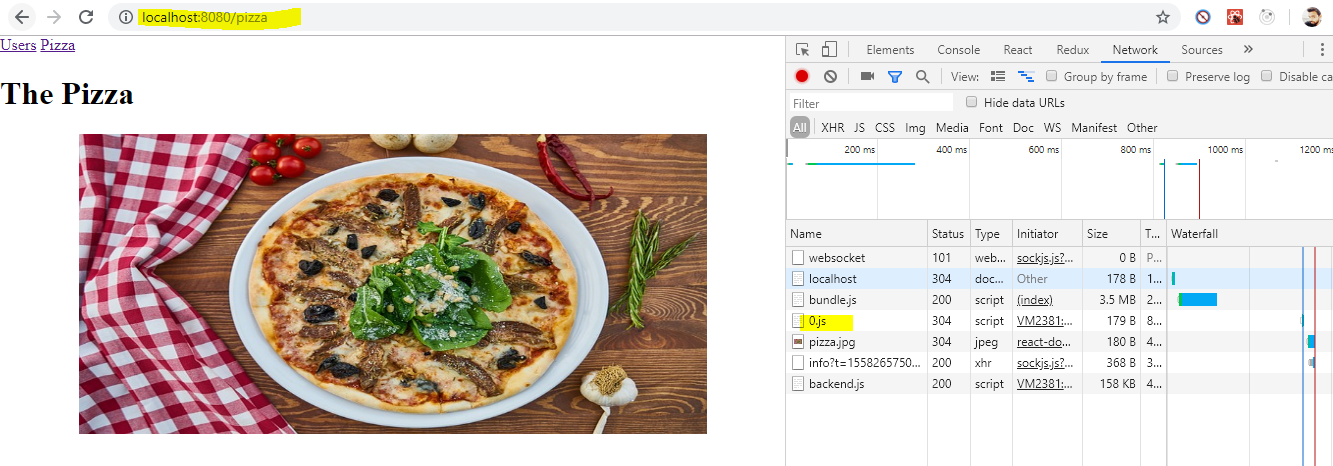


Now in the plugins array,

**I create a new instance of it, new HtmlWebpackPlugin and we pass an object to the constructor where we configure this plugin. I'll setup a template, so the html file I want to use as a basis  /source/index.html, constructing a path to it. So now we're pointing to the file we want to use as a template, I want to inject my stuff into the body that's extra configuration we can pass to the plugin here and the output file, filename should also be index.html, that's the file it will eventually generate, again for the server only in memory,**

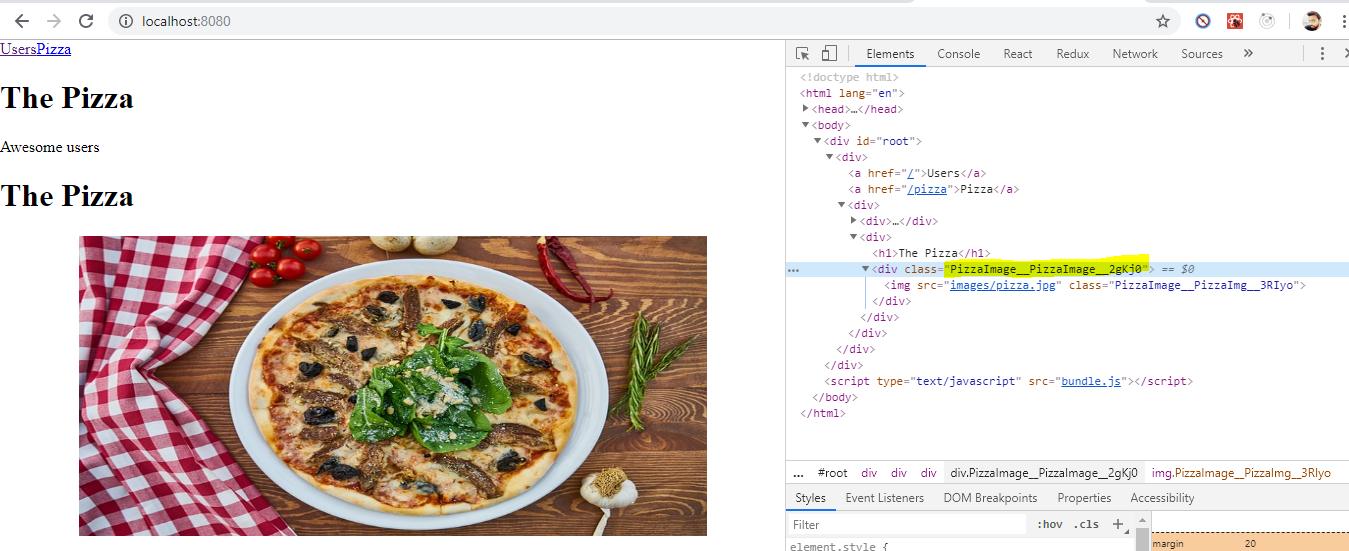
**later when we use a different workflow for building this, for production, it will give us the real files we could upload to a server.**



**Lets now try it -** 

**if we inspect the network tab and I go back to users and reload the app, watch the network requests, you see that 0.js file? That's our lazy loaded javascript file we downloaded, so lazy loading is working here and our react application is working**

**he styling also seems to kick in, if we inspect the pizza image we see we're using css modules here and we get the styles we setup in our css files.**



Now we have a working workflow, a working project setup where we can create a react app with the features you learned about in this course.

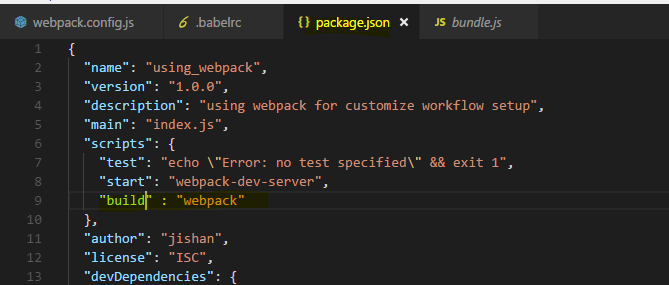
**Now that's only the development workflow, the development server,**

**if we now want to ship this app to a real server, we'll have to do some adjustments to the setup to have it prepared and to have it be optimized for production.**

## **Imp : Creating a Production Workflow**

Lets work on the production workflow.

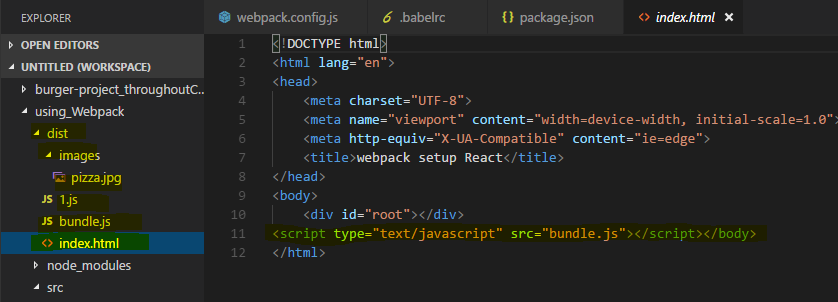
**Add a new script in package.json file to create a production build for the application**



**I want to have a second script, let's name it build**

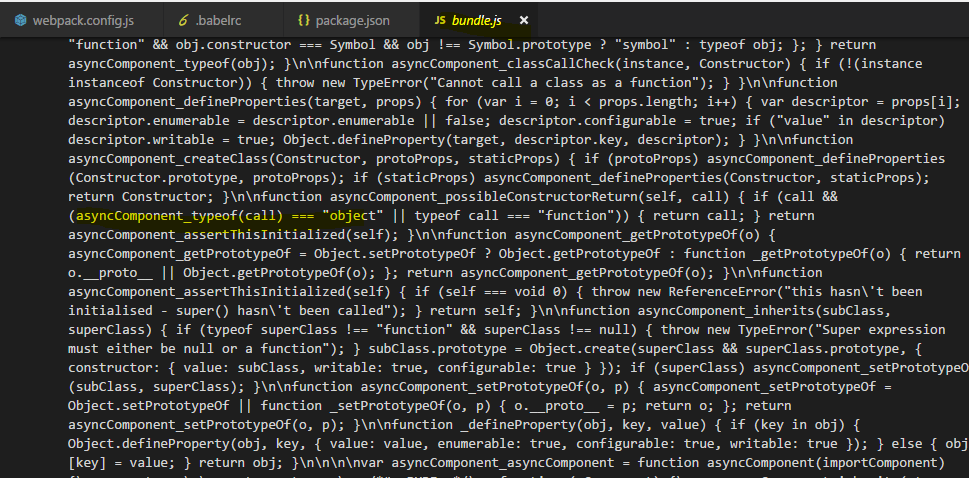
**here I don't want to run the webpack-dev-server but the webpack. Now if I would run it like this, it would use the setup of the development workflow. We can actually do this to see the output because the development server did everything in memory, now we can create the actual files by running npm run build.**

**So npm run build will now run webpack itself and take the same config file we worked with before but now it gives us this dist folder where you now see the image file it copied and you see the javascript files here. You see the index file it copied where it also injected our import and that is all the code we have**



**Also we can see the bundle.js file automatically injected into index.html file**

now what you can see this file already is it's not really optimized at all, this certainly isn't the code we want to ship



for now. The goal instead is to have a dedicated workflow for production which does some optimizations.

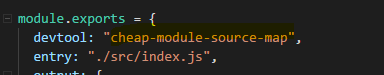
**I'll add a second config file, webpack.prod.config.js. Now it won't automatically take this file into account, we will have to point at it but that's actually super simple.**

Now I'll copy the setup from the other file, from the development workflow so that we can adjust it in the production workflow

Let's see what we need to adjust.

Let's start at the top,

**the dev tool, here I'll remove the eval to create more optimal source maps which are less resource intensive, you can then always decide whether you want to deploy them or not but it's nice to have source maps here too to quickly find some bugs in the production workflow, if there are any.**



Now the entry file still should be index.js and the output also still makes sense as it is, now

the loaders, what about the loaders? Well we want to apply the same transformations as in development so we don't need to touch the loaders.

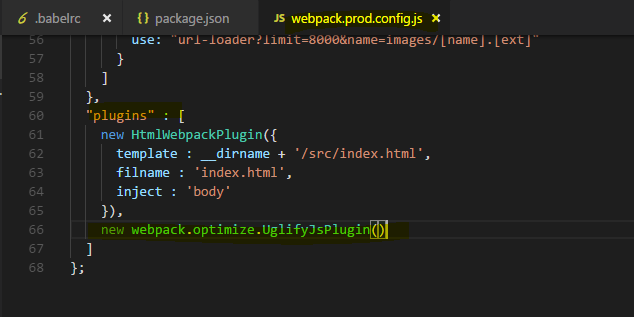
we'll do our work in the plugin section,

**now there we have a plugin for connecting the html file and I want to keep that, what I want to do now is I also want to uglify my output, I want to optimize it and that actually is a plugin that's built into webpack,**

we don't need to install it. What I'll do is I'll simply import webpack itself into that file,



so webpack via require webpack and then we can use one of the build in plugins. So down in the plugins section here, I'll add this plugin, here I'll add new webpack, optimize and there we have the uglify js plugin.

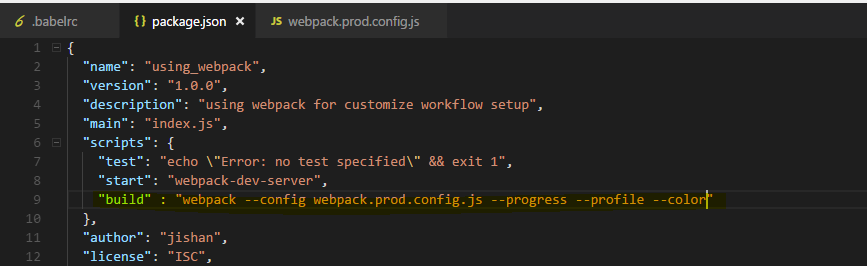


This plugin now also can be configured and you can always check out the official github pages of all these plugins and loaders to do so, I'll take the default setup though

**and with that I got my production workflow where I also optimize the files, where I minify them to ship as little code as possible.**

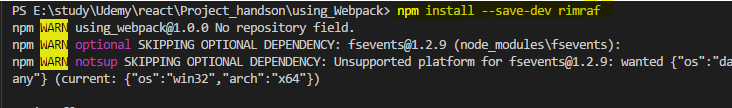
**Now to use this configuration, I need to adjust my package.json, the build scripts here where I point at webpack, I don't want to use the default config instead I want to use a special one, so I'll use --config, that's a flag we can pass to the webpack script and there I'll point to webpack.prod.config.js file since this is in the same folder as the package.json file,**

**I don't need to specify any path, I can just specify the filename. With that I'll also add some additional options, progress, profile and colors to adjust the console output of the webpack build workflow**

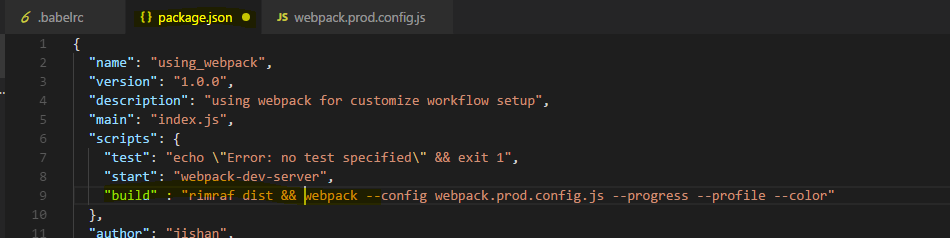


I will install one additional package with npm install --save-dev,

**it's called rimraf and it allows us to delete a folder or files and I want to delete the dist folder at the start of every build process so that we create a brand new one**



So I'll execute rimraf dist first and then two && signs to then run this project or this command here after the deletion is done. With that, let's run build again, npm run build and let's see if this gives us a nice output,



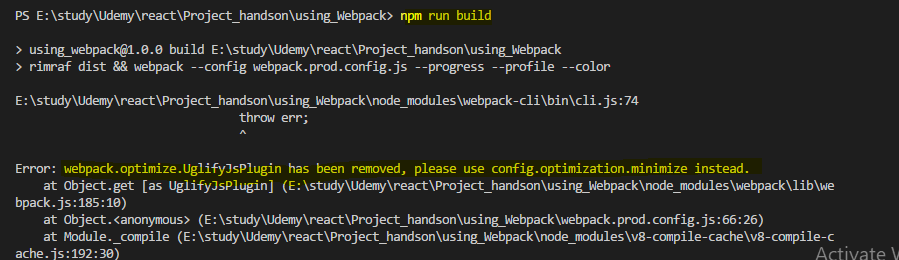
**Now with that, it's building the project we can see it here, it should succeed and it does, it gives us the dist folder still but if we have a look at the bundle file now, you see this is minified code. It's a lot of code because it includes the complete react library but this is all minified and therefore optimized and the same of course in our 0.js file.**

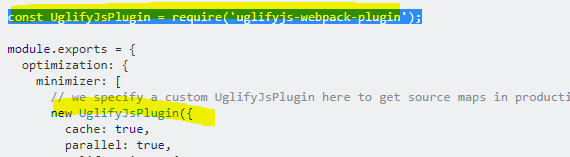


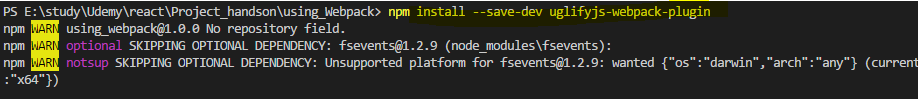
**So now we got our dist folder and this would be what we upload to a server, the content of the dist folder.**

### **Note : Error**

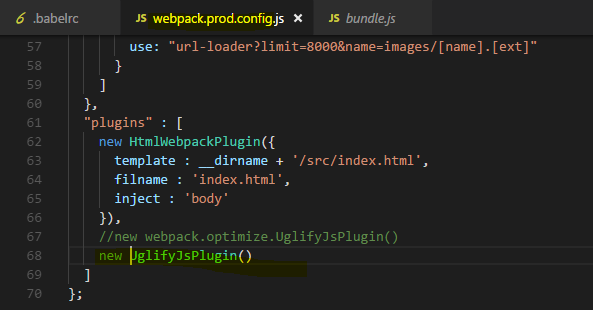
Because of latest versions



**Then try this , it will be resolved** 

So lets do this - 





**This is now our own production and dev workflow and I hope it became a bit clearer how webpack works and how you can set up such a custom workflow.**

## **Wrap Up**

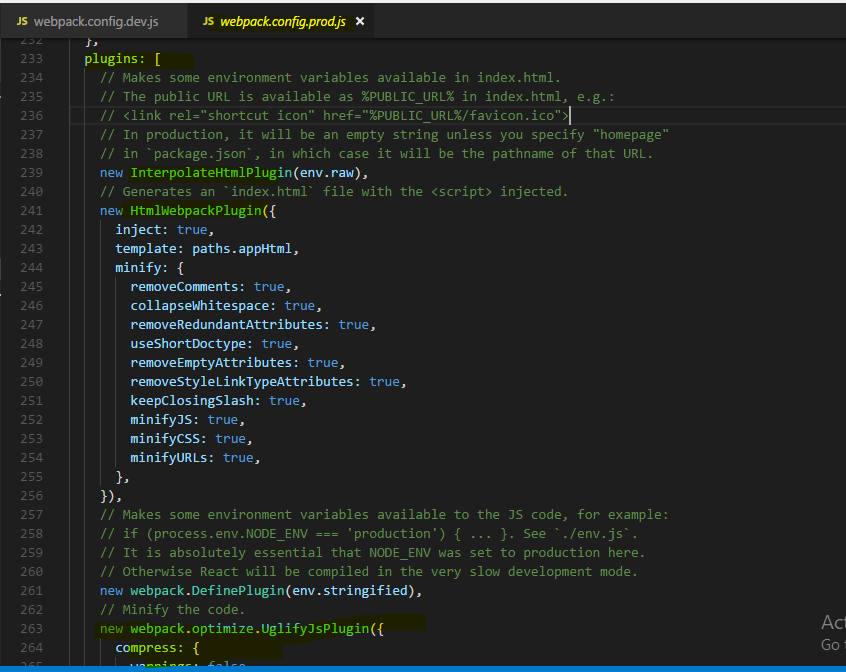
So over the course of this module, we built our own basic project setup with webpack as a build tool.

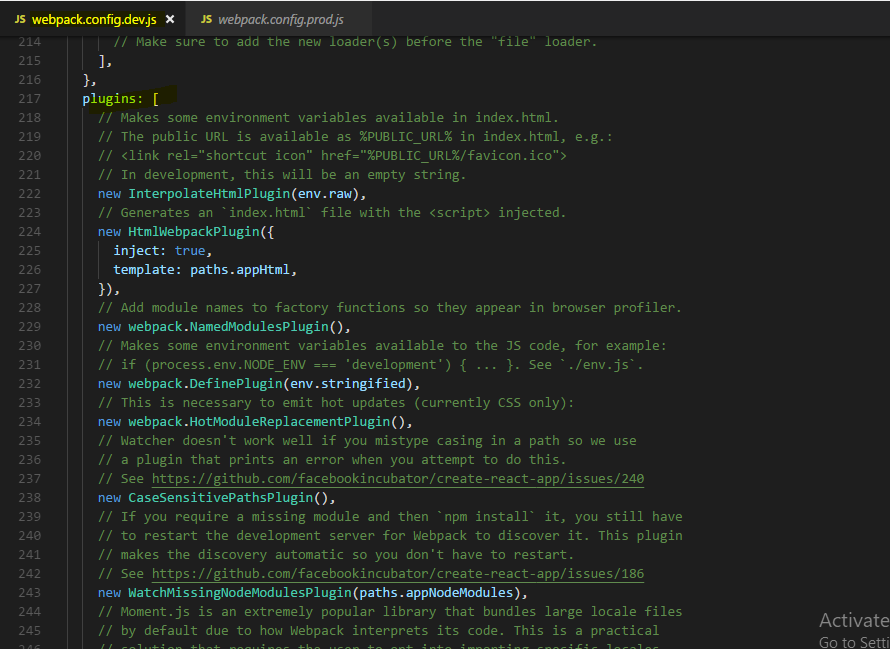
Now you can go into way more detail, there's a lot you can configure,

you can add more loaders, more plugins and fine tune everything to your needs.

Definitely dive into the e-check, that configuration we have for create react app, there you can have a look at the webpack config they used when setting up the create react app config though it's a very elaborate and therefore complex config.

This is a simpler one because I want you to understand how webpack works and what it does and I hope this became a bit clearer over this module. Feel feel free to adjust and fine tune this Configuration and take it as a starting point for your next projects, always dive into the github pages where you can find the documentations of the various loaders and plugins to really configure everything as you need it.

Create react app , webpack config for reference - 



## **Adding babel-polyfill**

Adding babel-polyfill

The current setup won't support all browsers theoretically supported by React. Features like Promises and Object.assign()  are missing in older browsers - especially in IE of course.

If you need to support these browsers, you need to add a polyfill (a package which provides these features for older browsers). babel-polyfill  is a great and easy-to-use choice.

Add it like this:

npm install --save babel-polyfill

Add the following import to the top of your index.js file:

1. import 'babel-polyfill';

Change the config of your env  babel preset in the .babelrc  file:

1. "presets": [
2. ["env", {
3. "targets": {
4. "browsers": [
5. "> 1%",
6. "last 2 versions"
7. ]
8. },
9. "useBuiltIns": "entry"
10. }],
11. "stage-2",
12. "react"
13. ],

useBuiltIns  was added and by setting it to 'entry' , the import in the index.js  file (import 'babel-polyfill' ) is actually changed to import whatever features need to be supported for your chosen browsers and environment. More information can be found here: <https://github.com/babel/babel-preset-env#usebuiltins-entry>

## **Useful Resources & Links**

* Webpack Docs: <https://webpack.js.org/concepts/>
* More about Babel: <https://babeljs.io/>

# **html-webpack-plugin**

<https://medium.com/a-beginners-guide-for-webpack-2/index-html-using-html-webpack-plugin-85eabdb73474>

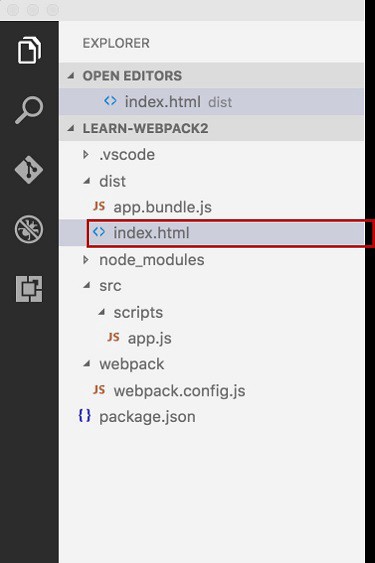
**using html-webpack-plugin to generate index.html**

Now that we got webpack configured to package and bundle our js code in a single file -app.bundle.js, we need an **index.html** for our web app with a script tag havingsrc='app.bundle.js'.

We have two options - either create it manually or have it created automatically by using ‘**html-webpack-plugin**’

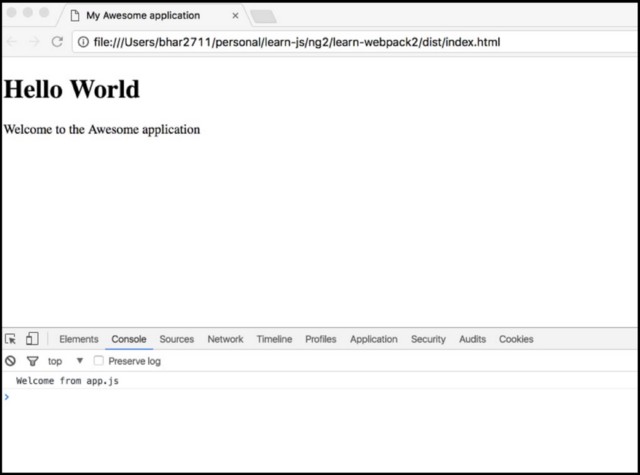
## **Option 1: Create index.html manually**

Lets first create the index.html file manually. Inside the ‘dist‘ folder, where webpack outputs the app.bundle.js file, add a file ‘**index.html**’ with the required script tag, as shown below:



Add below code to the above created index.html

If you open the index.html in the browser, you should see our javascript code working:



(click on image to see in expanded popover view)



## **Option 2: Using webpack-html-plugin to create index.html**

While creating index.html manually works good, it would be nice if webpack can create the index.html automatically for us with an included <script> tag with its src pointing to our app.bundle.js.

Webpack can do this for us with the help of html-webpack-plugin. Using this plugin has some added advantages like auto-hashing the **‘src’** attribute of the embedded <script> tag every time the webpack is run, which makes browser to get the latest version of the file from server instead of using a cached one whenever it has a new hash.

Lets see how to setup the plugin –  
1. First, remove the manually created index.html that we created in the above section.

2. Install **html-webpack-plugin**:

$ npm install html-webpack-plugin

3. Configure the webpack.config.js to use the plugin :

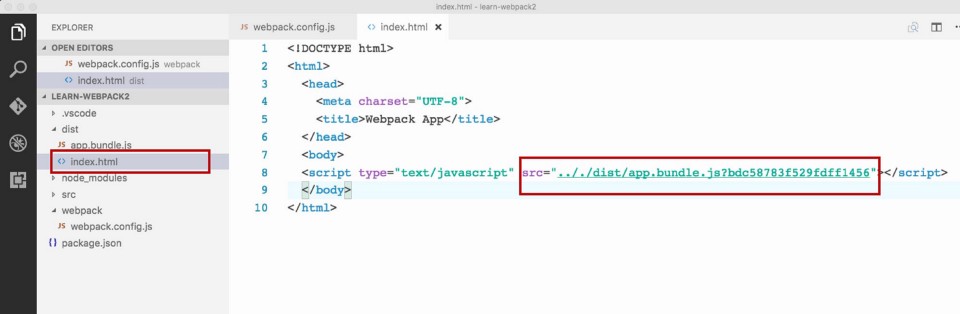
In the webpack.config.js, ‘require’ the plugin and to setup the options for the plugin, we need to add a plugins property in the configuration as shown below:

Here we added a new **HtmlWebpackPlugin**object in the **plugins** array of webpack.config object. Notice the **hash** option has been set to true which will add a unique hash to the src of the embedded <script> tag.

4. Now run npm start command for webpack to rebuild the package.

npm start

5. This time we would see 'html-webpack-plugin' automatically created an index.html for us, with an included <script> tag with its src pointing to the generated app.bundle.js.



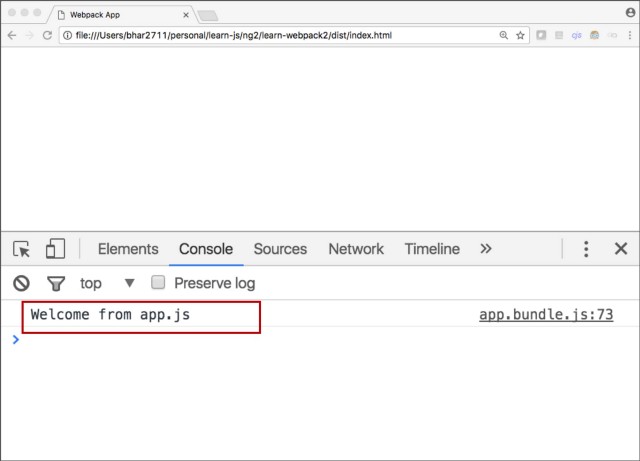
(click on image to see in expanded popover view)



## **Using custom template for generating index.html**

Great, so now we got **webpack** create **index.html** for us using the **HtmlWebpackPlugin**.

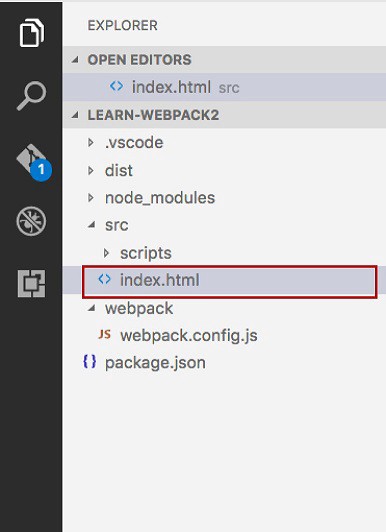
However, if you noticed in the screenshot above, the **index.html** generated by **HtmlWebpackPlugin**, is very basic, just embedding the <script> tag. Its lacking any other HTML content, including our custom title, the page header and the welcome message that we have added when we created the file manually in the above section. If we look at the index.html file, **HtmlWebpackPlugin** created in the browser, presently, except our js code working, it will look blank:



(click on image to see in expanded popover view)

If we want custom HTML content in the **HtmlWebpackPlugin** generated index.html, we can create a template html file and tell the plugin to use it as a template in its options as below:

1. Create a template HTML file — inside our src folder, create a file index.html:



(click on image to see in expanded popover view)

2. Add below code to src/index.html:

Notice the **placeholder elements** **<%= htmlWebpackPlugin.options.title %>**and **<%= htmlWebpackPlugin.options.myPageHeader %>** we have in the above html file which we would set as template to HtmlWebpackPlugin for generating index.html. We will provide the values for these placeholder options in the plugin’s configuration options webpack.config.js.

3. Now, in the webpack.config.js file, update the configuration of HtmlWebpackPlugin to use ‘template‘ property as below:

Notice the new options we added to the configuration object of **HtmlWebpackPlugin**

* **template: './src/index.html'** – this option specifies which file to use as template for the index.html being created.
* **title: 'My Awesome application'** – this is a custom property we added to the config object, the value of this property would be used to embed in the placeholder element <%= htmlWebpackPlugin.options.title %> we have in our template html file.
* **myPageHeader: 'Hello World'** – this is another custom property we added to the config object, the value of this property would be used to embed in the placeholder element **<%= htmlWebpackPlugin.options.myPageHeader %>**we have in our template html file.

3. Run **npm start** command for webpack to rebuild the package.

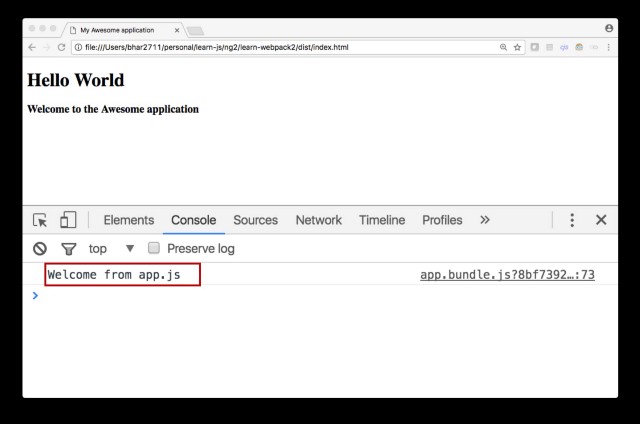
**npm start**

4. This time html-webpack-plugin would use the **index.html** we created under src folder as template and thus the generated **index.html** in the ‘dist’ folder would contain our custom HTML in addition to the embedded **<script>** tag.



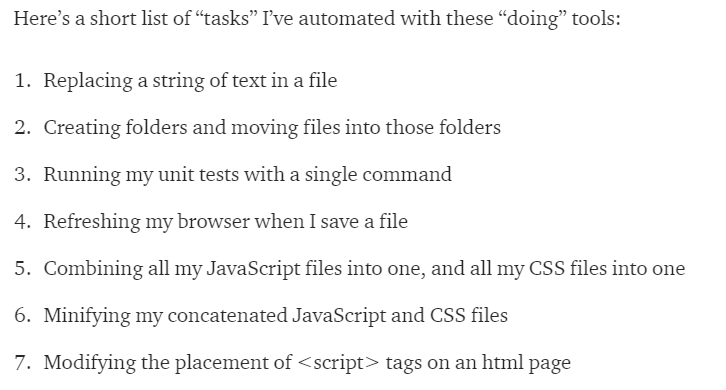
(click on image to see in expanded popover view)

5. And below is how the index.html would now look in the browser:



# **Also refer this explanation**

<https://medium.freecodecamp.org/making-sense-of-front-end-build-tools-3a1b3a87043b>



## **A build is just a production ready version of your app**

But when it’s time to move your app to production, having multiple JavaScript or CSS files isn’t ideal. When a user visits your site, each of your files will require an additional HTTP requests, making your site slower to load.

So to remedy this, you can create a “build” of our app, which merges all your CSS files into one file, and does the same with your JavaScript. This way, you minimize the number and size of files the user gets. To create this “build,” you use a “build tool.”

Below is a screenshot of an app in development. Notice how it has 5 <script> tags and 3 <link> tags? If you look on the left side, notice the DEVELOPMENT folder has 10 files inside of it?

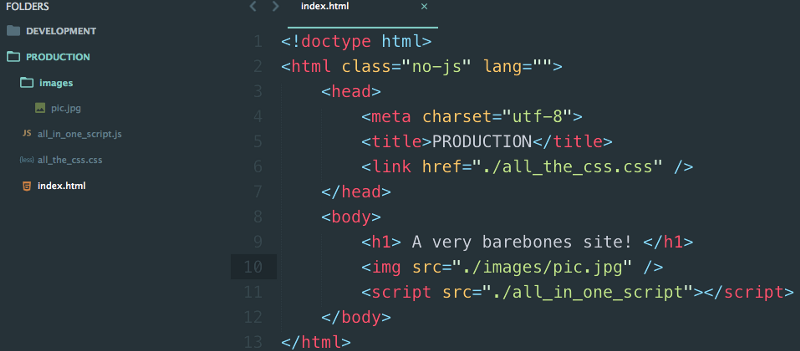


Your app in development

And below here is the same app after a build tool has worked its magic.

Notice how we just have a single script tag and a single link tag? And now the PRODUCTION folder has just 4 files, compared to the DEVELOPMENT folder’s 10.

The app is line for line the same. We’ve just compacted it into a neat small package we call a “build.”



Your app in its build form

You might wonder why a build is even worth it, if all it does is save your users a few milliseconds of load time. Well, if you’re making a site just for yourself or a few other people, you don’t have to bother with this. Generating a build of your project is only necessary for high traffic sites (or sites that you hope will be high traffic soon 😎).

Using some combination of tools like these on top of Node+npm lets you automate a lot of tasks in your project. The price you pay is that these tools have a steep learning curve.

