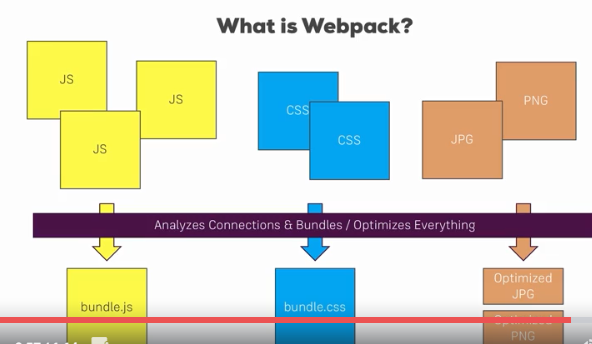
# **Webpack**

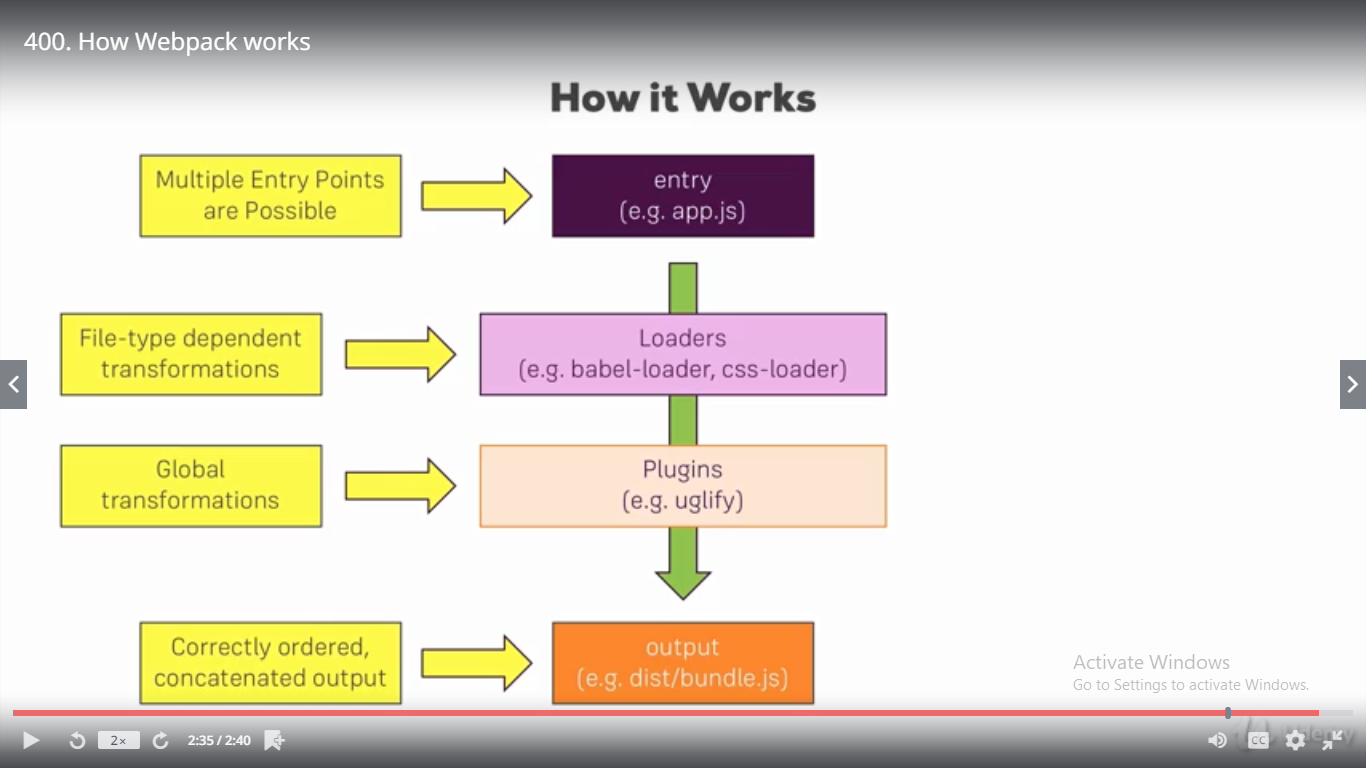
**Webpack is a bundler but it actually is more than that, a bundler alone would just concatenate files, webpack does that but it also allows you to optimize your files and you hook in various plugins and so-called loaders to also transform your files and for example transpile next generation javascript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





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

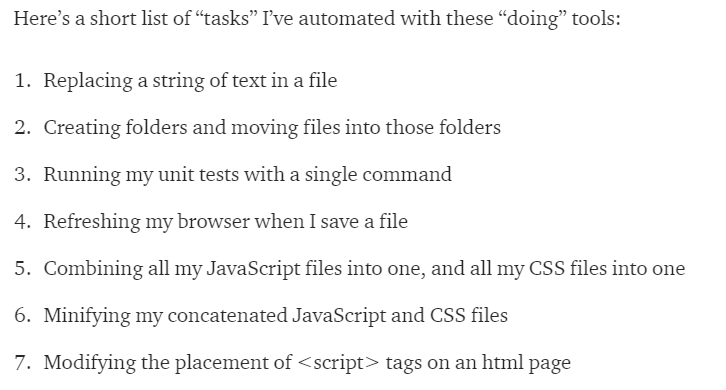
**loaders are applied on a per file level so we can for example say all javascript files should get handled by loaderX, all css files should get handled by loaderY, babel-loader and css-loader are two popular examples which get used in a lot of projects, so loaders are file dependent or apply file dependent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

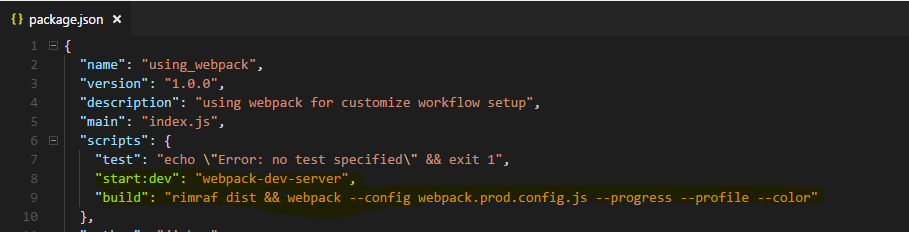




**we install new dependencies and we mark them as development only dependencies ,it makes it clear which dependencies we use only for setting up the build workflow and which dependencies really have an impact on the running application,**

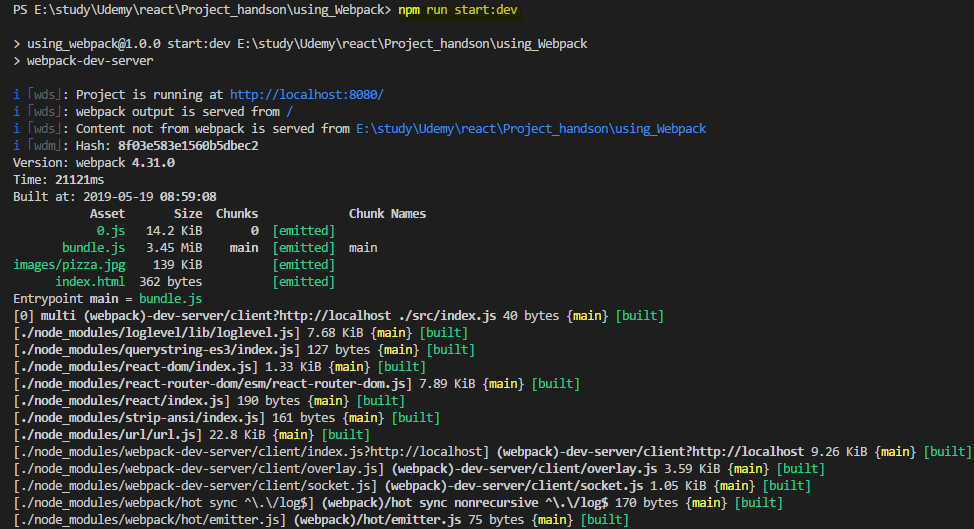


**Webpack-dev-server is this development server we want to use so that we can test our application locally on the machine and webpack is well basically the build tool itself, webpack-dev-server wraps this build tool, they're both from the same team.**



**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,**

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.**



**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,**

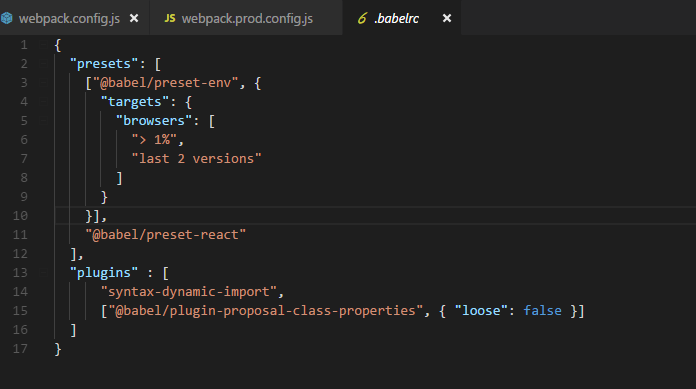
**we also import images and stuff like that so that's all the stuff we have to handle with loaders later.**

**cheap-module-eval-source-map : 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.**

**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.**

**Well for javascript, we will use the babel loader, babel is the defacto standard for transpiling next generation javascript to current gen javascript**



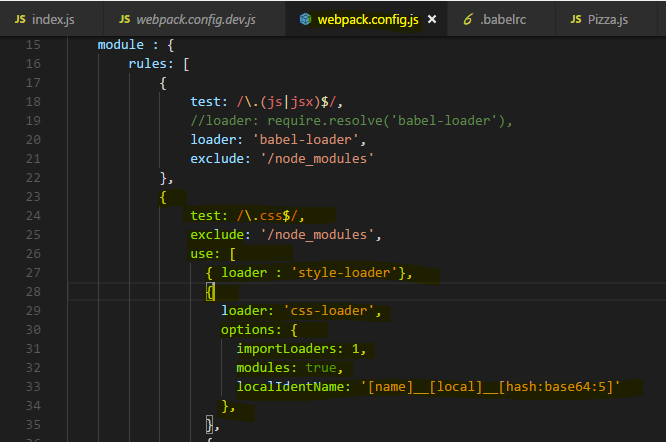
**we can make babel aware of all the presets we want to apply**

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

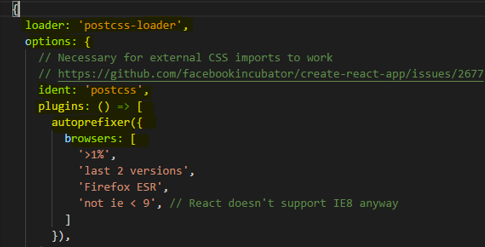
**Now for css, I actually need a more complex setup, for js we also have a quite complex set up but that's all handled through babel and our babel config,**

**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.**

**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.**



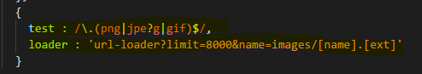
**postcss loader to handle sass and stuff like this**



## **Creating rules for Images**

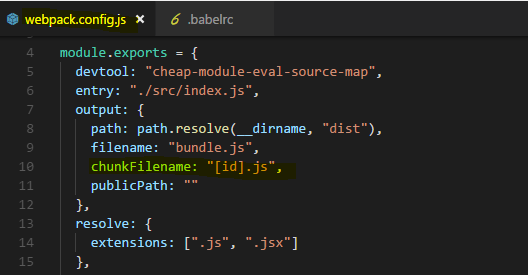
**URL loader is a loader which will take our images and if they are below a certain limit we define, it will actually convert them into data 64 URLs which it can inline into our documents, so we don't have to download extra file. But for bigger files, it would be inefficient so files above that limit 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.**

**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**

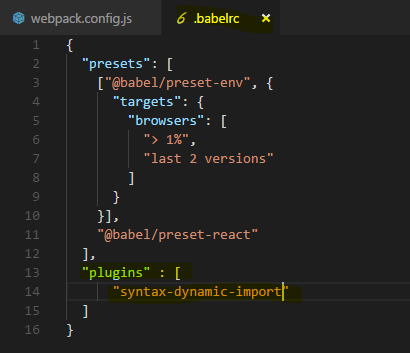


## **Lazy Loading support in webpack**

**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,**





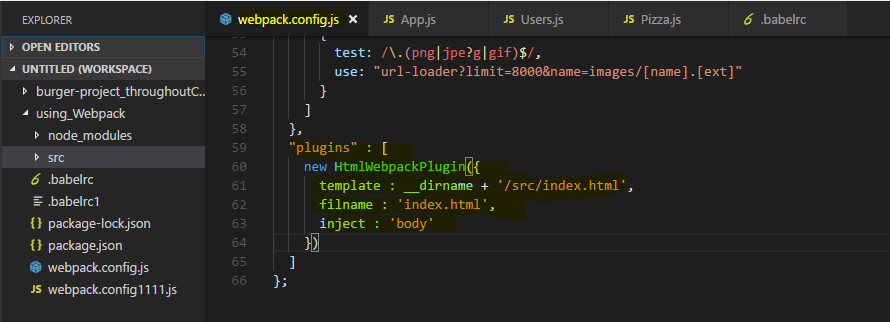


## **Plugins – Injecting the script into index.html file**



**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,**

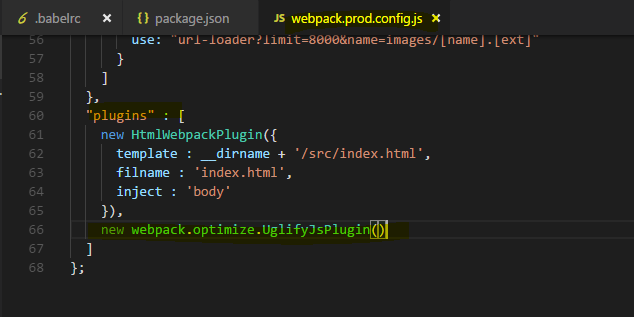
**later when we use a different workflow for building this, for production, it will give us the real files we could upload to a server.**



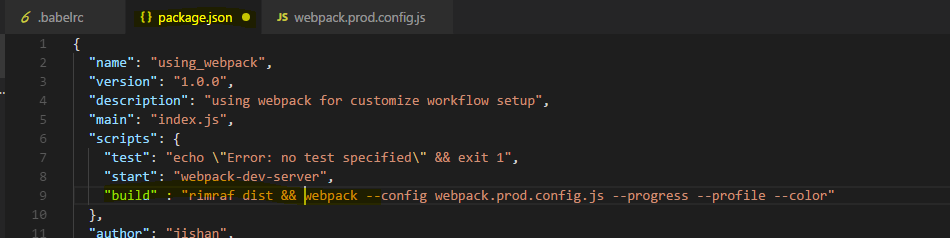
## **Creating a Production Workflow**

**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**

**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**



**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**



**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.**

Lets look in

**webpack.config.js**

const path = require("path");

const autoprefixer = require("autoprefixer");

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

devtool: "cheap-module-eval-source-map",

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js",

chunkFilename: "[id].js",

publicPath: ""

},

resolve: {

extensions: [".js", ".jsx", ".json"]

},

module: {

rules: [

{

test: /\.js$/,

use: "babel-loader",

exclude: /node\_modules/

},

{

test: /\.css$/,

exclude: /node\_modules/,

use: [

{ loader: "style-loader" },

{

loader: "css-loader",

options: {

importLoaders: 1,

modules: true,

localIdentName: "[name]\_\_[local]\_\_[hash:base64:5]"

}

},

{

loader: "postcss-loader",

options: {

ident: "postcss",

plugins: [

autoprefixer({

browsers: [

"> 1%",

"last 2 versions"

]

})

]

}

}

]

},

{

test: /\.(png|jpe?g|gif)$/,

use: "url-loader?limit=8000&name=images/[name].[ext]"

}

]

},

"plugins" : [

new HtmlWebpackPlugin({

template : \_\_dirname + '/src/index.html',

filname : 'index.html',

inject : 'body'

})

]

};

**Webpack.prod.config.js**

const path = require("path");

const autoprefixer = require("autoprefixer");

const HtmlWebpackPlugin = require('html-webpack-plugin');

const webpack = require('webpack');

const UglifyJsPlugin = require('uglifyjs-webpack-plugin');

module.exports = {

devtool: "cheap-module-source-map",

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js",

chunkFilename: "[id].js",

publicPath: ""

},

resolve: {

extensions: [".js", ".jsx", ".json"]

},

module: {

rules: [

{

test: /\.js$/,

use: "babel-loader",

exclude: /node\_modules/

},

{

test: /\.css$/,

exclude: /node\_modules/,

use: [

{ loader: "style-loader" },

{

loader: "css-loader",

options: {

importLoaders: 1,

modules: true,

localIdentName: "[name]\_\_[local]\_\_[hash:base64:5]"

}

},

{

loader: "postcss-loader",

options: {

ident: "postcss",

plugins: [

autoprefixer({

browsers: [

"> 1%",

"last 2 versions"

]

})

]

}

}

]

},

{

test: /\.(png|jpe?g|gif)$/,

use: "url-loader?limit=8000&name=images/[name].[ext]"

}

]

},

"plugins" : [

new HtmlWebpackPlugin({

template : \_\_dirname + '/src/index.html',

filname : 'index.html',

inject : 'body'

}),

//new webpack.optimize.UglifyJsPlugin()

new UglifyJsPlugin()

]

};