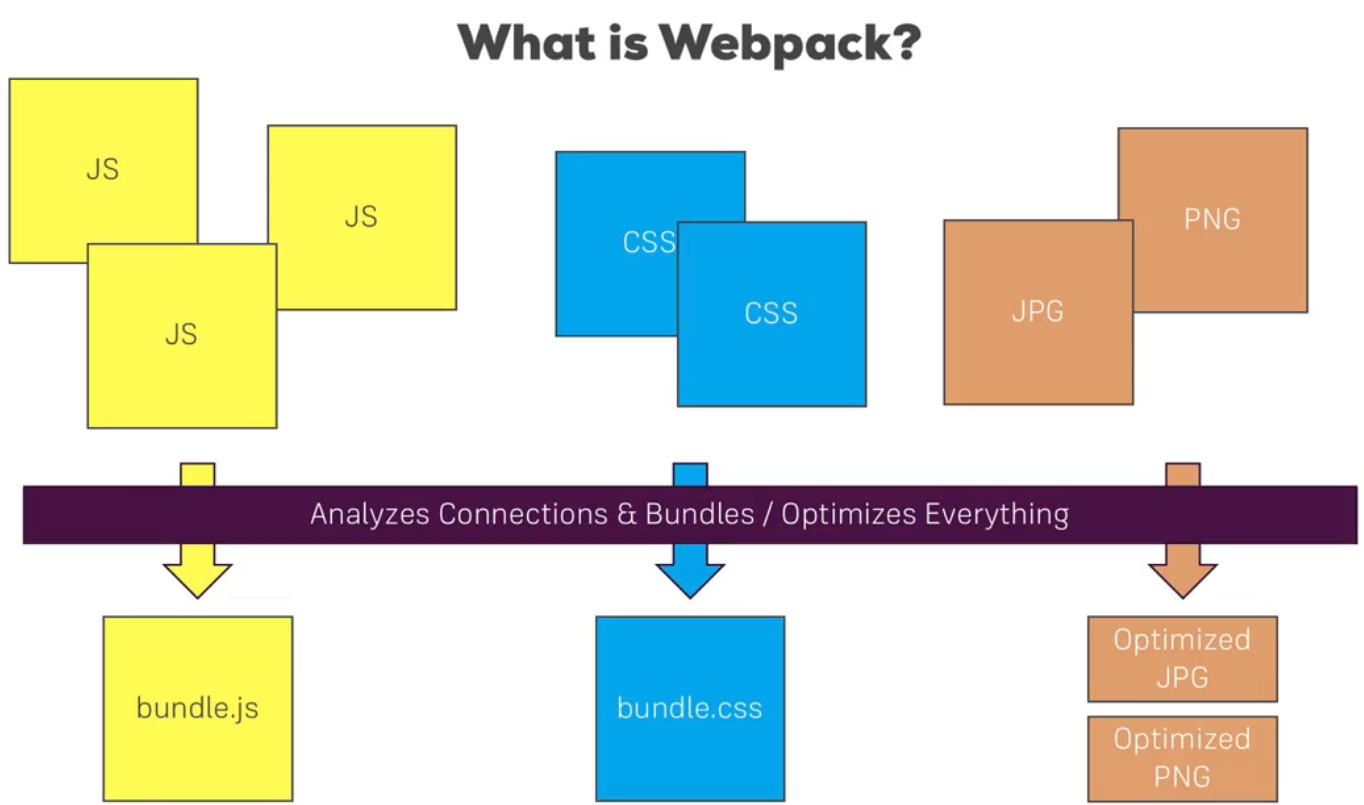
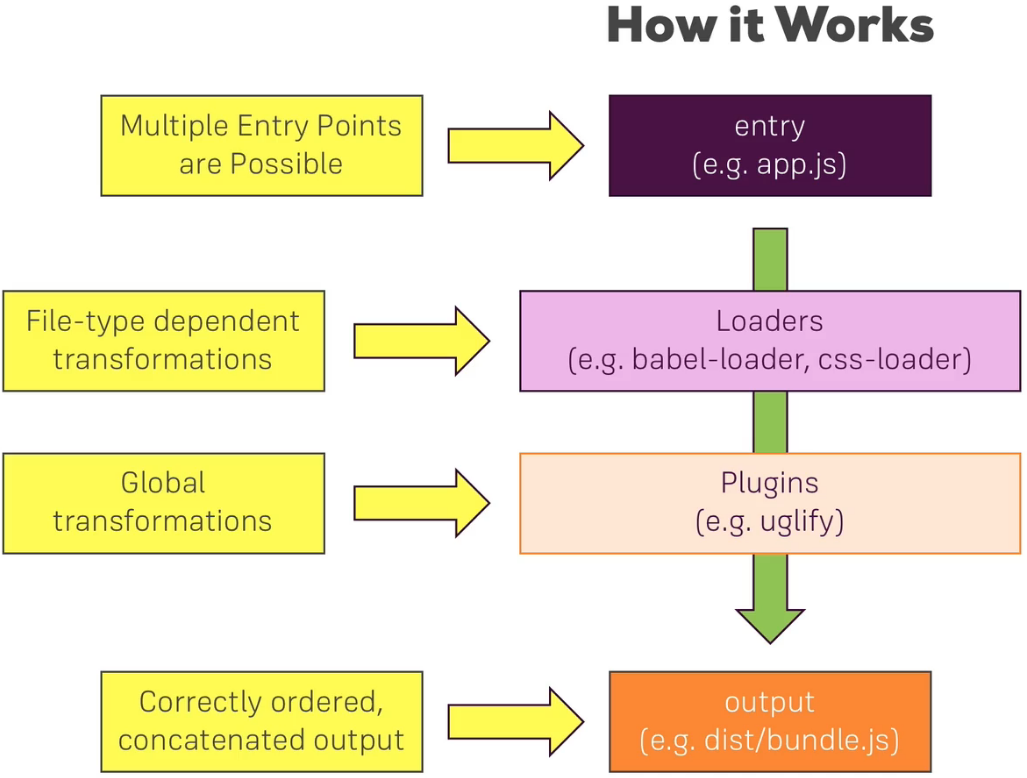
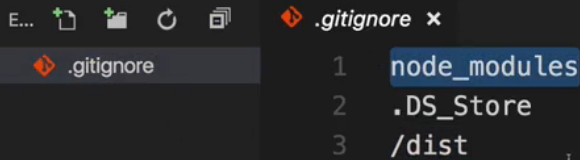
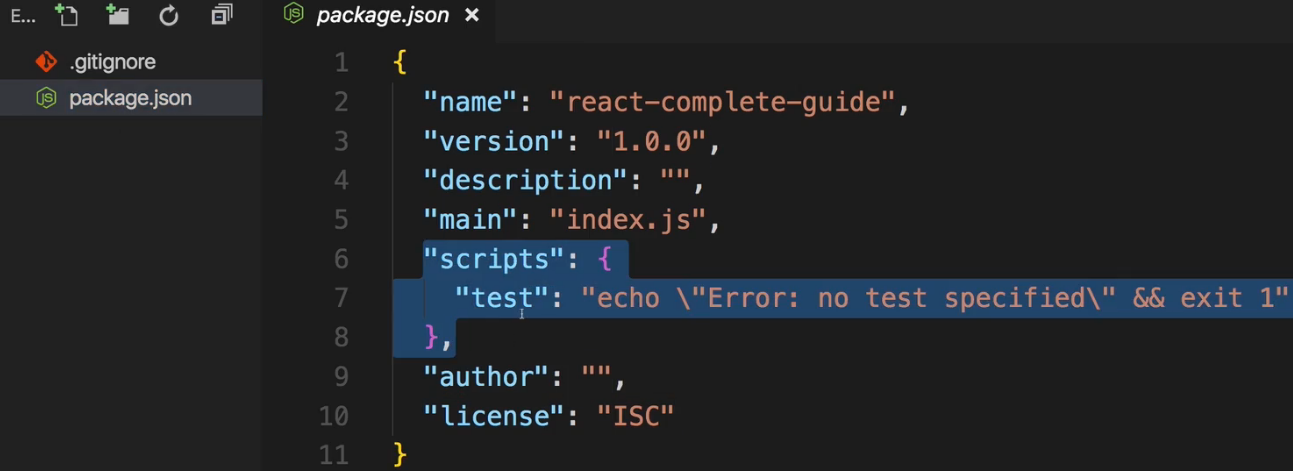
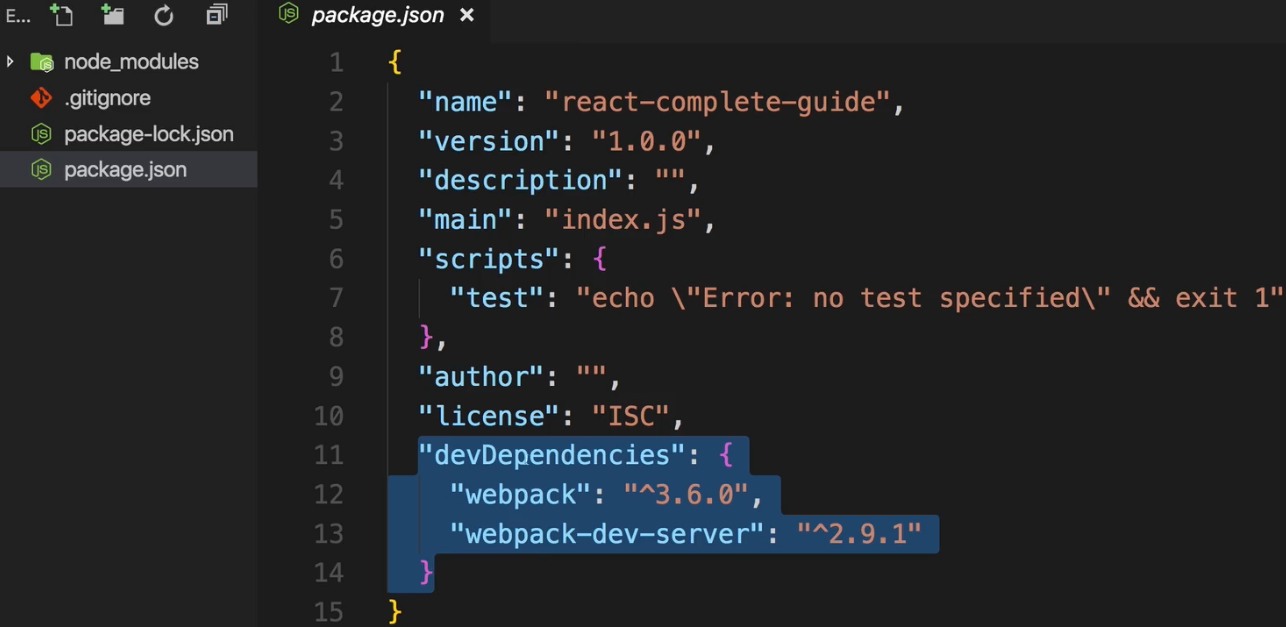
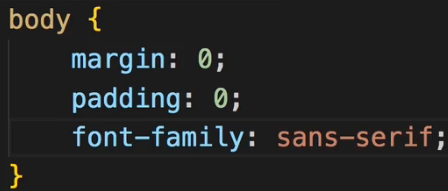
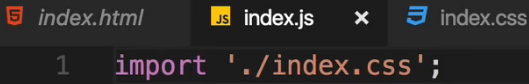
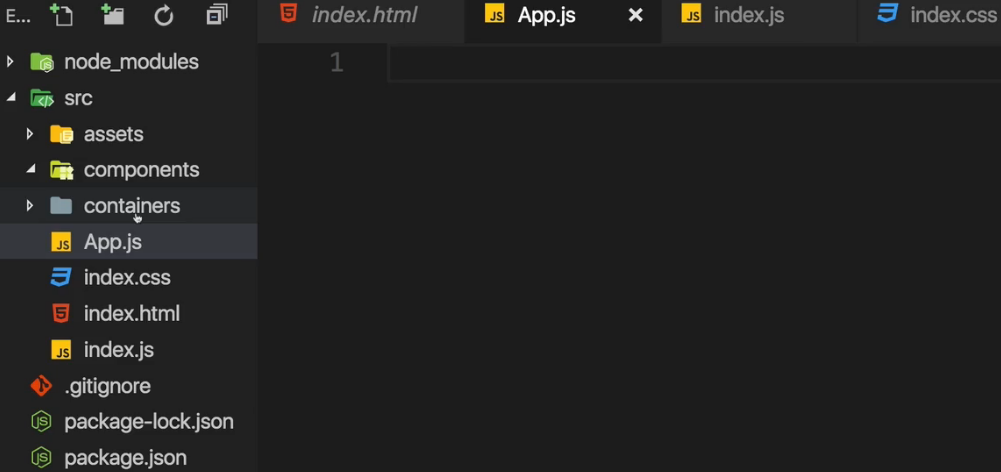
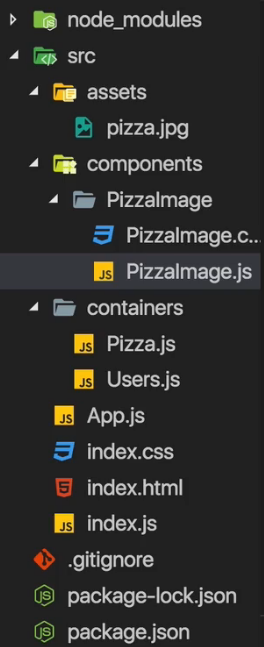
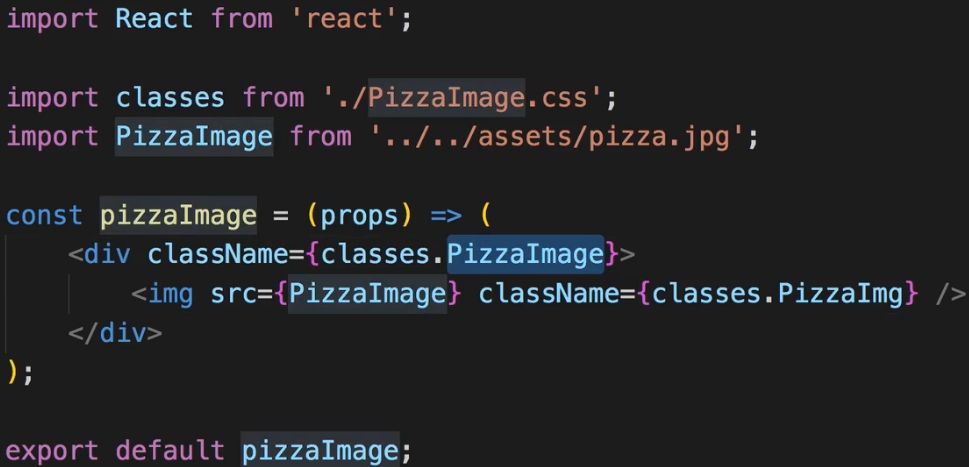
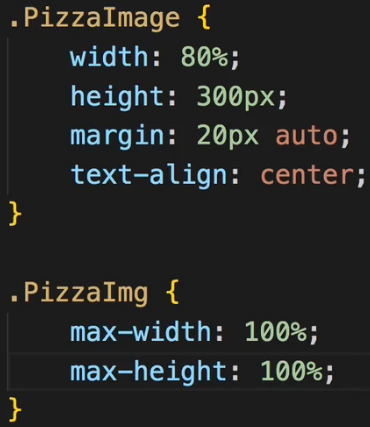
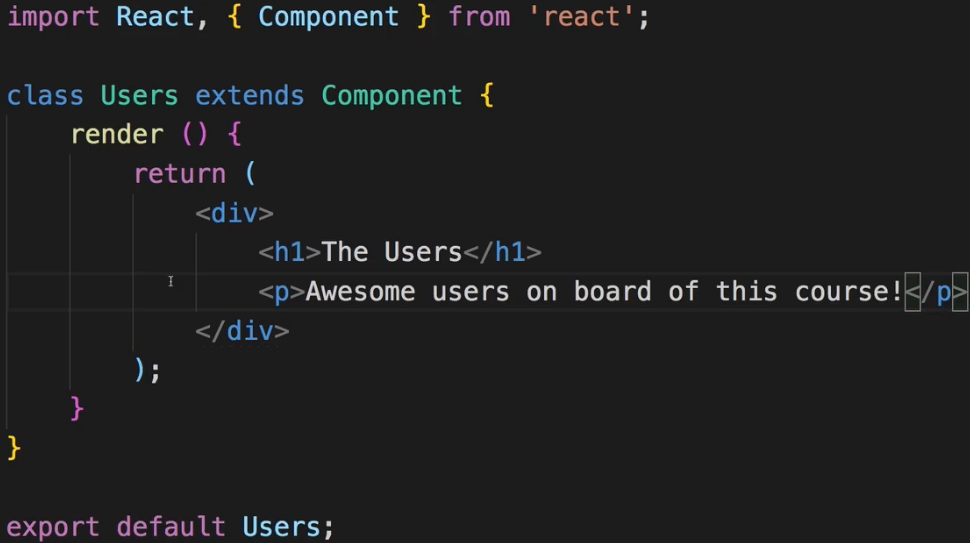
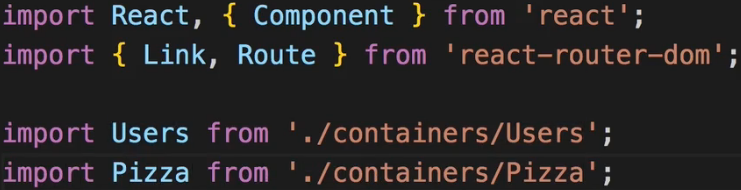
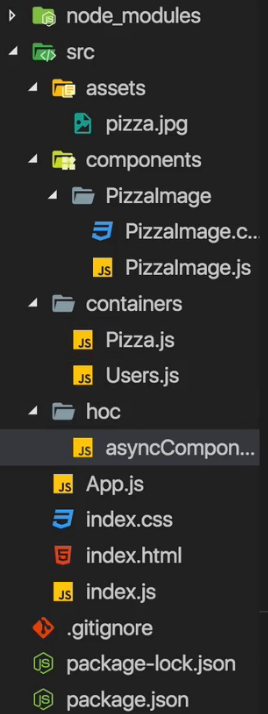
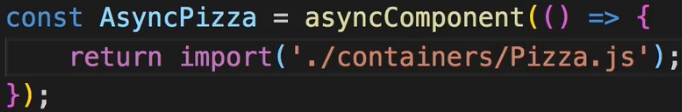
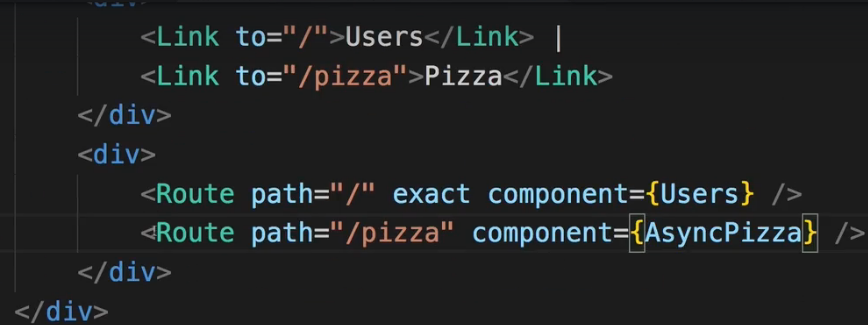
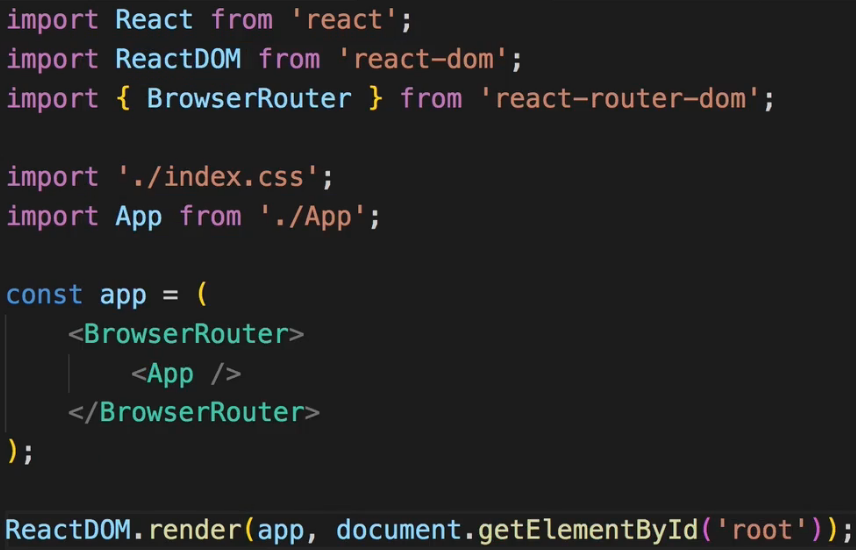
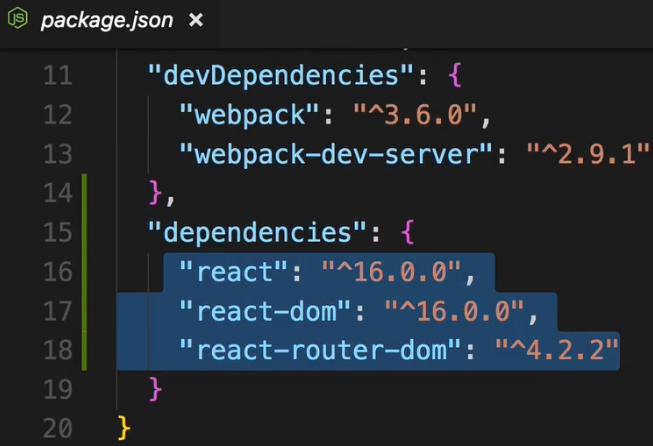
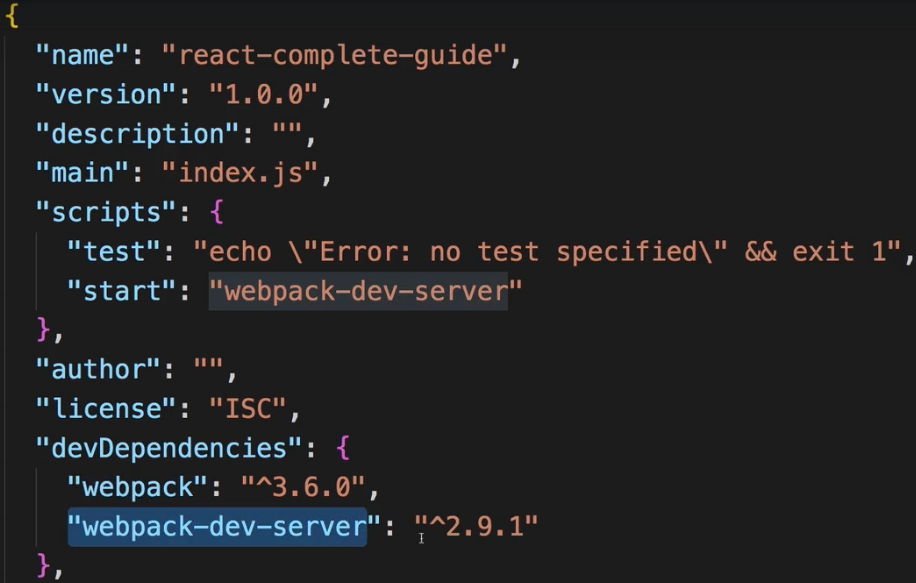
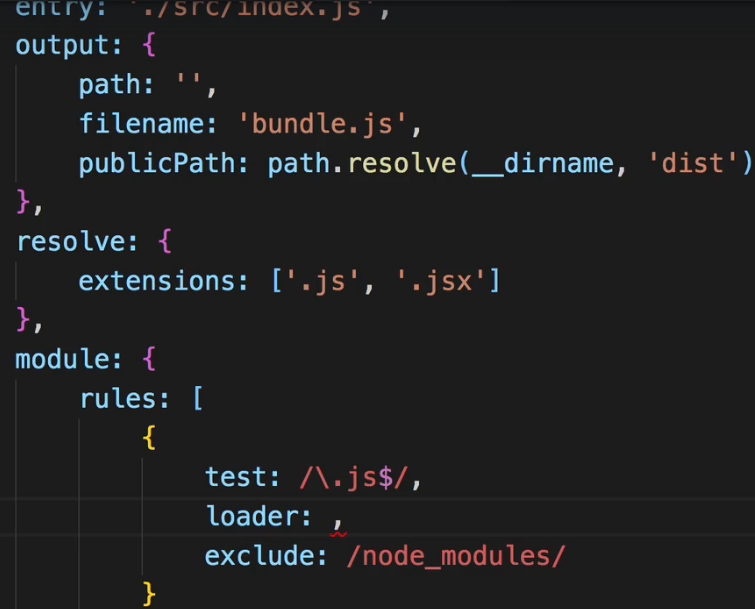
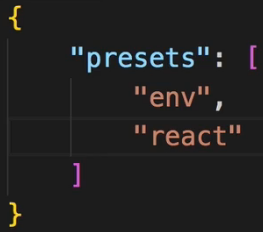
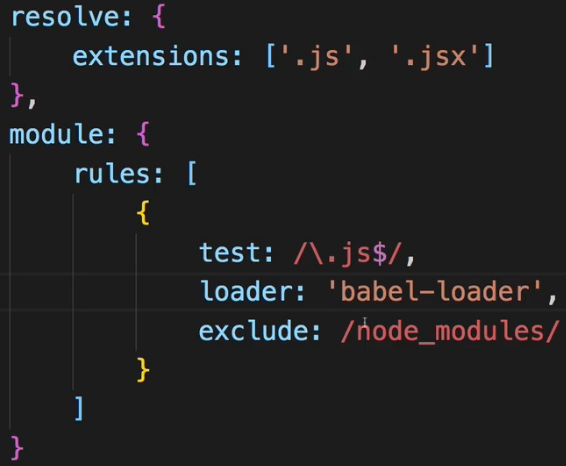
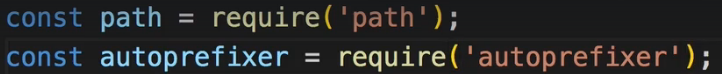
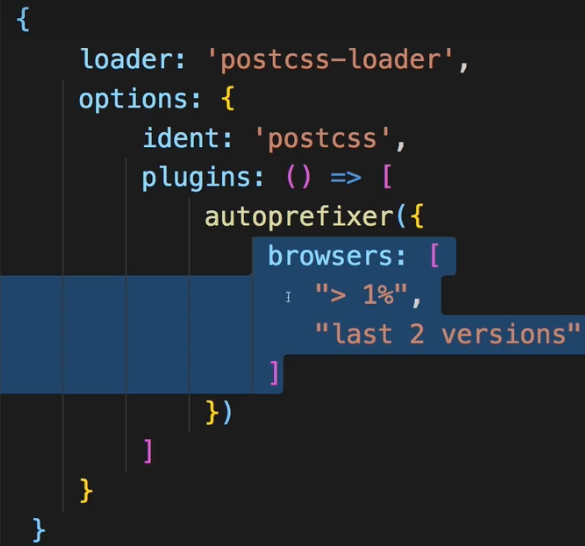
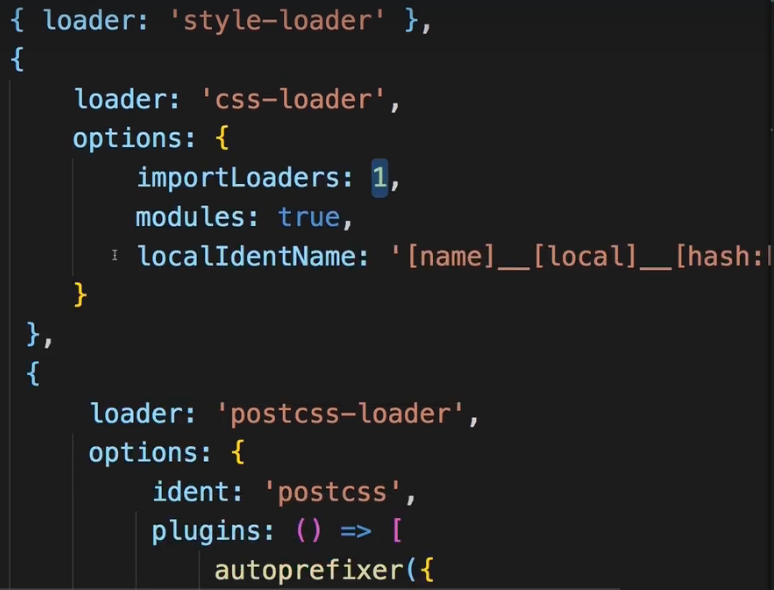
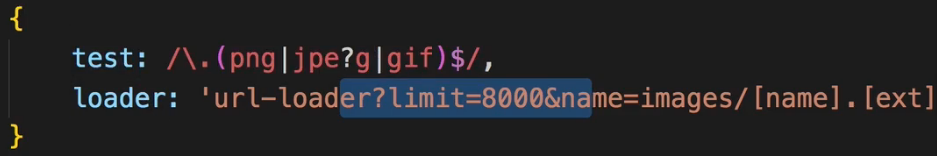
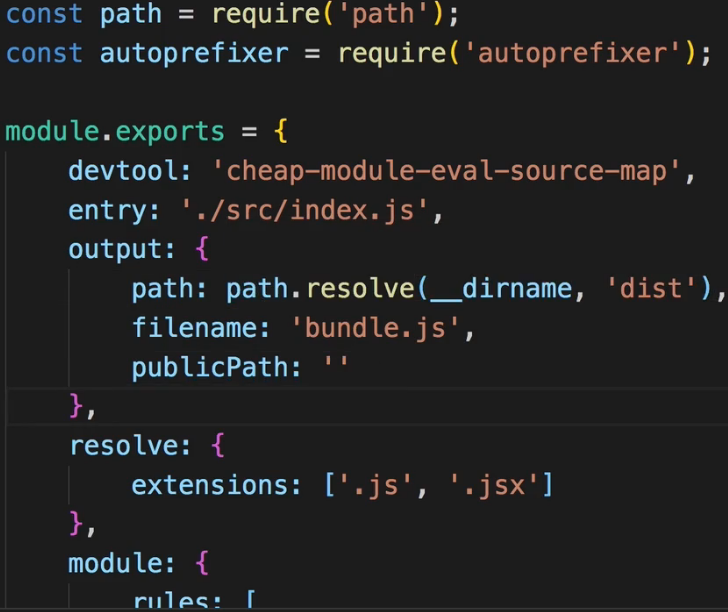
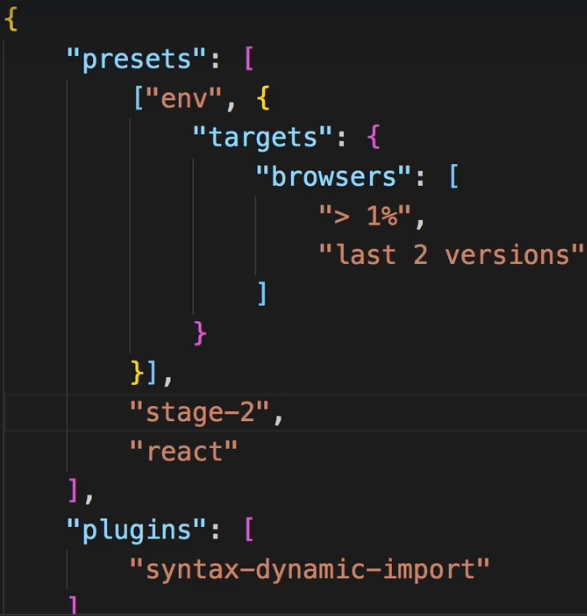
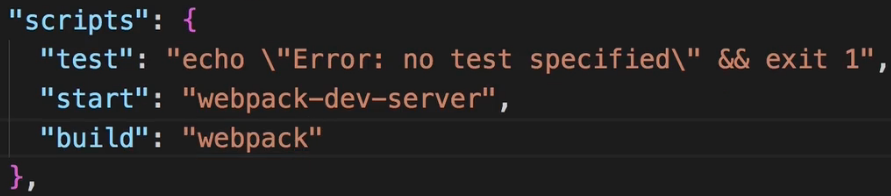
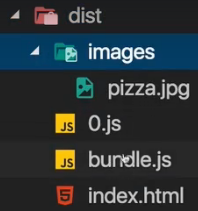
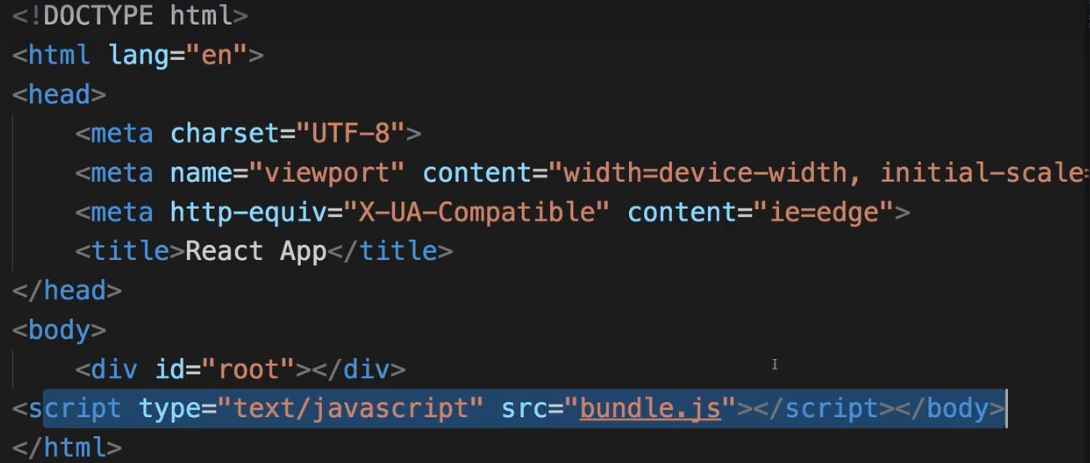
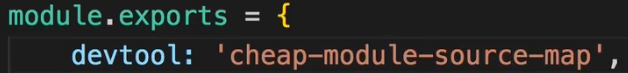
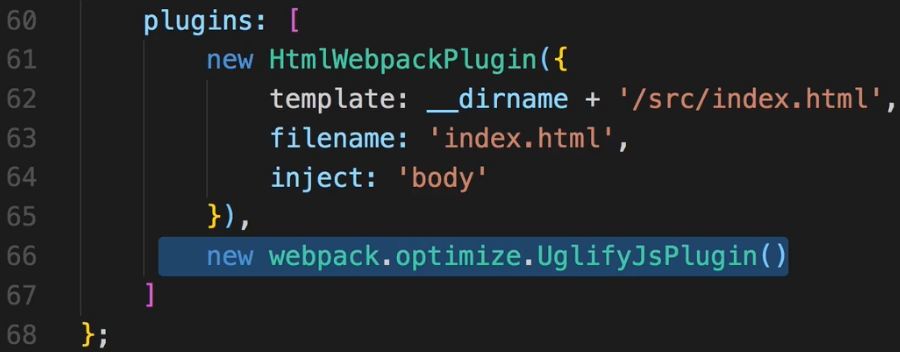
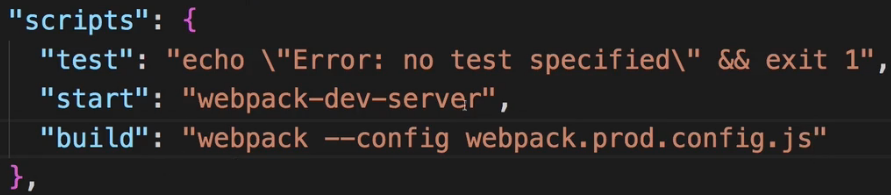
**Section 22 done: 20/20 Bonus: Working with Webpack**  
**Introduction**  
\* Webpack & Creating a Workflow.  
\* Throughout this course we used create-react-app and the project created with it to work in our React application, to build our React application.  
\* I strongly recommend using create-react-app, especially for SPAs.  
\* For Multi-Page Applications, creating multiple apps for the different pieces also is viable but there you might have your own build workflow deeply integrated into your whole Full Stack Project anyways.  
\* But for SPAs, create-creat-app gives you an awesome Workflow with a lot of next-generation features you can use, best practices when it comes to optimizing your code and much more.  
\* Still, to understand what’s happening behind the scenes and to get a deeper understanding of what the Workflow does for you, I want to set up a basic Workflow together with you in this module.  
\* I’ll introduce you to **Webpack**, the core Build Tool which is used behind the scenes and we’ll set up a project which actually supports the same code we wrote with the create-react-app application, in our setup here.  
\* Let’s start by understanding what exactly Webpack is since it plays a crucial role.  
**Important: Use Webpack 3**  
\* Important: When installing Webpack (we'll do that in one of the next lectures), run npm install --save-dev webpack@3  because the latest version (version 4) has a slightly different syntax.   
So for the same config & code as shown in the videos to work for you, you need version 3.  
If you want to update to Webpack 4.x, the following migration guide should help: <https://dev.to/flexdinesh/upgrade-to-webpack-4---5bc5>  
**Introducing Webpack**  
\* Webpack is a bundler but it actually is more than that.  
\* A **bundler** alone would just concatenate files.  
\* Webpack also allows you to optimize your files and you hook in various **plug-ins** and so called **loaders** to also transform your files and for example transpile next-generation JavaScript to current generation JavaScript.  
\* But in its core the idea behind Webpack is to have multiple JavaScript, CSS, JPG, whatever files and bundle them together. It analyzes connections between these files like imports and then bundles everything together, allows you to optimize it and allows you to also run some additional logic to transform your code or do whatever you need to do with it.  
\* Webpack is the de facto standard for setting up projects these days.  


**How Webpack works**  
\* Behind the scenes Webpack has 4 important features.  
**=> Entry Point =>** It always needs at least 1 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 ReactDOM.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 in turn has more dependencies. So Webpack can build up a dependency graph, starting with that root entry file so that it can understand which files make up our application if we give it an entry file.   
**=> Correctly ordered, concatenated output =>** It analyzes all the dependencies and bundles them together into an output we specify, like a bundle.js file in a `dist` folder, so we specify the filename and where it should go. And there it’ll all these dependencies into that file, correctly ordered and in 1 concatenated output file.  
\* This is the core functionality but it’s more than that.  
\* Inbetween there are 2 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 Loader X or CSS files should get handled by Loader Y, babel-loader and css-loader are two popular examples which get used in a lot of projects. So Loaders apply file-dependent transformations.  
**=> Plug-ins =>** Where Loaders are applied on a per file bases, plug-ins then take the concatenated file, 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 which we will set up together in this module.  
**Basic Workflow Requirements**  
\* 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-Gen 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 Autoprefixing** so that we can write simple CSS rules and get automatic prefixing so that it supports the best syntax in all browsers our application supports.  
=> And we also want to **support images** of course, we should be able to import images as we did it in the create-react-app and then use it in our template.  
=> Finally, we want to **optimize the code**, before we build 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 create-react-app gives us, though that also gives us more Linting and so on, but these are the core requirements I also want to build in this custom workflow.  
**Project & npm Setup**  
\* We’ll start with .gitignore file only.  
  
\* Now the first thing is 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**  
\* Now a package.json file has been added by npm. This is the file which we’ll use to set up our build scripts and to also manage the dependencies of our project.  
  
\* Let’s install the 2 important dependencies:  
**npm install --save-dev webpack webpack-dev-server**  
=> With --save-dev we install new dependencies and we mark them as development-only dependencies which doesn’t really have a big impact here, doesn’t make a difference, we could also use --save in our project, but 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’ll see that they’ll be grouped differently in 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.  
\* The second one is the development server we want to use so that we can test our application locally on this machine.  
  
\* Now anyone can just run **npm install** to install all the dependencies and the node\_modules folder gets re-created.  
\* A good place to start though is to not start working on the Workflow but let’s first set up a basic folder structure and a basic React app so that we know in which environment we want to work and that we can then adjust the Webpack configuration to that environment.  
**Creating a Basic Folder & File Structure**  
\* I’ll create a fairly similar folder structure to create-react-app.  
  
\* If you have any imports from CDNs of fonts or styles, you can add that in the <head> section in index.html.  
\* 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 setup.  
\* To the <body> I’ll add a DIV with a specific ID: root. This will be the DIV where we’ll later mount our React application to.  
\* In `assets` folder we can store images or anything like that we use.  
\* I also want to set up a `components` folder and a `containers` folder so basically the same setup I used in my 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 `src` folder and an index.css file which I want to use for some global styling.  
   
  
\* 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.  
**Creating the Basic React Application**  
\* I plan on using CSS Modules here.  
\* In PizzaImage folder:  
    
\* Now you might notice that React of course isn’t installed yet, we’ll do this in the next lecture where we install all the dependencies we need.  
\* My build Workflow should essentially copy that file to a new destination then and give me a link to that file in this constant which I can therefore output here.  
\* I want to center any text and since <img> is an inline style, it should be centered for that reason.  
\* The <img> itself I’ll set max-width and max-height to 100% so that it respects the boundaries of the container while still keeping the aspect ratio.  
\* In Pizza.js container:  
\* In Users.js container:  
   
\* Now I want to load the containers via Routing.  
\* In App.js:  
\* The | is there just to visually separate it.  
  
\* The “/pizza” Route should now be Lazy Loaded.  
  
\* So for **Lazy Loading** let’s create a `hoc` folder and **asyncComponent.js** file.  
  
  
  
\* By the way all the file extensions .js are missing because I want to have the same setup as in create-react-app where it’s automatically adding these file extensions.  
  
  
  
\* This is my app setup thus far.  
\* Now of course we need to mount this app.  
\* For that I need to go to the index.js:  
  
**Installing Production Dependencies**  
\* We’re going to use --save because we want to indicate that there are production dependencies.  
**npm install --save react react-dom react-router-dom**  
  
\* We already installed Webpack a while ago.  
\* We now need to install a couple of other dependencies to build a Workflow and I’ll add them step by step whenever we need them so I now want to start with setting up the Webpack configuration such that it is able to handle Next-Gen JavaScript, CSS, images and all that fun stuff.  
**Setting Up the Basic Webpack Config**  
\* We already installed the Webpack tool and I said that I’ll manage the Workflow through the “scripts” we have here which we then can run with `npm run scriptName` 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 here I want to simply execute webpack-dev-server, this has to been the name of that package, which simply exposes a function you execute with this call.  
  
=> So this will spin up the dev server.  
\* However, that server out of the box won’t work.  
=> We need to configure it so I’ll add a new file next to the package.json file, the **webpack.config.js**. This is a special filename, Webpack will automatically look for this file when executing either `webpack` or `webpack-dev-server` as we do here. So it’ll automatically take this file into account.  
\* Now in this file I’ll set up my DEVELOPMENT Workflow, we’ll later create a separate file for the PRODUCTION Workflow because there we’ll differ in some aspects, we’ll apply some extra optimization and so on.  
\* Here we need to export something which Webpack will then use, we do this with the NODE EXPORT format: **module.exports = {}**. So this is the syntax Webpack understands to read in the config we now specify in this JavaScript object. And I did mention the 4 important aspects of that configuration earlier on the slide.  
=> This tells Webpack that it should look at the index.js 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 files, we need to give Webpack some extra setup, we need to do this for JavaScript files too though since we use JSX which is not native JavaScript and since we use Next-Gen JavaScript features too. We also import images and stuff like that. So that’s all the stuff we’ll have to handle with Loaders laters.  
\* devtool: here we can define which kind of source maps - if any - Webpack should generate.  
=> The “cheap-module-eval-source-map” in the end is the best kind of source maps you can generate for development, very detailed, good performance, they allow you to easily debug your original code in the browser.  
=> `**publicPath**` is 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 the `publicPath` 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 folder structure will be the folder structure as we deploy it in the end.  
=> more important is the `**filename**`, here we define what the file should be named. It’s a convention to use bundle.js because it is a bundled JavaScript in the end. Now you might wonder what happens to CSS and images - these will be handled differently by certain Loaders we add and the Loader 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 assests be stored, and there I’ll actually import a Node Module which is called path, which I do with the Node import style where we 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 **path.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: **\_\_dirname**. And it refers to the directory this is run in. And then the 2nd argument to path.resolve() is the folder where we want to create it and path.resolve() will then create an absolute path, taking the full path to the current folder on our operating system and appending /dist.  
  
**!!! the path.resolve() goes to the path: we’ll fix it later.**  
\* So it will in the end output a file in a dist folder which it will create which is named bundle.js.  
**Adding File Rules**  
\* 1 important thing we have to add upfront is to make sure that Webpack automatically appends .js at the end of these imports here.  
=> We add the `resolve` configuration, that takes a JavaScript object {}. And there we can tell Webpack that it should be aware of certain extensions and if it encounters and import without an extension, it should try these extensions and see if it finds a file with one of these. So that’s an array of possible extensions. And I want to support .js and .jsx here.  
  
\* Now the next and most important thing are our Loaders where I want to make sure that we do correctly handle different file types.  
=> This is done in `**module**`. Because an import, a dependency is referred to as a module. So you could kind of translate module with file. Here we set up what Webpack should do with the individual files, and we control this with the `**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 which commonly check the file extension. I have to escape the . in .js otherwise it would mean any character. If it ends with .js, then I want to apply a certain `**loader**`. Now a `loader` is like a third-party plug-in which does something to the file. Now for .js files I also want to add an additional configuration which is `**exclude**` which allows me to exclude certain patterns. So it shouldn’t try to transform anything in node\_modules since these are third-party libraries we already do import, they are already optimized, it should only touch our own files.  
  
\* Now what about this `loader`?  
\* Well for JavaScript we’ll use the babel-loader. Babel is the de facto standard for transpiling Next-Gen JavaScript to Current-Gen JavaScript. Let’s install and configure it next.  
**Introducing Babel**  
**npm install --save-dev babel-loader babel-core babel-preset-react babel-preset-env**  
\* The babel-loader however just provides 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 the babel-core.  
\* We now also need some configurations to basically tell Babel which kind of Next-Gen JavaScript or which special JavaScript syntax we want to support.  
\* And here, these are so called presets, so we need the babel-present-react and babel-preset-env for environment adjusting preset which basically has a look at the features we use and automatically transpiles everything correctly.  
=> We need to add a new file **.babelrc**.  
=> That’s a config file Babel will automatically read if it’s in our root folder. It takes a JSON config.  
=> There we have `**presets**` node. Now we can make Babel aware of all the presets we wanna apply.  
  
\* We use “**env**” and “**react**” presets which we installed.  
\* Now the “env” preset will receive some special configuration from my side.  
=> I do this by enclosing it into an array and then the second argument after the preset is a JavaScript object {} where I can configure it. `**targets**` means which browser versions we want to support. It automatically identifies which Next-Gen Features I use but to know what Next-Gen actualy means, it needs to know what my Current-Gen is, so which browsers do I target with my app. There we can set up `**browser**`, that’s an array where we can use the browser list type of configuration, you can google for browserlist to find a github repo to learn more about the syntax you can use for configuring browsers. The syntax I’ll show you here is a pretty common setup. So we want to target > 1% market share or last 2 versions.  
  
\* Now in webpack.config.js:  
  
\* So now it will be applied to JavaScript files compiling the JSX and Next-Gen 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.js component we need to support the CSS and an image.  
**Adding CSS File Support**  
\* Let’s add a new rule in the `module` `rules`.  
\* Now for CSS I actually need a more complex setup.  
\* For JS we also have a quite complex setup but that’s all handled through Babel in our .babelrc config, for CSS we’ll configure it here in the Webpack config file.  
=> So instead of the `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 loaders we want to apply. I want to install 2 loaders.  
**npm install --save-dev css-loader style-loader**  
\* The css-loader basically tells Webpack what to do with these .css imports.  
\* The style-loader 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.  
\* Now the order in which we add them in `use` matters.  
\* 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 you write it like this. So it first takes the css-loader and then it applies the styling with style-loader on the extracted CSS code.  
\* We can give this css-loader another property `**options**`. Here we want to set `**modules: true**` to enable CSS Modules and the localIdentName to define how the generated CSS classes due to CSS Modules should look like.  
  
\* We’re not done with CSS though, I also want to add Auto-prefixing.  
**npm install --save-dev postcss-loader**   
\* Now postcss 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. It’s simply a loader which allows us to transform the CSS. For example we could also use it to handle SASS and stuff like that, here, however, I’ll simply add it as another loader and therefore under the previous ones because it should run before them. The `ident` is important internally. The `plugins` are now the steps we should apply or we want to apply to transform everything. Here I want to run `**autoprefixer**`, that needs to be installed, it’s a separate tool.  
**npm install --save-dev autoprefixer**  
\* That’s a third-party library which is able to autoprefix our CSS properties.  
\* We need to import it into the config file.  
  
\* That’s actually a function we need to execute and we pass an object to configure it and there again we pass a list of browsers.  
  
\* Now since we run 1 additional loader before css-loader, we actually need to inform css-loader about that. That’s just a special setup css-loader needs. `importLoaders` set to 1 because we run 1 loader before css-loader is applied.  
  
\* And that’s our CSS-handling setup.  
\* Let’s make sure that we now also can handle images since we also import these.  
**Creating Rules for Images**  
\* The JPG / JPEG might be written with E so that’s why it’s optional using the ?.  
\* These are the file endings we support. For these files let’s install a loader.  
**npm install --save-dev url-loader**   
\* That’s a loader which will take our images and if they are bellow a certain limit we define, it’ll actually convert them into data64 urls which it can inline into our documents so we don’t have to download an extra file but for bigger files that’d be inefficient so files above that limit 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.  
\* We configure it with query params so that the config gets automatically passed on to a fallback which we’ll use if the limit is exceeded. Now that fallback is another Loader we also need to install though.  
**npm install --save-dev file-loader**  
\* This loader in the end simply copies the file you could say, it copies it into a new directory and gives us a link to it and that is the fallback we’ll use automatically here if we exceed the limit.  
\* The limit uses bytes, not kilobytes.  
\* The name is the path where we want to store that file.  
\* [name] is the original name and [ext] is the extension.  
\* So it’s gonna get copied to our dist folder because this `output` setup is taken into account even though we’re not creating a bundle.js file.  
  
\* Now you can add more rules for different file types, for fonts for example, but I’ll keep it to this for now.  
\* Let’s now turn our heads towards plugins you might wanna use.  
**Lazy Loading**  
\* Now regarding Plugins it’s actually super simple for us here.  
\* We don’t need any for the development Workflow, we’ll add 1 later when we set this up for production, there I want to add a Plugin but for now that is all.  
\* Now let’s see if it works if we run our ”start” script which runs the dev server.  
  
=> **We got an error that the `path` is not an absolute path. I accidentally used publicPath instead**.  
\* Now at least it doesn’t crash when we start the server.  
=> It does give us an error though: that it doesn’t understand this import syntax here, the dynamic import syntax.  
  
\* So how do we make it 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.js which is downloaded initially. To support code-splitting in Webpack and code-splitting is just a different name for Lazy Loading, we add `**chunkFilename**`. This simply determines what these other files should be named. They have to have a variable part in their name because they are generated by Webpack dynamically. [id] is dynamically generated number 0, 1, 2… .  
  
=> Additionally, we need to make Babel aware of this dynamic import syntax which it otherwise doesn’t understand.  
**npm install --save-dev babel-plugin-syntax-dynamic-import**  
\* Now in .babelrc file:  
  
\* Now let’s run npm start again.  
=> Now we get this error that this syntax where we assign properties like this isn’t liked.  
  
=> The reason for this is that the “env” preset which we’re using 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.  
=> Let’s add another preset.  
**npm install --save-dev babel-preset-stage-2**  
\* These are DRAFT FEATURES which may eventually get supported but which we already want to use.  
  
\* Let’s now try again with npm start.  
\* Now we don’t get any error in the console.  
\* We’ll face 1 problem though - the application is now served and by default it’s doing that on localhost:8080. There, however, we won’t see the 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’s something we’ll have to do on our own, let’s do that next.  
**Injecting the Script into the index.html File**  
\* To make our project work and servable by the webpack-dev-server or also 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 allows us 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 webpack.config.js:  
  
\* And we create a new instance of it.  
\* `template` is the HTML file I want to use as a basis.  
\* Now I want to `inject` my stuff into the body, that’s extra configuration we can pass to the plugin.  
\* And the `filename` is the output file, that should also be that HTML file, that’s the file it’ll eventually generate.  
\* Again, for the dev-server only in memory, later when we use a different Workflow for building this for production, it’ll give us the real files we could upload to a server.  
  
\* Now we see an empty page.  
=> We do have some errors in the browser console though.   
\* Now these are React errors which is a good sign - our application seems to run.  
=> **We forgot the export the App.js**.  
  
\* If we inspect the Network tab and reload and click Pizza, we see the 0.js file that’s our Lazy Loading.  
\* We have a working Workflow, a working project setup where we can create a React app.  
\* 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.  
**Creating the Production Workflow**  
\* In package.json I want to have a “build” script.  
  
\* Now if I run it like this, it would use the setup we have for the development Workflow.  
\* This will now run Webpack itself and take the same config file we worked with before but now it gives us this `dist` folder.  
   
=> You see the index.html file where it injected our import.  
\* Let’s delete this, it’s not optimized.  
\* The goal is to have a dedicated Workflow for production which does some optimizations.  
\* Let’s add **webpack.prod.config.js** file.   
\* Now it won’t automatically take this file into account, we’ll have to point at it, that’s super simple.  
\* I’ll copy the setup from the webpack.config.js and we can adjust it in here.  
=> In the `devtool` I remove the “-eval-” to create more optimimal 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.s  
  
\* Now 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 Plugins section.  
\* There we have a Plugin for connecting the HTML file and I want to keep that.  
=> **Now I also want to uglify my output, I want to optimize it, and that’s actually a Plugin which is built into Webpack, we don’t need to install it**.  
  
\* This Plugin now also can be configured. And you can always check out the official GitHub page of all these Plugins and Loaders. I’ll take the default setup though, so I don’t need to pass an object.  
  
\* 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” script here. I don’t want to use the default webpack config, instead I want to use a special one.  
\* Since it is in the same folder as the package.json file, I don’t need to specify a path, just the name.  
  
\* With that I’ll also add some additional options, --progress -profile --color to adjust the console output of the webpack build Workflow.   
  
\* And I’ll install 1 additional package:  
**npm install --save-dev rimraf**  
=> 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 we run this command after the deletion is done.  
\* If we have a look at the bundle.js file now we see that this is a minified code now and therefore optimized and the same of course for the 0.js.  
\* Now this `dist` folder would be what we upload to a server, the content of the `dist` folder.  
\* This is now our own Workflow.  
**Wrap Up**  
\* 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 ejected configuration we have for create-react-app, there you can have a look at the Webpack Config they use when setting up the create-react-app config though it’s a very elaborate and therefore complex config.  
\* We created a simple one to understand how Webpack works and what it does.

Adding babel-polyfill

Section 22, Lecture 373

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>

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