Important Notes

Node.js can be used for:

1. Your personal robot, such as by automating tasks (with webpack, etc.), or by adding files to your computer (using the fs package – you don’t need to remember this – it’s kinda irrelevant, but it’s just an example.)
2. Backend server that will communicate with databases (such as by using mongodb package), make requests to a server (such as by using express package, etc.)

With NPM, you can install packages directly related to Node.js, but you can also download other random packages such as bootstrap or lodash, which relate to general CSS & HTML, so NPM is not just stuck in grabbing Node.js related packages

When you do “let express = require(‘express’)”, the variable name, express, is essentially the package, and you can access the methods, and properties of the package by simply using dot notation. So just remember that when you import a package like this (which has to be done only when you are in Node.js – Javascript web environment is different), remember to always see the variable as the package itself, and when you do dot notation to call methods or properties, you are calling them from that package itself.

Always do “NPM init –y” for each of your project repositories. This creates the package.json file in your project that is basically your grocery list that stores all your files that you use for your project. If your node\_modules gets deleted, you can just type “NPM install”, and you will get it all back.

You should add the node\_modules folder to your .gitignore file, so it doesn’t appear when you push your project to Github

To install webpack, type in “npm install webpack webpack-cli --save-dev” You need to use --save-dev because you only want to use the webpact package for development. You don’t want it to be saved in your dependencies because then this will make the users of your site need to download it whenever they load your website. By using --save-dev, the webpack package will get saved to your dev dependencies instead (in your package.json file.

To start using webpack, first go to your package.json and add the “scripts” property if it is not yet there, then create an object, and set a property name that will represent what you call on the command line (ex, set the property name as dev, so you can call npm run dev), then for the value, enter in “webpack”.

You also need to create a file in your main directory called “webpack.config.js”, and inside that file, you need to write the configurations for the webpack package. To do this, first create a variable named “module.exports” and set it equal to an object. Inside that object, you need to give it propertys that will ultimately configure webpack to your liking. So first, you need to give it an “entry” property, and this should have a value of the path to the file that you want webpack to watch, process, and bundle. Then you need to give the object an “output” property, which should be equal to an object. Inside this output object, you should have a property of “filename” equal to whatever name (ex. bundles.js) you want the output file to be (the output file is what webpack creates when it processes/bundles the file you pointed to in entry). Then you want to add a “path” property to the output object, and set it equal to “path.resolve(\_\_dirname, ‘top directory name here’)”, this will tell webpack where it should place that output file (Note: make sure to put “const path = require(‘path”) at the top of the file – path is a package already installed on node.js, so you don’t need to install it with NPM). Now, outside of the output object, and back to the main object, you need to add a “mode” property, and set it equal to “development” (if you are n development – if you want to publish it to a server, I think you set mode equal to “build”). Lastly, you will want to add the “watch” property and set it to “true” - this will tell webpack to watch for changes in your file that you pointed to in entry, and whenever there is a change, webpack will re-run itself, and output an updated file, so you **don’t** need to cancel webpack and rerun to see changes. Also, make sure that once you run “npm run dev”, and when the bundles.js(or other) file is created, make sure to go into your index.html template, and include that javascript file in there at the bottom of the body (ex. write <script src=”bundles.js”><script>.

Now that your javascript it set up, you will want to configure webpack some more, so that it also adds in all your css to your bundles.js folder. Naturally, webpack is only configured to accept js files, so in order for it to understand, process, and bundle your css files, you need to add in some extensions/packages that will modify webpack, so webpack will know what to do when it encounters files ending in .css. Before I get into that, it’s important to note how you should set up your css. You should go into your main directory, and you should already have an “assets” folder that has a “scripts” folder that stores your js files, and your assets folder should also have an “images” folder that stores your images, and now you should add a styles folder, which will store all your css files. Inside your styles folder, you will want to create one main css file called “styles.css” (it’s convention…), and this file should be the main css file where you import all your other css files (such as header.css, footer.css). Now, once you created the styles.css file, you will now want to go to your main javascript file (the one that your webpack config entry points to – NOT THE BUNDLES.JS that webpack outputs), and import the styles.css file (you can do this by typing “import ../styles/styles.css” (or something similar). So now, whenever you make any changes to any of the css files (like header.css), and save it, since it is imported inside the main styles.css, that will be updated, and since styles.css is imported into the main javascript file that webpack is watching, that will get updated too, and webpack will sense the update, and output a new bundles.js(or other named output file), and since this is what your index.html is connected to, your server will automatically be updated. However, css is not completely set up just yet. Yes, you connected it to the main javascript file, but webpack does not understand css right off the bat, so we need to install packages, so webpack will know what to do and how to process css files. So go to the terminal, and type in “npm install css-loader style-loader --save -dev”. css-loader is a package that allows webpack to understand how to process and bundle the css files, so the css from your css files will appear in the bundles.js file now. style-loader is what allows webpack to actually apply the css to the server/website – without style-loader, your css will not get updated. Now, in order to use these packages, go into your webpack.config.js file, and add a new property to the object called “module” (this is what tells webpack how to react to different languages), so give this a value of an object with a single property of “rules”. Make “rules” equal to a list. Inside that list, create only 1 object for now, and inside that object, give it a property of “test” and set it equal to “/\.css$/i” – this tells webpack that when you see a file ending in .css, then … this is where the next property comes in. The next property name (inside the same object as “test”) is “use”, and set this equal to a list that contains “style-loader” and “css-loader”. So now webpack knows that when it sees a file ending in .css, it knows to use those loaders to handle it, which gives us the functionality above (bundles our css into bundles.js file, and displays it on the server/website).

Now, if we want to add some automation & further functionality to our development process, we should add in PostCSS, which is a package that modifies our css after we’ve created it, and makes it readable by browsers. We can install postcss plugins to give us more functionality in our css, such as by having easy-to-create variables, allowing us to nest our css, and adds an autoprefixer, which essentially makes sure our css is readable by all browsers (it adds the ---webkit stuff). Before we start to install postcss and it’s plugins, make sure that you are always ending your npm install’s with –save-dev up to this point, because none of these packages need to be stored in our dependencies (they don’t need to be downloaded by our users to use our site – they just need to be used for development). So to start implementing PostCSS (which you should because of the autoprefixer). First, go to the command line and type “npm install postcss-loader --save-dev”, then go into your webpack.config.js file, and go to the module > rules > “use” line, and you need to add in the postcss-loader here to the end of the list, so that webpack knows to add this for .css files. But postcss-loader is different from the others because it doesn’t come with anything enabled. You will need to add in plugins to add the postcss functionality. So, at the end of the list (still inside the list), create an object, and give it a property of “loader” and set it equal to postcss-loader, then give it another property of “options”, and set it equal to an object that has a property of “plugins”, and you want to set this to a list of the postcss plugins that you want to use, so rather than putting a long lsit there, just create a variable at the top of the file, preferably called “postCSSPlugins”, and set it equal to a list, and you can now just put this variable name in the value for “plugins” property. Now, in the postCSSPlugins list at the top of the file, you want to import the plugins, so go to command line, and install (1) postcss-simple-vars – which allows you to create the variables by doing $mainBlue: #12d4d2, and using that variable with $mainBlue (2) postcss-nested – which allows you to nest your css (3) autoprefixer – which adds the –webkit and such. Make sure you end this npm install with --save-dev. Now, to activate these plugins, go to the empty postCSSPlugin list, and add these plugins there. Remember, they are packages, so to import/list them, you should use require(“postcss-nest”), require(“autoprefixer”), etc. And now, you are done setting up your webpack configuration for css and your postCSS configuration, and you are free to use variables, and nesting in your css, and postCSS will modify it to make sure all browsers understand it. Note, that right now, you can make saves in your css files, and you don’t need to rerun the npm run dev (because you set watch: true awhile back), but you will still need to refresh your server/browser to see your css changes. We will probably get into that soon to make it so it auto-refreshes on save.

Yes, it’s time. So if you want to do any of these things: (1) make it so when you save your css or js file, your server/browser gets automatically updated (2) When you save an html file, your server/browser gets automatically reloaded/updated (3) You want to access your local host / development server from another deivce, such as a mobile phone, or tablet, then you can do all this with a singular package, go to your command line, and type in “npm install webpack-dev-server --save-dev”. This is a webpack package that is designed to make your development server fully optimized by doing the things outlined above. To leverage/activate this package, first, go into your package.json file, and go to the “scripts” property, and change the “dev” property inside of it to make it equal to “webpack-dev-server”, so now when you run “npm run dev”, it will run the webpack-dev-server instead of just regular webpack. Now, go into the webpack.config.js file, and go into the main object, and add a new property called “devServer” and set it equal to an object. Inside that object, add the property “contentBase” and set it equal to “path.join(\_\_dirname, ‘main folder that contains your html file, bundles.js, and assets folder”) – this tells webpack to “serve” this file up to the browse (basically since this folder contains your html file, and since that is what you want to show on your browser, now webpack knows that). If you have any issues, maybe come back to this part to make sure it’s correct, cause it’s a little confusing, so you might have to check online if you have any issues. Next, add the property “hot”, and set it equal to: true - this tells webpack to apply that page auto-refresh, without refreshing the page. To make this work, you need to go into the main js file (where all your other js files and main css is imported into), and add:

if (module.hot) {

module.hot.accept()

}

Back to the properties, you can add in a port property and set it to 3000, but I believe you don’t have to, and instead just visit localhost:8080 because I think 8080 is the default. These settings above will configure webpack so your css and javascript will automatically update on the browser/dev server. If you want to make it so when you make changes and save your html document, the browser refreshes, then you just need to add another property to the devServer object. Add the property “before” and set it equal to a function (make sure to include a comma at the end of the “}”). For this function, give it parameters of app, server (in that order), and in the body of the function, write “server.\_watch(“./app/\*\*/\*.html”)” – note, yours may be different, just make sure it points to the main folder, and the html files inside of it. The /\*\*/ part basically says get through any subdirectory and the /\*.html says watch all files ending in html, so together, this property, and function watches for any changes in any html files, and will reload the browser/server whenever there is a save in them. Note that you can remove the “watch” property in the main object (inside module.exports) because the devServer property already watches for changes in the css and js, so the watch property servers no purpose. Finally, if you want to access your dev server/browser from a mobile phone or tablet or other computer, you just need to add in a property to the devServer object. Simply add in the property “host”, and set it’s value equal to “0.0.0.0”. Now, to access your browser from any other device, first make sure it’s on the same network as your computer, then go into your setting preferences > network, then at the top right look for the number starting with 192.168.X.X, and enter that value in the url of your other device, and enter in the appropriate port number (either 3000 or 8080 depending on what you set). Also, just a side note, now that you are using webpack-dev-server as your “npm run dev”, now whenever you run that command, a new bundles.js file actually isn’t created because webpack-dev-server stores the bundles.js in random memory, so it’s faster, and so now you can delete the bundles.js file from your directory because you don’t need it since you will be using webpack-dev-server from here on out to develop your server.

When you are importing your css files into your main styles.css, webpack and postcss will automatically see that as an import, and this will cause your browser to have to download all of your different css files, rather than just the one main styles.css file, which is no good! So, in order to make it so postcss and webpack see your imports as if it were the actual css code, you will want to install the plugin “postcss-import”, and make sure you add this to the postCSSPlugins list in your webpack.config.js file in order to leverage it.

Speaking of impots into your main styles.css file… You should always include @import ‘normalize.css’; in your styles.css. Note that normalize.css is a package that you will need to install, and make sure you install it into your dependencies (don’t include the --save-dev). Normalize makes it so your code is better fit for all browsers, and I believe it resets the margins & padding. Maybe look into what this does, also find out what the install name is.

When you are working with media queries in your css, it can be a hassle to always have to type out @media (min: 530px), so to make things easier, install the postcss-mixins plugin, and add it to your postCSSPlugins list in the webpack.config.js file. Now go into your styles folder, and add a “\_mixins.css” file to your “base” folder, and inside this file, you should add all your mixins. Remember, a mixin is just a reusable bit of code. To create a mixin, do the following:

@define-mixin atSmall {

@media (min-width: 530px) {

@mixin-content

}

}

The “atSmall” should be where you provide the name for the mixin. Then inside the mixin, you should put what code you want to run when the mixin is called, so we want to run the @media(min-width:530px) media query, and inside that media query, we want to store the @mixin-content (which will be the content we provide in our css file after we call this mixin with @mixin atSmall). Always make sure that after you create a new css file, that you import it into your main css file, so you can actually use it.

If you want to create a background-color that is an rgba value of your main theme colour (for example, if you want your header to be the main theme colour, but with an opacity of 0.3, so it’s see through), then you can install a postcss plugin named postcss-hexrbga, then once you leverage it by adding it to your postCSSPlugins list in the webpack.config.js file, you can go to your css file, and set background-color: rgba($mainBlue, 0.3); then postcss will convert the actual color to the rgb color. Alternatively, you could just get the hex value of your color, and find some hex to rgb converter online, and just do it that way.