**Babel**

Babel converts ES6 and ES7 code into older versions of Javascript allowing it to work on all browsers

Install Babel globally – **npm install -g babel@6.26.0** (swap out latest version number)

Next run command to take js file and output another one from babel

e.g – **babel input.js -o output.js –** this will create the output file but not the babel version of the JS. We will need to install a dependency local to the director of the project and not globally to get this done

Next we’re going to create a package.json file – **npm init**

Now we’re going to create out babel version of input – babel input.js -o output.js --presets env

In your project directory, you would usually create a **src** folder and a **public** folder.

src – Contains input file. this folder contains all the code you write in ES6/ES7

public – Contains output file. this is what the server will hold and will contained all the babellised JS code

Side note – The **npm\_modules** folder would never get uploaded to a server, it’s too big. You just use it locally. If you wanted to use this project or give to someone and have them use all the dependencies you are using, you simply update your **package.json** file and fill in the necessary dependencies in that file. Then all you have to do is run **npm install** and it will then install all the dependencies you listed in your package.json file.

Then with your input file in src and output file in public, run the following command to get the babel version – **babel src/index.js -o public/scripts/bundle.js --presets env**

Instead of having to write the above command every time, you can add it to the scripts section of your **package.json** file and just run that command every time you want to use that script. Below is the following section **from package.json:**

"scripts": {

"build": "babel src/index.js -o public/scripts/bundle.js --presets env"

}

Now all the you have to do is run the following command – **npm run build**

This will now run this script every time

Now, you don’t need to run this command everytime you make a change, instead after **env** add **–watch**  and run **npm run build** again. Now any changes you make to index.js will automatically go to bundle.js

Next, we’re going to uninstall packages we have globally and reinstall them locally.

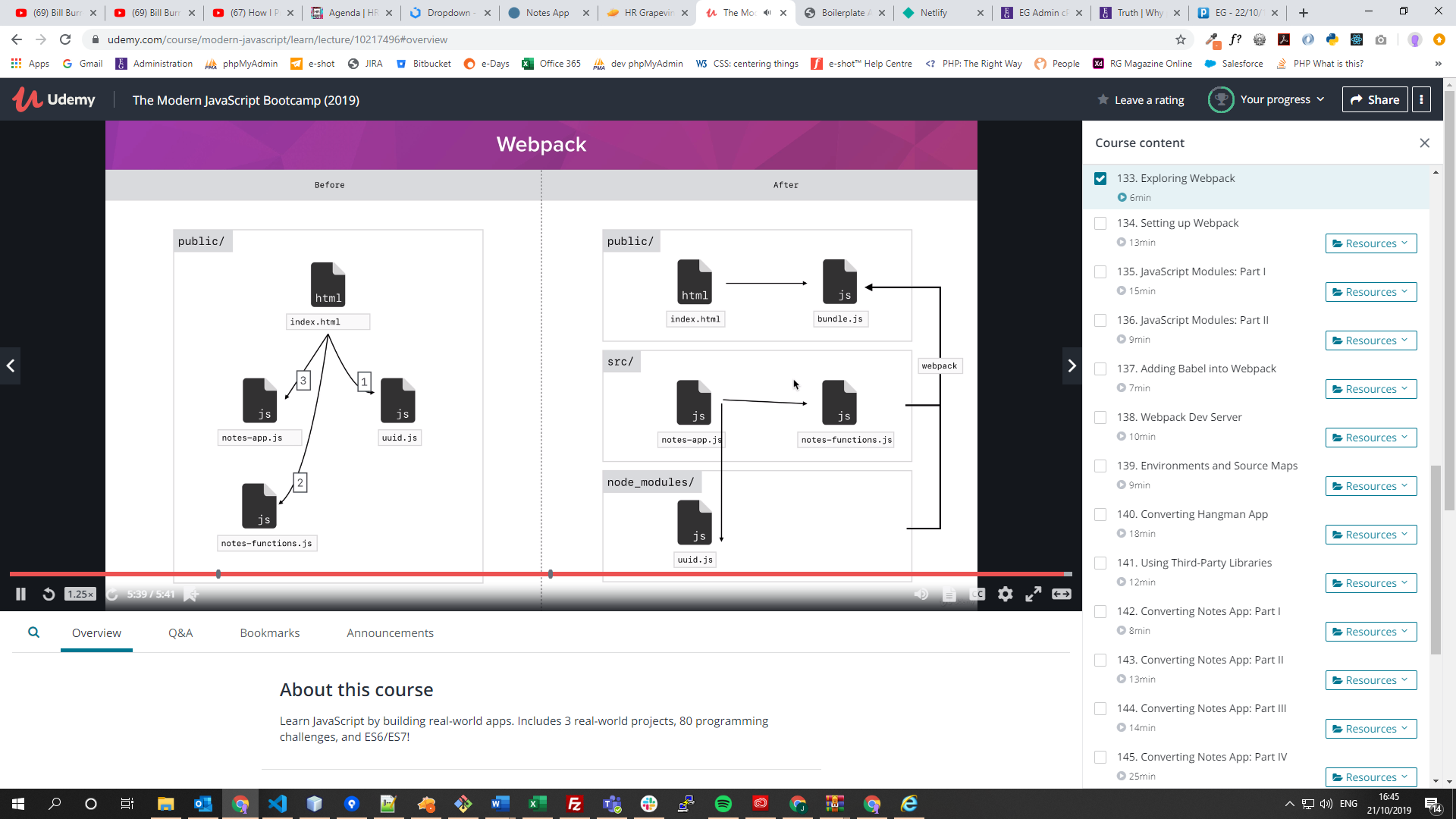
**npm uninstall -g babel-cli**

**npm install** [**babel-cli@26.6.0**](mailto:babel-cli@26.6.0)

This will now add babel-cli to your dependencies in your **package.json**

This means that now if you run npm install, these will get installed locally

**Webpack**



**npm install webpack@4.5.0 webpack-cli@2.0.14**

The above command will install webpack and webpack-cli into your node\_modules as well as adding it to your package.json file under dependencies. Next add it to your scripts in package.json so you can run webpack with a simple command. See below:

"scripts": {

"webpack": "webpack",

"build": "babel src/index.js -o public/scripts/bundle.js --presets env --watch"

}

Create a **webpack.config.json** – this will allow you to configure webpack so that you can bundle everything together. Below is the initial configuration which shows you entry and output points.

const path = require("path")

module.exports = {

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

output: {

path: path.resolve(\_\_dirname, 'public/scripts'),

filename: 'bundle.js'

}

}

You will then run npm run webpack

**Importing and Exporting modules**

You may have several JS files and want to use methods or functions from another file.

You can export methods from files and import them in another.

You can export as many as you want and have one default one from each file.

To export methods, see below:

console.log('utilities.js')

const add = (a, b) => a + b

const name = 'Jack'

export { add, name }

You could put the word export before every **const** but this is the most efficient way.

To set the default, see below:

const scream = (word) => `${word.toUpperCase()}!`

console.log(scream)

export { scream as default }

Then, in your JS file that you want to use this for, you need to import them in the following way

import { add, name } from './utilities'

import otherScream from './scream'

console.log('index.js')

console.log(add(2,5))

console.log(name)

console.log(otherScream('Howdie'))

Because you set default in scream.js, when you import you can name it whatever you like

On line 1 – import the method names from utilities.js (you can omit file extensions)

**Pulling in Babel for webpack**

First of all run – **npm install babel-loader@7.1.4**

We then need to add a few lines to our **webpack.config.js**

**SEE BELOW**

const path = require("path")

module.exports = {

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

output: {

path: path.resolve(\_\_dirname, 'public/scripts'),

filename: 'bundle.js'

},

module: {

rules: [{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader',

options: {

presets: ['env']

}

}

}]

}

}

Add module

**Rules** – these are to add rules which is an array which takes an array of objects

**Test** – Regular expression to tell babel to look for all files that end in js - \ ignore the . $ tells it to look for .js at the end of the file

**Exclude** – directory for webpack to ignore

**Use** – Tell it to use babel loader and tell it use the preset env as an option

We now can remove babel from our scripts in package.json and rename our webpack command from webpack to build.

"scripts": {

"build": "webpack"

},

NOTE – NOW WE DO NOT NEED TO USE BABEL OUTSIDE THE CONTEXT OF WEBPACK, WEBPACK WILL TAKE CARE OF EVERYTHING FOR US

**Setting up webpack-live-server**

Instead of having to run the webpack build script every time, we want webpack to watch our files so that every time we save it will just reconfigure and refresh the page. Webpack live server solves this.

Install webpack live server by running – **npm install webpack-dev-server@version**

Only thing we now have to change is the config in our webpack.config.js to tell it where to look for file changes and where to server them.

We’re going to add a new route property along with **entry, output and module** called **devServer**

**contentBase –** Absolute path to tell dev where the files you want to convert, live

**publicPath –** The path, relative to the absolute path, where you want the files you want to convert live

Seems a bit silly having to provide content base then the path within that but that’s what you have to do

devServer: {

        contentBase: path.resolve(\_\_dirname, 'public'),

        publicPath: '/scripts/'

    }

Then just re-configure your package.json file to run web server

"scripts": {

    "dev-server": "webpack-dev-server --mode development",

    "build" : "webpack --mode production"

  }

**Also, a note about the above:**

**dev-server –** we have a dev version so that it doesn’t convert everything when we save, it just tests the work, this saves time and therefore development speed

**build –** now we only need to run this when we want to convert everything

**Adding Source map**

At the moment, when you look at the console, you’ll see that the line of code that the console tells you about where something is, is actually wrong. This is because babel runs in the background and the code it delivers is after that process has been run

However we can add source-map to our webpack config and this will show us the correct line a certain line of code is on.

 devtool: "source-map"

Then, run live server again and source map will now work

NOTE – WHEN YOU’RE DONE WITH A PROJECT, YOU CAN JUST DELETE **node\_modules.** If you need it again, you can just run **npm install**

In our boilerplate case we delete **node\_moduels** and delete all the other files in src, keeping just index.js as this file is our entry point

**BELOW IS HOW WE CONVERTED OUR HANGMAN GAME USING WEBPACK BUT THIS CAN APPLIED TO ANY PROJECT**

First thing we want to do is make a copy of our **boilerplate** directory and rename it to the name of our package

Then we move our styles and images folders over to the public directory in our new hangman directory

Then, take our index.html from our old hangman game and paste it over the top of our new hangman index.html. The only thing we need to do now is, delete the multiple script tags, leaving just one and change the source to point to bundle.js

Now, our main JS file in our old hangman game is **app.js.** Our main JS file in our new hangman game is **app.js** which is our entry point four our app. So we copy everything from **index.js** and paste it into **app.js**

Then we just move across **requests.js** ad **hangman.js** to our **src** folder in our new project

At the moment this won’t work. In index.js we are referring to methods in the other two files. We will have to export these and import them back in app.js, the way we saw earlier.

If you exporting something fairly large like the hangman class, it’s best to export this as the default

At the bottom of the **hangman.js** file, we do the following

export { Hangman as default }

And now at the top of **index.js** we just import that in the following way

import Hangman from './hangman'

Note – when we import we could name Hangman as anything, as this is the default

Next, in **request.js** we need to only export the getPuzzle function the same way we export Hangman above

export { getPuzzle as default }

Then import the same way above

import getPuzzle from './requests'

Now when we run over dev server, it won’t work. This is because although babel will run, it won’t replace anything that’s not new syntax, it just replaces syntax. So newer array methods such as [].includes, don’t get converted. So we get the error below. Therefore, we need to add a polyfill.

requests.js:1 Uncaught ReferenceError: regeneratorRuntime is not defined

In order to solve this, we need to install babel-polyfill.

To install, run this command

**npm install babel-polyfill**

Then we just need to make a small adjustment to our package.json

We update our entry point to include babel-polyfill. We can enter change our entry point to an array so we can have multiple entry points

entry: ['babel-polyfill', './src/index.js'],

NOTE – WE FINALLY ADD THE FOLLOWING LINE TO OUR BOILERPLATE **package.json**

"babel-polyfill": "^6.26.0",

And the same entry points as above in our webpack config file in our boilerplate