**Section 2: Git Essentials**

Git -> Version Control System

Git Advantages

* History -> Which date which code added, track changes
* Collaboration -> Involve a team, update everyone in the team, git merge for us
* Feature branches -> separate branches for different tasks and then merge them to main branch

Git Vocabulary

* Project = **Repository (repo)**
* Working directory
* Staging (preparing, getting ready) = **control what gets committed (choose before commit)**
* Commit = **Gits way of saving**
* Push -> push from local repo to server (Github)
* Pull -> pull from server to local repo

To create a git repo -> **git init**

To see changes, if any -> **git status**

To add staging area -> **git add <file name>**

To save changes -> **git commit -m “commit message”**

To go back to last commit -> **git checkout -- .**

Get existing repo -> **get clone “web address”**

Create a GitHub repo -> I named it “**Git-a-web-dev-job-course**”

Clone course files from the brad’s repo-> **git clone https://github.com/LearnWebCod**

**e/travel-site-files.git**

Then, change the cloned file to github repo name you just created (**Git-a-web-dev-job-course**)

To see which repo in github the project registered -> git remote -v (this shows origin as brad’s)

This says the repo will be pushed to learnwebcode which is the repo we cloned

To change repo -> **git remote set-url origin** [**https://github.com/erol-eyupoglu/Git-a-web-dev-job-course.git**](https://github.com/erol-eyupoglu/Git-a-web-dev-job-course.git)

Then **git remote -v** shows -> origin https://github.com/erol-eyupoglu/Git-a-web-dev-job-course.git (fetch)

origin https://github.com/erol-eyupoglu/Git-a-web-dev-job-course.git (push)

after changing origin you can push the files to github server -> **git push origin master**

**General commands after a change:**

git add -A

git commit -m “commit message”

git push origin master

**Section 2: Intro to NodeJS & NPM**

Automation and Organization

Package Management

Install Node on server or local personal computer

* We will use Node.js to automate things in local computer

For example, we can create a html file and put code in it and show any error by using fs.

var fs = require('fs');

fs.writeFile(\_\_dirname + "/index.html", "<h1>html is great</h1>", function(error) {

if (error) {

return console.log(error)

} else {

console.log("Congrats")

}

})

Moreover, we can programmatically download and save an image file from internet using https or http packages (in this case it was https)

var https = require('https');

var myPhotoURL = 'https://hips.hearstapps.com/hmg-prod.s3.amazonaws.com/images/golden-retriever-royalty-free-image-506756303-1560962726.jpg?crop=0.672xw:1.00xh;0.166xw,0&resize=640:\*';

https.get(myPhotoURL, function(response) {

response.pipe(fs.createWriteStream(\_\_dirname + '/mydog.jpg'));

});

Using node, we can automate many things.

**NPM**

Initiate the npm packages -> npm init -y

Install packages loadash and normalize.css

* npm install lodash
* npm install normalize.css

to install all packages from the package.json file -> npm install

In the real world you'll usually want to use the newest version of a package that is available on NPM. However, to make it easier to follow along with the video lessons in this course I strongly encourage you to use the same versions of packages that I'm using. Here's how you can achieve that:

1. Within your project folder **delete** the node\_modules folder and also **delete** the package-lock.json file.
2. You'll see a downloadable file associated with this lesson that is named package.json. Download this file and replace the package.json file that is currently in your project folder with this newly downloaded file.  
     
   If you've already made it to the later chapters of this course and are just now circling back to this lesson you will have a **"scripts"** area in your existing package.json file; copy your existing **scripts** object code into your clipboard before replacing your package.json file so that you can paste your existing **scripts** area back into the file you download from this article. This way you don't lose the npm run dev and npm run build tasks you've already setup.
3. In your command line run npm install and be aware that it could take a few minutes because it is installing all of the packages that we'll need for the entirety of our Complex App.

**Section 4: Webpack Essentials**

Bundles your assets

Npm package

How to install webpack

* Create a folder ‘scripts’ in assets
* Create a new file in this folder ‘App.js’
* alert("helloooooooooo");
* open terminal in course path
* run the command -> npm install webpack webpack-cli –save-dev
* then create a file named exactly ‘webpack.config.js’ in course home dir
* const path = require('path')
* module.exports = {
* entry: './app/assets/scripts/App.js',
* output: {
* filename: 'bundled.js',
* path: path.resolve(\_\_dirname, 'app')
* },
* mode: 'development',
* watch: true
* }
* module.exports: this js object will be used to bundle,entry: which file to be bundled, output: bundled.js file will be created in path location, watch: after every change it will be bundled again,
* in package.json file modify like this (scripts added):
* {
* "name": "travel-site",
* "version": "1.0.0",
* "scripts": {
* "dev": "webpack",
* "test": "echo \"Error: no test specified\" && exit 1"
* },
* After config, run this command: npm run dev
* Add this in index.html after footer section
* <script src="bundled.js"></script>

Very recently PostCSS released a new major update to their package. They have changed the way we're required to configure their plugin inside our webpack.config.js file. Everything in the next video lesson is still accurate with one exception. The newest version of PostCSS requires us to nest our options inside an extra wrapper object called postcssOptions. For example this code from the following lesson:

{loader: 'postcss-loader', options: {plugins: postCSSPlugins}}

Must now be this instead:

{loader: "postcss-loader", options: {postcssOptions: {plugins: postCSSPlugins}}}

**CSS with Webpack**

Css autoprefixer -> adjusts the css code to web browsers like -moz-clomns…

* Create styles folder in assets and styles.css within it
* Load css from js ->
* import '../styles/styles.css'
* npm install css-loader style-loader –save-dev (as we changed the package.json file no need to re-install these)
* then go to webpack config file and add this
* module: {
* rules: [
* {
* test: /\.css$/i,
* use: [‘style-loader’,'css-loader']
* }
* ]
* }
* This allows for every file ends with .css, webpack will use css-loader to load and bundle it within bundled.js. style-loader will show the css in the page

Why we are loading CSS for JS???

In dev process, it is better in terms of speed and automation.

PostCSS

* Post-css loader installed with npm
* Change webpack config, post-css needs some additional options: loader, options, plugins -> which css properties we want to display
* use: ['style-loader', 'css-loader', {loader: "postcss-loader", options: {postcssOptions: {plugins: postCSSPlugins}}}]
* Add postCSSPlugins const to webpack config file
* const postCSSPlugins = [
* require('postcss-simple-vars'),
* require('postcss-nested'),
* require('autoprefixer')
* ]
* Then test the post css code in styles.css:
* $mainBlue: #2f5572;
* .large-hero {
* h2 {
* color: $mainBlue;
* }
* }
* This code is not a valid css code but because we modify postcss settings it became valid because webpack created a file for browsers to understand the code

By default, the css-loader will attempt to handle any images we reference in our CSS (e.g. background images, etc...). While this is great in certain situations, for our usage in this course we want to disable this and we'll manage our image files manually. With this in mind, when you list 'css-loader' in your **webpack.config.js** file you'll want to add an option to the end of it like this 'css-loader?url=false' instead.

For a complete reference, you can also simply use the **webpack.config.js** file I've added as a resource for this text lesson.

**Section 5: CSS Architecture**

Identify patterns in design

Rules to follow for creating class names and selectors

***File Architecture***

Go to app/styles and create a modules directory

Then create \_large-hero.css file (it starts with \_ which means it is a partial)

Then import it in the main css file:

@import 'modules/\_large-hero';

npm install postcss-import –save-dev

in webpack config file add this to postplugins array

require('postcss-import'),

create base folder and create \_global.css file and cut and paste the css code in styles.css

@import 'base/\_global.css';

Import normalize.css

@import 'normalize.css';

In html file, large-hero img file we want to position the following h1 h2 and p sections inside the img. So first create a div and put these texts inside it. After that give that div a class name = large-hero\_\_text-content

Create the css for targeting this new class. Add these

.large-hero {

position: relative;

}

.large-hero\_\_text-content {

position: absolute;

top: 0;

left: 0;

width: 100%;

text-align: center;

}

Absolute position and top 0 left 0 will place the text in the left upper corner of the image and then width 100% and text-align center will center **horizontally**

To center vertically:

top: 50%;

transform: translateY(-50%);

change top from 0 to 50% and transform: translate (-50%) – means to upper. translate will position an element vertically relative to itself

***Identify patterns in design***

We want to target h1 element in large-hero div.

One way to do it, .large-hero h1 {…} but it is not recommended

We will do it by using BEM principles

BEM -> **Block** (independent resusable part). **Element** (belongs to a block). **Modifier** (change to default state of object).

If you want to target an element belongs to block for example **large-hero**, it is **large-hero\_\_title**

For example, there is a button. If u want to modify its color you can create btn—orange. Or size, btn—large.

**BEM advantages**

Css selectors should target elements directly with classes

Move blocks around and reuse them

Blocks can be nested inside other blocks

Single responsibility

There is no reason to code a pattern more than once

Makes the relationship between HTML and CSS clear.

So we want to change styling of large-hero block’s h1 and h2 elements. First give them appropriate class names:

<div class="large-hero\_\_text-content">

<h1 class="large-hero\_\_title">Your clarity.</h1>

<h2 class="large-hero\_\_subtitle">One trip away.</h2>

Then design css

large-hero\_\_text-content {

position: absolute;

top: 50%;

transform: translateY(-50%);

left: 0;

width: 100%;

text-align: center;

}

.large-hero\_\_title {

font-weight: 300;

color: #2f5572;

font-size: 4.8rem;

}

.large-hero\_\_subtitle {

font-weight: 300;

color: #2f5572;

font-size: 2.9rem;

}

But we can nest them in block element (large-hero). & means block element which is .large-hero.

.large-hero {

position: relative;

&\_\_text-content {

position: absolute;

top: 50%;

transform: translateY(-50%);

left: 0;

width: 100%;

text-align: center;

}

&\_\_title {

font-weight: 300;

color: #2f5572;

font-size: 4.8rem;

}

&\_\_subtitle {

font-weight: 300;

color: #2f5572;

font-size: 2.9rem;

}

}

**PostCSS vs Sass: What are the differences?**

**PostCSS:** *Transform CSS with JS plugins*. PostCSS is a tool for transforming CSS with JS plugins. These plugins can support variables and mixins, transpile future CSS syntax, inline images, and more; **Sass:** *Syntactically Awesome Style Sheets*. Sass is an extension of CSS3, adding nested rules, variables, mixins, selector inheritance, and more. It's translated to well-formatted, standard CSS using the command line tool or a web-framework plugin.