transfer state forward and back

transfer state forward and back

mongo ds129053.mlab.com:29053/web322\_a71 -u <dbuser> -p <dbpassword>

mongorestore -h ds129053.mlab.com:29053 -d web322\_a71 -u jlee465 -p Vkvkdi0^

mongodb://user:password123@ds121014.mlab.com:21014/team-api-test

npm install

git init

heroku login

create heroku

git push master heroku

data-service.js --> closer

populate(join?????)

app.use(cors()); \*\*\*\*\*\*

jquery

$(function() { // this is the same as: $(document).ready(function(){

jquery-ajax

let tbody = $("#employees-table tbody");

let row = $('<tr>').append(fNameTD).append(lNameTD).append(positionTD); // create the row and append all of the TD elements.

## SASS

SASS, or “Syntactically Awesome StyleSheets” is an extension of CSS that adds power and elegance to the basic language. It allows you to use [variables](http://sass-lang.com/guide#topic-2), [nested rules](http://sass-lang.com/guide#topic-3), [mixins](http://sass-lang.com/guide" \l "topic-6), [inline imports](http://sass-lang.com/guide#topic-5), [and more](http://sass-lang.com/documentation/file.SASS_REFERENCE.html), all with a fully CSS-compatible syntax. Sass helps keep large stylesheets well-organized as well as getting small stylesheets up and running quickly.

There are two syntaxes available for Sass. The first, known as SCSS (Sassy CSS) is an **extension** of the syntax of CSS and the primary syntax that we will be **using in this course**. The other syntax (“Sass”), provides a more concise way of writing CSS. It uses indentation rather than brackets to indicate nesting of selectors, and newlines rather than semicolons to separate properties.

### Getting Started

SASS functions as a CSS precompiler - it adds functionality to CSS in a layer above it and we must run a script / program to convert our SASS files into regular CSS. Where do we find this program? There are [plenty](http://sass-lang.com/install) to choose from, ranging from simple command line tools to elaborate GUI applications. However, we will be sticking with something simple…

### Using NPM (of course)

NPM - what can’t it do? Today we’ll be using it to obtain a package called “[node-sass](https://www.npmjs.com/package/node-sass)”:

Even though we aren’t using Node.js for our application, it still pays to have a “package.json” file in our working directory to manage our dependencies. In the root of your current working directory (If you’re following along with the code examples, it will be the “SASS” directory), create a very simple “package.json” file with an empty list of dependencies:

{

"dependencies": {}

}

Once you have saved this file, close it and open your Integrated Terminal in order to execute the command:

npm install --save node-sass

Next, we need to add a few directories and files to our working directory:

* CSS
  + [empty]
* SASS
  + \_reset.scss
  + main.scss

And finally, to make sure our new “node-sass” CLI works to “watch” our **main.scss** file for changes and correctly update a new file **CSS/main.css**, we must add the following “scripts” property to our “package.json” file:

"scripts": {

"build-css": "node-sass scss/main.scss css/main.css --include-path scss -w"

}

Now, your complete “package.json” file should look something like:

{

"dependencies": {

"node-sass": "^4.5.3"

},

"scripts": {

"build-css": "node-sass scss/main.scss css/main.css --include-path scss -w"

}

}

Notice how the “build-css” script is set to run “node-sass” with “**scss/main.scss**” as the source and “**css/main.css**” as the destination. There is also the **include-path scss** option (which will help us to import files) and a **-w** flag (which will instruct the script to watch our source file for changes and automatically update the destination!

To get node-sass running, simply execute the command:

npm run build-css

Now, every time we make a change to **main.scss** our SASS will be compiled and the resulting CSS will be saved in the CSS folder as **main.css**. We can leave this process running in the background and not have to worry about any additional “compile” steps.

You will know when the process has completed successfully when you see the following green message in the terminal: **Rendering Complete, saving .css file…**. Similairly, you will know that there was an error compiling your SCSS if you see a red JSON string in the terminal with the **“message”** property reading: **“Invalid CSS after …“** (… somewhere close to your error within the SCSS file)

## Working with SCSS

With our script humming along in the background waiting for changes, why don’t we try out some of the great features of our new (Awesome) CSS extension language?

Please Note: If you’re following along with the “Code Samples” the following functionality is located in the **“week2/SASS”** folder

### Variables

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you’ll want to reuse. Sass uses the $ symbol to make something a variable. Here’s an example:

$font-stack: Helvetica**,** sans-serif;

$primary-color: #333;

body {

**font**: 100% $font-stack;

**color**: $primary-color;

}

When the Sass is processed, it takes the variables we define for the **$font-stack** and **$primary-color** and outputs normal CSS with our variable values placed in the CSS. This can be extremely powerful when working with brand colors and keeping them consistent throughout the site.

body {

**font**: 100% Helvetica, sans-serif;

**color**: #333;

}

### Nesting

When writing HTML you’ve probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn’t. Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice. With that in mind, here’s an example of some typical styles for a site’s navigation:

nav {

ul {

**margin**: 0;

**padding**: 0;

**list-style**: none;

}

li { **display**: inline-block; }

a {

**display**: block;

**padding**: 6px 12px;

**text-decoration**: none;

}

}

You’ll notice that the **ul**, **li**, and **a** selectors are nested inside the **nav** selector. This is a great way to organize your CSS and make it more readable. When you generate the CSS you’ll get something like this:

nav ul {

**margin**: 0;

**padding**: 0;

**list-style**: none;

}

nav li {

**display**: inline-block;

}

nav a {

**display**: block;

**padding**: 6px 12px;

**text-decoration**: none;

}

### Partials

You can create partial Sass files that contain little snippets of CSS that you can include in other Sass files. This is a great way to modularize your CSS and help keep things easier to maintain. A partial is simply a Sass file named with a leading underscore. You might name it something like **\_partial.scss**. The underscore lets Sass know that the file is only a partial file and that it should not be generated into a CSS file. Sass partials are used with the **@import** directive.

### Import

CSS has an import option that lets you split your CSS into smaller, more maintainable portions. The only drawback is that each time you use **@import** in CSS it creates another HTTP request. Sass builds on top of the current CSS **@import** but instead of requiring an HTTP request, Sass will take the file that you want to import and combine it with the file you’re importing into so you can serve a single CSS file to the web browser.

In our SASS directory, we have two scss files: **\_reset.scss** and **main.scss**. We want to import **\_reset.scss** into **main.scss**.

*// \_reset.scss*

html**,**

body**,**

ul**,**

ol {

**margin**: 0;

**padding**: 0;

}

*// base.scss*

**@import** 'reset';

body {

**font**: 100% Helvetica**,** sans-serif;

**background-color**: #efefef;

}

Notice we’re using **@import ‘reset’;** in the **main.scss** file. When you import a file you don’t need to include the file extension **.scss**. Sass is smart and will figure it out for you. When you generate the CSS you’ll get:

html**,** body**,** ul**,** ol {

**margin**: 0;

**padding**: 0;

}

body {

**font**: 100% Helvetica, sans-serif;

**background-color**: #efefef;

}

## Mixins

Some things in CSS are a bit tedious to write, especially with CSS3 and the many vendor prefixes that exist. A mixin lets you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible. A good use of a mixin is for vendor prefixes. Here’s an example for **border-radius**.

**@mixin** **border-radius**($radius) {

-webkit-border-radius: $radius;

-moz-border-radius: $radius;

-ms-border-radius: $radius;

**border-radius**: $radius;

}

**.box** { **@include** **border-radius**(10px); }

To create a mixin you use the **@mixin** directive and give it a name. We’ve named our mixin **border-radius**. We’re also using the variable **$radius** inside the parentheses so we can pass in a radius of whatever we want. After you create your mixin, you can then use it as a CSS declaration starting with **@include** followed by the name of the mixin. When your CSS is generated it’ll look like this:

**.box** {

**-webkit-border-radius**: 10px;

**-moz-border-radius**: 10px;

**-ms-border-radius**: 10px;

**border-radius**: 10px;

}

## Extend/Inheritance

This is one of the most useful features of Sass. Using **@extend** lets you share a set of CSS properties from one selector to another. It helps keep your Sass very DRY (“Don’t Repeat Yourself”). In our example we’re going to create a simple series of messaging for errors, warnings and successes.

**.message** {

**border**: 1px solid #ccc;

**padding**: 10px;

**color**: #333;

}

**.success** {

**@extend** **.message**;

**border-color**: green;

}

**.error** {

**@extend** **.message**;

**border-color**: red;

}

**.warning** {

**@extend** **.message**;

**border-color**: yellow;

}

What the above code does is allow you to take the CSS properties in **.message** and apply them to **.success**, **.error**, & **.warning**. The magic happens with the generated CSS, and this helps you avoid having to write multiple class names on HTML elements. This is what it looks like:

**.message,** **.success,** **.error,** **.warning** {

**border**: 1px solid #cccccc;

**padding**: 10px;

**color**: #333;

}

**.success** {

**border-color**: green;

}

**.error** {

**border-color**: red;

}

**.warning** {

**border-color**: yellow;

}

## Operators

Doing math in your CSS is very helpful. Sass has a handful of standard math operators like **+**, **-**, **\***, **/**, and **%**. In our example we’re going to do some simple math to calculate widths for an **aside** & **article**.

**.container** { **width**: 100%; }

article**[**role**=**"main"**]** {

**float**: left;

**width**: 600px **/** 960px **\*** 100%; *// gets 600px as a percentage of 960px*

}

aside**[**role**=**"complementary"**]** {

**float**: right;

**width**: 300px **/** 960px **\*** 100%; *// gets 300px as a percentage of 960px*

}

We’ve created a very simple fluid grid, based on 960px. Operations in Sass let us do something like take pixel values and convert them to percentages without much hassle. The generated CSS will look like:

**.container** {

**width**: 100%;

}

article**[**role**=**"main"**]** {

**float**: left;

**width**: 62.5%;

}

aside**[**role**=**"complementary"**]** {

**float**: right;

**width**: 31.25%;

}

Content Sourced from: [Offical SASS Documentation](http://sass-lang.com/)

© 2017 - Seneca School of ICT