CSS is older than the hills, almost as old as HTML itself, but it provides **dramatic** improvements in **style** and **sanity** in your R Shiny applications. And it will save you time!  In this post I’ll give you a quick overview of CSS; I’ll show you how to add CSS to R Shiny; and finally I’ll introduce you to SASS, which is a pre-processor of CSS.  It gives you a subset of functions, variables and other structures, that when compiled becomes normal CSS.

Are you convinced? Good!  Let’s take a look at the possibilities. CSS stands for Cascading Style Sheets. It describes how HTML elements are to be displayed (in any media). Every Cascading Style Sheet (whether it is contained in a .css file, or embedded in the head element of an HTML document) is a series of instructions called statements. Every CSS statement consists of two parts. There is the outside part of the bracket, which is the selector for the elements and a set of rules for the elements. The outside part tells you which elements will be affected, and the inside tells you the rules that are applied.

One of the best things about CSS is you can one set of rules that allows you to control the layout of multiple web pages all at once! As long as you keep the pages consistent, you don’t have to re-do any of the style!

There are currently 3 main ways you can add CSS styling to your code:

1. Add styling directly to HTML tags

2. Add CSS to your HTML header

3. Add style sheets with the www directory

But if we are to “cut to the chase,” the best way is to **add style sheets to your www directory**. But I do want you to know about the other options, just in case you encounter it, and so that you can judge for yourself which method works best for your various projects.

**1. Add styling directly to HTML tags… here is an example:**

shinyUI(fluidPage(

headerPanel(

h1("New Application",

style = "font-weight: 500; color: #4d3a7d;"))

))

The downsides to adding styling directly to HTML tags are the following:

– They are easy to lose track of in large projects

– Impossible to reuse in different objects. If you are using the element in more than one place, you will have to repeat the styling again.

**2. Add CSS to your HTML header**

ui <- shinyUI(semanticPage(

tags$head(

tags$style(HTML("

@import url('my-font.url');

h1 {

font-weight: 500;

color: #48ca3b;

}

This is a tiny bit better, because you are defining it outside of the actual HTML elements, but at the same time, because you are still doing it in the code, you are not using a separate file. And because you’re not using a separate file, you cannot cache this. That means you are still missing some of the options that make CSS really good.

**3. Add style sheets with the www directory**

ui <- shinyUI(semanticPage(

tags$head(

tags$link(rel = "stylesheet", type = "text/css", href = "styles.css")

)

))

Adding a separate file makes the code much cleaner. You have a completely separate place for all of the rules. And you can add more than one file, s you can add a bit of structure to however you want to style your stuff.

Other benefits:

* Allows code to be reused
* Allows caching

If you want to use CSS in your Shiny projects, then you may want to consider an ‘awesome’ approach to producing CSS…

**SASS vs. CSS**

So what about SASS vs. CSS?  First of all, what is SASS? SASS (Syntactically Awesome Style Sheets) is a pre-processor of CSS; SASS code always compiles into CSS. In other words, SASS is a way of writing CSS without writing CSS. It gives you a subset of functions, variables and other structures, that when compiled becomes normal CSS. The difference is that SASS provides other tools that make more sense for a developer.  SASS is **CSS for programmers**.

The four big differences between the two are in the table below.

SASS is object-oriented, so you don’t have simple statements. You have actual objects that work in the SASS environment, such as conditions, variables, and loops.  It works more as a language, and not as a syntax. SASS allows nesting, which means that you can define elements inside of other elements.

If you know a bit of CSS, this means that whenever you define an element inside of another element, you are saying that the full rule needs to be the element inside of other elements.

So instead of creating multiple rules, you can just create one. Because it’s object oriented, it allows you to define sets of properties outside of the actual statements, and reuse them multiple times.

**MIXIN – reuse statements**

SASS has 3 big advantages. The first advantage is the **Mixin**. It’s basically a statement that doesn’t really exist in your code but can be used as an object with multiple attributes.

Take for example, displaying an HTML element as a Flex element. In this case we have a typical CSS statement. We have a row class that is being given some CSS properties.

.row {

display: -webkit-flex;

display: flex;

}

We can extract these properties into the **Mixin**. And you do this by adding the Mixin keyword from SASS.  You name it something and you add whatever rules you want inside.

Creating a Mixin:

@mixin flex {

// write the css here

display: -webkit-flex;

display: flex;}

And then when you are creating your SASS code, instead of setting every attribute by hand, you simply include the Mixin into the statement. If you want to use it 10 times, you just include it 10 times instead of defining it 10 times!

Using a Mixin:

.row {

@include flex;

}

.column {

@include flex;

}

**SASS: FUNCTIONS**

The second big advantage of SASS is functions –SASS actually allows you to define a function and it works in a way that you probably find familiar:  name of the function, parameters, body of the function, and return statement.

So you can define a function by using the **@function**. You give it a name. You give it some parameters. And you give it a return statement.

For example, here is a function that defines the width of an element:

@function column-width($col, $total:8) {

@return percentage($col/$total);

}

.col-3 {

width: column-width(3);

}

Generated CSS:

.col-3 {

width: 37.5%;

}

SASS allows you to use a variety of functions that are already pre-built such as percentage, sum, and average. In the above case we are creating a function that takes two parameters: a ‘col’ and optional ‘total’ (if not specified the default value is set to 8). And it returns the number divided by the total formatted as a percentage. It lets you create a width that works a bit like the Bootstrap columns. You have a total of how many columns in the project, and then you say “from this specific class I want it to occupy X amount of the screen.”

**LOOPS**

The third thing that SASS does really well is that you can make LOOPS. You can make loops and conditions. Loops are an easy way to create large amounts of CSS without having to repeat yourself. If, for example, we use the same function from before,

We can put that inside of a loop…

@for $i from 1 through $total {

.col-#{$i} { width: column-width($i) };

}

…and create one class for each of those possible widths that we talked about previously.

.col-1 {

width: column-width(1);

}

.col-2 {

width: column-width(2);

}

.col-3 {

width: column-width(3);

}

.col-4 {

width: column-width(4);

}

.col-5 {

width: column-width(5);

}

.col-6 {

width: column-width(6);

}

.col-7 {

width: column-width(7);

}

.col-8 {

width: column-width(8);

}

**How to Use sass in R**

R Studio released this really cool library called `sass`:

.install.packages("sass")

`sass` lets you do all of this coding that I discussed before without having to use an external tool for it.  So the main way you can use it is that the library exposes a function called “sass.” And this function takes either a string or a file and returns compiled CSS for whatever you fed it.

library(sass)

sass("

$weight: 500;

h1 { font-weight: $weight; }

h2 { font-weight: $weight \* 0.8; }

")

So that means you can define it directly in the code, wherever you want. You can have variables that are set in the application, and then the styling is defined whenever you **run** the application. If you have, for example, a variable that defines the background color of the application, you can then generate the actual styling rules just for that.

The most typical way, and the way I think you should use it, is by actually giving it a sass (.scss) file. A sass file can have a lot of different things.

Below you can see an example of a file structure you can have in a project. This is the one I usually like to use. It basically consists of a main file and then all of the other files that have small parts of the code and are usually divided according to some logic.

/styles

|

| main.scss

│

└─── modules#####

│

│ \_config.scss

│

│ \_functions.scss

│

│ \_mixins.scss

│

│ \_static.scss

│

...

│

└─── partials

│

│ \_reset.scss

│

│ \_layout.scss

│

│ \_component.scss

│

│ \_animations.scss

│

...

│

└─── vendor

│

│ some\_library.css

│

│ another\_font.css

│

...

│

...

The main file is usually the file that passes the function. It lets you import every single file that you want. Here is an example:

// Modules and Variables.

@import "modules/config";

@import "modules/mixins";

@import "modules/functions";

@import "modules/static";

// Partials.

@import "partials/\_reset";

@import "partials/\_layout";

@import "partials/\_component";

@import "partials/\_animations";

// Third-party.

@import "vendor/some\_library";

@import "vendor/another\_font";

One important thing — the **order** matters. If you want to have a file with all of your variables defined in the beginning, it should be the **first** file that you pass. And then you can use all of those variables in the rest of the files. Here is an example:

// Color pallet.$primary-red: rgb(217, 30, 73);

$primary-dark-red: rgb(166, 43, 78);

$primary-gray: rgb(147, 149, 151);

$primary-light-gray: rgb(167, 169, 172);

// Fonts

$main-font: 'Noto Sans', sans-serif;

// Sizes (in px).

$top-navigation-height: 0;

$bottom-navigation-height: 65;