Tutorial **Sass**

alfonso marin

Tomado de: https://alfonsomarin.com/sass/

Si desarrollas con Ruby, para tí Sass será un viejo conocido, pero si desarrollas para otras plataformas puede que te pase como a mí, que aun conociendo de su existencia y sabiendo de sus virtudes te cueste dar el paso a empezar a utilizarlo, pues son muchos años escribiendo CSS directamente.

Para solucionar ese problema, he decidido introducirme a fondo en esta tecnología e intentar desarrollar un tutorial más amplio del que se pueda encontrar directamente en la página oficial de SASS.

Este tutorial lo dividiré en varios posts e intentaré incluir toda la información que he podido capturar y recopilar sobre esta tecnología.

I. Instalación y primeros pasos

¿Qué es Sass?

Si todavía hay algún despistado que no sabe de lo que estamos hablando, SASS es un metalenguaje para escribir hojas de estilo que nos ayudará a generar ficheros CSS más optimizados, incorporando mayor contenido semántico a nuestras hojas de estilo y permitiendo utilizar funcionalidades que normalmente encontraríamos en lenguajes de programación tradicionales, como el uso de variables, creación de funciones, estructuración de código en varios ficheros, etc.

Sass no es una tecnología nueva, nació en el 2007 y ha ido ganando popularidad con el tiempo hasta convertirse en una herramienta imprescindible a la hora de construir proyectos web de medio o gran tamaño.

Escrito en Ruby, consta de un compilador encargado de traducir nuestros ficheros en sintaxis Sass a CSS. A la hora de escribir nuestras hojas de estilo podremos hacer uso de dos sintaxis: **SCSS** (Sassy CSS) u **Original Sass** .

El primero pretende asemejar la sintaxis Sass a la sintaxis CSS estándar, haciendo que cualquier fichero CSS sea sintácticamente válido a nivel de Sass.

Por otra parte, Original SASS es la sintaxis original de Sass que elimina muchos de los elementos CSS que se consideran inncesarios como las llaves o los puntos y coma, obligando a estructurar nuestras reglas de cierta forma para que la semántica que aportan dichos elementos quede implícita en la estructura.

Para que nos quede más claro podemos ver un ejemplo de ambas sintaxis:

[one_third]

```
/* SCSS */
p{
  width: 500px;
  color: #000;
}
```

[/one_third] [one_third_last]

```
/* Original SASS */
p
  width: 500px
  color: #000
```

[/one_third_last]

Como podemos apreciar, SCSS es idéntico a CSS, pero Original Sass simplifica la sintaxis a base de obligar a seguir una estructura más estricta.

Aunque la sintaxis original queda más limpia y se comprende mejor una vez que te acostumbras, normalmente se recomienda hacer uso de SCSS, especialmente en los inicios pues se tiene la ventaja de que realmente es CSS estándar ampliado con las mejoras ofrecidas por SASS.

En este tutorial, los ejemplos se mostrarán usando SCSS.

Instalación de SASS

Para poder utilizar SASS necesitamos disponer del compilador que nos permita convertir nuestros ficheros en sintaxis SCSS a CSS, y para instalar dicho compilador debemos instalar previamente Ruby.

Los usuarios de Windows podrán hacerlo fácilmente a través de http://rubyinstaller.org, los usuarios de Mac OS X ya lo llevan instalado de serie, y los usuarios de Linux podrán instalarlo con su gestor de paquetes preferido.

Una vez instalado Ruby y su gestor de paquetes Ruby Gems, instalar Sass será tan sencillo como introducir esta línea en nuestra consola de comandos:

```
$ gem install sass
```

Sencillo, ¿verdad? Para comprobar que funciona, podemos coger cualquier fichero css que tengamos y cambiar su extensión a scss para luego pasárselo como parámetro al comando sass:

```
$ mv test.css test.scss
$ sass test.scss
p {
  width: 500px;
  color: #000; }
```

Como hemos comentado anteriormente, la sintaxis SCSS es una ampliación de la sintaxis CSS, por lo que cualquier contenido CSS también es contenido SCSS válido.

En el ejemplo anterior, simplemente ha formateado la salida porque no hemos hecho uso de ningún uso específico de Sass. Si queremos que el contenido compilado se guarde en un fichero, se lo indicaremos como segundo parámetro:

```
&gt; sass test.scss test.css
```

Invocar este comando cada vez que modifiquemos el fichero .scss original para generar el código CSS es excesivamente tedioso. Por ese motivo, podremos hacer uso de la opción –watch del compilador, la cual monitorizará los cambios del fichero original y lo compilará automáticamente al fichero destino en caso de actualización:

```
&gt; sass --watch test.scss:test.css
```

De esta forma, podremos modificar el fichero test.scss desde nuestro editor preferido y cada vez que guardemos el fichero se compilará automáticamente a CSS.

Como veremos en este tutorial, una de las recomendaciones a la hora de trabajar con Sass es segmentar nuestra maquetación CSS en varios ficheros, de forma que podamos hacerla lo más modular posible.

Si vamos a hacer uso de un conjunto de ficheros .scss, en vez de monitorizar fichero a fichero será preferible monitorizar le directorio que los contiene.

Normalmente se recomienda tener un directorio con los ficheros Sass y otro directorio con los ficheros compilados a CSS, para luego hacer uso de la opción –watch y así monitorizar todos los cambios que se produzcan en cualquiera de los ficheros situados en el directorio SCSS:

```
&gt; sass --watch stylesheets/scss:stylesheets/css
```

Herramientas Visuales

Si te da alergia la línea de comandos, o simplemente te atraen más las herramientas de escritorio, puedes hacer uso de herramientas visuales que también realizan la compilación automática de ficheros.

Existen varias alternativas multiplataforma de pago como CodeKit o Compass App. Yo os recomiendo **Scout**, que es multiplataforma y además es gratuita.

Lo único que tendremos que crear un proyecto indicándole donde tenemos situados los fuentes y posteriormente indicarle el directorio donde alojaremos los ficheros fuente .scss y el directorio donde compilar. Una vez configurado, en la sección de «log» podremos ver la salida de las conversiones automáticas que realizará el programa.

Conclusiones

Realmente no hemos visto casi nada de Sass, únicamente hemos preparado el entorno y hemos visto las herramientas que podemos utilizar.

En el siguiente artículo veremos la sintaxis Sass a fondo y explicaremos qué funcionalidades nos ofrece esta tecnología a la hora de crear nuestras hojas de estilo.

II. Anidación, Variables, Funciones e Importación

Tras ver cómo instalar Sass y cómo configurar nuestro entorno de trabajo, ha llegado el momento de analizar en profundidad qué funcionalidades nos ofrece y descubrir la potencia que esconde este preprocesador CSS. En este artículo analizaremos la sintaxis del metalenguaje y comprobaremos cómo podremos generar hojas de estilo más ricas a nivel semántico y mucho más eficientes.

Anidación (nesting)

La principal característica de Sass es poder definir reglas de maquetación de forma anidada, evitando que tengamos que repetir constantemente los prefijos de alcance en los selectores CSS.

Como ya sabrás, uno de los problemas de los selectores CSS es que cuanto más específicos sean estos, más tendremos que repetir una y otra vez la cadena de elementos que conforman el selector, y según queramos añadir bloques a elementos inferiores tendremos que repetir cada vez la cadena de selección, como se puede ver en el siguiente ejemplo:

```
#content            { border: 1px solid black; }
#content p.info     { color: #fff; }
#content p.info a   { text-decoration: none; }
```

Como se puede observar, tenemos que ir repitiendo los elementos base del selector según vamos estilizando los elementos más internos. Esto tiene 3 principales problemas:

1. El documento CSS se hace poco legible
2. Tenemos que hacer un uso excesivo de elementos repetitivos y por consiguiente copy/paste, lo cual induce a errores
3. A la larga tendemos a ser más vagos y por pereza vamos haciendo selectores más pobres para «ahorrar tecla»

Sass evita estos inconvenientes ofreciéndonos la posibilidad de anidar unos selectores dentro de otros. Veamos cómo se escribiría el ejemplo anterior en Sass:

```
#content {
  border: 1px solid black;
  p.info {
    color: #fff;
    a { text-decoration:none;}
  }
}
```

Como podemos ver, no tendremos que repetir las cadenas de selección completas pues **Sass** se encargará de introducirlas cuando lo compilemos a CSS, con lo cual no solo ganamos en limpieza, sino que el documento queda estructurado de una forma más natural, pues el anidamiento de selectores está más alineado con el anidamiento real que esperamos encontrar en el documento HTML que estamos estilizando.

Sass quiere evitar repitamos elementos en nuestro maquetación CSS. Por ese motivo, además de anidar selectores también podremos anidar propiedades de forma que no tengamos que repetir constantemente cosas como «border-left». Podemos verlo en este ejemplo:

[one_half]

```
/* SCSS */
.bordemolon {
  border: {
    style: solid;
    left: {
      width: 4px;
      color: #888;
    }
    right: {
      width: 2px;
      color: #ccc;
    }
  }
```

[/one_half] [one_half_last]

```
/* CSS generado */
.bordemolon {
  border-style: solid;
  border-left-width: 4px;
  border-left-color: #888;
  border-right-width: 2px;
  border-right-color: #ccc; }
```

[/one_half_last]

Aunque en estos ejemplos hemos anidado visualmente los elementos en los ficheros fuente, esto no es obligatorio en la sintaxis SCS (en la sintaxis Original

Sass sí), pero ya que son ficheros fuente que luego compilaremos, es recomendable realizar el tabulado visual para mejorar la legibilidad.

Uso de referencias en anidaciones

Sass también soporta el concepto de «hijo directo» que podemos representar en CSS con el símbolo >.

En el siguiente ejemplo se puede ver cómo podemos hacer uso de este operador:

[one_half]

```
/* SCSS */
.blog &gt; {
  .post {
    width: 800px;
    &gt; .title{
      font-weight: bold;
    }
  }
  .comments{
    margin-left: 20px;http://test.alfonsomarin.com/wp/movil/-/tutorial-sass-ii-sintaxis-basica/
  }
}
```

[/one_half] [one_half_last]

```
/* CSS generado */
.blog &gt; .post {
  width: 800px;
}
.blog &gt; .post &gt; .title {
  font-weight: bold;
}
.blog &gt; .comments {
  margin-left: 20px;
}
```

[/one_half_last]

Uso de referencias en anidación

Al compilar nuestro código a SCSS a CSS, **Sass** prefijará por norma general cualquier selector anidado con la concatenación de sus selectores padre, siempre concatenándolos con espacios como se puede ver en este ejemplo:

[one_half]

```scss
/* SCSS */
.blog {
  .post p{
    em{
      color: #fff;
    }
  }
}
```

[/one_half] [one_half_last]

```scss
/* CSS generado */
.blog .post p em {
  border: color: #fff;
}
```

[/one_half_last]

Esta es la regla general, pero un selector anidado puede indicar exactamente cómo quiere que se le añadan su selectores padre gracias al operador &, el cual representa precisamente a su selector padre.

El símbolo & lo podemos poner en cualquier posición de nuestro selector, y si no lo ponemos es como si realmente lo pusiésemos al principio y seguido de un espacio, es decir, el comportamiento por defecto: ponme mi selector padre delante de mí separado por un espacio.

Para que quede más claro, estas dos versiones del ejemplo anterior serían equivalentes:

[one_third]

```scss
/* v. 1: &amp;amp; implícito */
.blog {
  .post p {
    em {
        color: #fff;
    }
  }
}
```

[/one_third] [one_third]

```scss
/* v. 2: &amp;amp; explícito */
.blog {
  &amp;amp; .post p {
    &amp;amp; em {
        color: #fff;
    }
  }
}
```

[/one_third] [one_third_last]

```
/* CSS Generado */
.blog .post p em {
  color: #fff;
}
```

[/one_third_last]

A continuación se muestran unos ejemplos donde indicamos que se añadan los selectores padre en posiciones específicas. Merece especial atención el último ejemplo, pues gracias a esta técnica conseguimos definir selectores con pseudoclases tipo «:hover» gracias a que eliminamos el espacio de concatenación por defecto.

[one_half]

```
/* Insertar .blog entre .post y p */
.blog {
  .post &amp;amp; p {
    em {
        color: #fff;
    }
  }
}

/* Insertar al final */
.blog {
  .post p {
    em &amp;amp; {
        color: #fff;
    }
  }
}

/* Pseudoclase :hover en p */
.blog {
  .post p {
    &amp;amp;:hover em {
        color: #fff;
    }
  }
}
```

[/one_half] [one_half_last]

```
/* CSS Generado */
/* Insertar .blog entre .post y p */
.post .blog p em {
  color: #fff;
}

/* Insertar al final */
```

```
em .blog .post p {
  color: #fff;
}

/* Pseudoclase :hover en p */
.blog .post p:hover em {
  color: #fff;
}
```

[/one_half_last]

Variables

Una de las principales carencias de CSS es la imposibilidad de definir variables, teniendo que especificar una y otra vez aquellos valores que queramos aplicar de forma general a varios elementos. Un caso típico suelen ser el conjunto de colores base del tema que estamos maquetando.

Sass introduce la posibilidad de definir variables, donde las especificaremos prefijándoles el símbolo $. Estas variables se comportan como atributos CSS, y su valor puede ser cualquier valor que pudiera adquirir cualquier atributo CSS. Veamos unos ejemplos:

[one_half]

```
$color_link: blue;
$default_border: 1px solid black;
$std_margin: 5px;
a {
  color: $color_link;
  border: $default_border;
  margin: $std_margin 0px $std_margin 0px;
}
```

[/one_half] [one_half_last]

```
/* CSS Generado */
a {
  color: blue;
  border: 1px solid black;
  margin: 5px 0px 5px 0px;
}
```

[/one_half_last]

Una variable se podrá definir fuera o dentro de algún selector. Si se define fuera, dicha variable será global y podrá utilizarse en cualquier bloque, pero si se define dentro de un selector, la variable será local y únicamente se podrá utilizar en el selector que la contiene y en sus selectores anidados.

A continuación se muestra un ejemplo se muestra el uso de una variable global ($color_link) y una variable local ($var_local). En el propio ejemplo se muestra una asignación comentada que provocaría un fallo de compilación al intentar asignar una variable local fuera de su alcance.

[one_half]

```
$color_link: blue;
p{
  a {
    color: $color_link;
    $var_local: white;
    &amp;.link{
      color: $var_local;
    }
}
 /* color: $var_local;
```

[/one_half] [one_half_last]

```
/* CSS Generado */
p a {
  color: blue;
}
p a.link {
  color: white;
}
```

[/one_half_last]

Una buena práctica común consiste en definir todas las variables globales al principio del fichero, para que puedan localizarse rápidamente. Incluso en proyectos de gran envergadura, es común extraer todas las variables globales en un fichero exclusivo.

A la hora de definir una variable, podemos hacer uso de la directiva **!default** al final de la misma. Esta directiva indicará que la asignación que estamos realizando a la variable solo se haga en caso de que dicha variable no se haya definido anteriormente.

Esta funcionalidad es especialmente interesante utilizarla en bloques de código Sass que queramos reutilizar, pues si todas las variables implicadas incluyen esta directiva permitiremos que se puedan personalizar sus valores sin necesidad de modificar el código en sí.

Operaciones aritméticas

Otra funcionalidad que nos ofrece **Sass** es poder realizar operaciones aritméticas sobre los valores de las propiedades.

Por ejemplo, podríamos hacer width: 500px * 0.5, de forma que se calcularía 250px para la propiedad width.

Podemos utilizar los 4 operadores aritméticos +, -, * y / (suma, resta, multiplicación y división), y Sass siempre respetará las unidades de las propiedades (px, em, etc…), salvo que éstas entren en conflicto, como por ejemplo si intentamos multiplicar una cantidad especificada en 'px' con otra especificada en 'em'.

Las operaciones aritméticas son realmente interesantes si las combinamos con las variables explicadas en el punto anterior, pues nos permiten estructurar el *layout* de nuestro proyecto en base a un conjunto de valores fijos, de forma que si estos cambian, todo seguirá estando bien definido de forma proporcional.

Veamos un ejemplo de esto: supongamos que queremos crear un menú horizontal de ancho fijo en el que todos los botones tengan el mismo ancho:

[one_half]

```
$width-menu: 500px;
$num-botones: 10;

.menu {
  width: $width-menu;
  .boton{
    width: $width-menu / $num-botones;
  }
}
```

[/one_half] [one_half_last]

```
/* CSS Generado */
.menu {
  width: 500px;
}
.menu .boton {
  width: 50px;
}
```

[/one_half_last]

Como podrás imaginar, lo interesante del ejemplo anterior es que si cambia el ancho del menú o el número de botones, solo tendremos que modificar el valor de las variables y todo el *layout* se recalculará automáticamente.

Funciones

Además de usar operaciones aritméticas para calcular valores de propiedades, también podremos hacerlo utilizando distintas funciones que **Sass** nos proporciona. El listado completo de funciones podemos encontrarlo en la guía de referencia de Sass.

Entre todas las funciones ofrecidas por Sass, las más utilizadas son las relacionadas con operaciones sobre colores. Por ejemplo, podríamos aclarar un color dado (lighten), oscurecerlo (darken), saturarlo (saturate), etc. También existen funciones para tratar con valores numéricos (abs, max, min, …) o con cadenas (length, join, …)

Vamos a ver un ejemplo de cómo se utiliza este tipo de funciones:

[one_half]

```
$btn-bg-color: #ce4dd6;
.menu {
  .button {
    background-color: $btn-bg-color;
    &amp;:hover{
      background-color: lighten($btn-bg-color, 20%);
    }
    &amp;:active{
      background-color: darken($btn-bg-color, 20%);
    }
    &amp;.disabled{
      background-color: grayscale($btn-bg-color);
    }
  }
}
```

[/one_half] [one_half_last]

```
/*CSS Generado*/
.menu .button {
  background-color: #ce4dd6;
}
.menu .button:hover {
  background-color: #e5a0e9;
}
.menu .button:active {
  background-color: #93239a;
}
```

```
.menu .button.disabled {
  background-color: #929292;
}
```

[/one_half_last]

@import

Como desarrolladores web sabemos que lo ideal es tener todas nuestras reglas CSS en un único fichero para así no penalizar la carga de nuestras páginas, pero por otra parte nos gustaría desglosarlo en distintos ficheros agrupando aquellas reglas que semánticamente estén relacionadas, y así evitar tener que lidiar con un único fichero CSS gigantesco.

Una vez más **Sass** nos facilita la vida ofreciéndonos lo mejor de ambos mundos. Por una parte, podremos desglosar las reglas de nuestro proyecto en tantos ficheros como deseemos, y luego indicarle a Sass que compile dichos ficheros por separado o todos en un único fichero CSS.

La instrucción que nos permitirá hacer esto es **@import «fichero»**, a la cual le indicaremos el nombre del fichero (sin la extensión) que queremos importar. **Sass** compilará el fichero que le hayamos indicado y lo insertará en el fichero original sustituyendo la línea del @import por el contenido compilado, como se puede ver en este ejemplo:

[one_half]

```
/* Contenido de fichero colors.scss */
$color: #aaa;
p {
  em{
    color: $color
  }
}
/* Contenido de fichero test.scss */
@import "colors";

p.test{
  em {
    color: $color;
  }
```

```
}
```

[/one_half] [one_half_last]

```
/* Contenido generado test.css */
/* line 4, ../scss/_colors.scss */
p em {
  color: #aaaaaa;
}

/* line 4, ../scss/test.scss */
p.test em {
  color: #aaaaaa
```

[/one_half_last]

Aunque solo se muestra el contenido del fichero generado test.css, si estuviésemos monitorizando el directorio donde se encuentran estos ficheros .scss (usando Scout o el comando sass –watch, como vimos en el tema anterior) se generarían dos ficheros .css: test.css y colors.css.

Si no queremos que se generen ficheros separados para un determinado fichero .scss que unicamente utilizarmos para importarlo en otros, como podría ser el caso de colors.scss, podremos renombrar el fichero e insertar un _ al principio.

De esta forma, le estaremos indicando al compilador que no deseamos fichero .css de salida para dicha fuente, y semánticamente estamos indicando que ese fichero se utilizará únicamente para ser importado, no para generar código css por sí mismo. Por ese motivo, para diferenciarlos de los ficheros normales se les denomina *partials.*

Conclusión

En este tema hemos visto las primeras herramientas que nos proporciona **Sass** en su sintaxis y está claro lo mucho que nos pueden ayudar a la hora de crear nuestras hojas de estilo de una forma mucho más estructurada y eficiente. En el siguiente tema terminaremos de ver el resto de sus elementos.

III. Mixins, Interpolación y directivas de control

Seguimos analizando las funcionalidades ofrecidas por Sass donde empezaremos viendo los mixins, una de las más útiles a la hora de trabajar con este preprocesador. También veremos cómo funciona la interpolación y conoceremos algunas herramientas más que nos ofrece su sintaxis como los bucles @each o los condicionales @if.

Mixins

Un *mixin* es un fragmento de código Sass que podremos reutilizar para aplicar a los selectores que deseemos. Podríamos verlos como recetas donde agruparemos distintas reglas que luego se aplicarán a aquellos selectores a los que se asocie el *mixin.*

Para definirlos utilizaremos la palabra clave @mixin seguida del nombre que le queramos dar, seguido del bloque de reglas que queramos incoporar dentro de él:

[one_half]

```
// SCSS
$azulado: #165BFF;

@mixin link_chulo{
  color: $azulado;
  text-decoration: none;
  &amp;:hover{
        color: lighten($azulado, 20);
  }
}

.content {
  a {
    @include link_chulo;
  }
}
```

[/one_half]
[one_half_last]

```
/* CSS generado */
.content a {
  color: #165bff;
  text-decoration: none; }
  .content a:hover {
    color: #7ca3ff; }
```

[/one_half_last]

En este ejemplo hemos definido un *mixin* 'link_chulo' que hemos asignado a un determinado selector a través de *@include*.

Como ves, el mixin es un bloque de reglas reutilizable que podremos aplicar a tantos selectores queramos.

También he querido mostrar que dentro de un *mixin* podremos hacer uso de variables y que también podemos anidar las reglas, como hemos hecho con *$azulado* y *&:hover* respectivamente.

Los *mixins* son muy apropiados a la hora de aplicar propiedades que no son estándar o debamos incluir varias versiones con prefijos que puedan funcionar en distintos navegadores. Por ejemplo, si queremos definir un borde redondeado de 5 píxeles de radio, podríamos definir el siguiente *mixin*:

```
@mixin bode_redondeado_5px{
  border-radius: 5px;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  -o-border-radius: 5px;
}
.redondeado{
  @include borde_redondeado_5px;
}
```

Como podrás intuir, de esta forma generaremos un código más limpio y menos propenso a errores, ya que únicamente tendremos que incluir una línea cada vez que queramos el mismo borde de 5 píxeles de radio.

El problema en el ejemplo anterior es que solo podremos generar bordes de 5 píxeles de radio, teniendo que reescribir el mixin si queremos bordes de otro tamaño diferente. Para evitar eso podemos hacer uso de argumentos o parámetros como si de una función se tratase, de forma que podamos especificar qué valores deseamos que se apliquen dentro del mixin.

Podremos definir tantos argumentos como queramos, le indicaremos un nombre y opcionalmente un valor por defecto. A continuación se muestra un ejemplo donde hemos reescrito el mixin anterior para que soporte una variable, indicando además que su valor por defecto será 5px;

```
@mixin bode_redondeado($size: 5px){
  border-radius: $size;
```

```
  -moz-border-radius: $size;
  -webkit-border-radius: $size;
  -o-border-radius: $size;
}
.redondeado{
  @include borde_redondeado(10px);
}
```

@content: pasando bloques de contenido a un mixin

A la hora de utilizar un mixin, es posible definir un bloque de estilos específico que el mixin podría incorporar mediante la directiva @content. Veamos un ejemplo:

[one_half]

```
// SCSS
@mixin apply-to-ie6-only {
  * html {
    @content;
  }
}
@include apply-to-ie6-only {
  #logo {
    background-image: url(/logo.gif);
  }
}
```

[/one_half]
[one_half_last]

```
/* CSS generado */
* html #logo {
  background-image: url(/logo.gif);
}
```

[/one_half_last]

Como podemos ver, a la hora de invocar al mixin hemos definido el bloque de estilos asociado que sustituirá a la directiva @content dentro del mixin.

Interpolación

Las variables en Sass no solo las podremos utilizar para asignar valores a las propiedades, como ya hemos visto. También podremos utilizarlas para formar el nombre de las propiedades o incluso los propios selectores. Solo debemos utilizar #{$variable} allí donde queramos que sass sustituya por el valor de la variable:

[one_half]

```scss
/* SCSS */
$posicion_borde: left;
@mixin coche($marca, $color){
  .coche.#{$marca}{
          border-#{$posicion_borde}: 2px;
          background-color: $color;
          background-image: url('images/#{$marca}-#{$color}.jpg')
  }
}

@include coche('audi', 'green');
```

[/one_half]
[one_half_last]

```css
/* CSS generado */
.coche.audi {
  border-left: 2px;
  background-color: "green";
  background-image: url("images/audi-green.jpg"); }
```

[/one_half_last]

Como se puede comprobar en el ejemplo anterior, se pueden hacer uso de la interpolación en cualquier parte de nuestras reglas, donde también podremos utilizar los parámetros de los mixins pues no dejan de ser variables locales.

Directivas de control @if, @each, @for y @while

Sass nos ofrece distintas estructuras de control que nos permitirán incluir estilos en base a ciertas condiciones o construir conjuntos de estilos similares con pequeñas variaciones. Estas directivas van principalmente enfocadas en la generación de mixins o funciones reutilizables.

Por una parte tenemos @if, que nos permitirá establecer condiciones bajo las que se aplicarán las reglas o no. Estas condiciones podrán incluir comparadores típicos (==, !=, <, >) entre variables, constantes o cualquier expresión intermedia. Solo en caso de cumplirse la condición se ejecutará la generación de código del bloque asociado.

[one_half]

```scss
/* SCSS */
$animal: gato;
p {
  @if 1 + 1 == 2 {border: 2px solid black}
  @if $animal == gato {
    color: blue;
```

```
  } @else if $animal == perro {
    color: red;
  } @else if $animal == caballo {
    color: green;
  } @else {
    color: black;
  }
}
```

[/one_half]
[one_half_last]

```
/* CSS Generado */
p {
  border: 2px solid black;
  color: blue; }
```

[/one_half_last]

Las otras tres directivas, @each, @for y while, nos permitirán iterar sobre conjuntos de valores con los que podremos generar conjuntos de reglas que sean similares con pequeñas variaciones. Su estructura es muy similar a la que nos pudiéramos encontrar en los lenguajes de programación tradicionales. Veamos un ejemplo de cada una de ellas donde veremos cómo se definen y cual es su funcionamiento:

@for

Podremos definir una estructura @for de la siguiente manera:

```
@for $var from  [to|through]  {
  //Bloque de reglas donde podremos utilizar $var mediante
interpolación
}
```

$var será el nombre de la variable que queramos utilizar en nuestro bloque, tanto <*start*>como <*end*> tendrán que ser expresiones SassScript válidas que devuelvan números enteros, y por último si indicamos 'through' se tendrán en cuenta los valores <start> y <end> dentro del bucle, y si utilizamos 'to' no se tendrá en cuenta el valor <end> dentro del bucle. Veamos un ejemplo

[one_half]

```
/* SCSS */
@for $i from 1 to 3 {
  .todos-#{$i} { width: 2em * $i; }
}
```

```
@for $i from 1 through 3 {
  .casitodos-#{$i} { width: 2em * $i; }
}
```

[/one_half]
[one_half_last]

```
/* CSS Generado */
.todos-1 {
  width: 2em; }

.todos-2 {
  width: 4em; }

.casitodos-1 {
  width: 2em; }

.casitodos-2 {
  width: 4em; }

.casitodos-3 {
  width: 6em; }
```

[/one_half_last]

@each

Podemos definir una estructura @each de la siguiente manera:

```
@each $var in {
  //Bloque de reglas donde podremos utilizar $var mediante
interpolación
}
```

En este caso, <list> será cualquier expresión que devuelva una lista de elementós SassScript válida, es decir, una sucesión de elementos separados por comas. Veamos un ejemplo:

[one_half]

```
/* SCSS */
@each $animal in puma, sea-slug, egret {
  .#{$animal}-icon {
    background-image: url('/images/#{$animal}.png');
  }
}
```

[/one_half][one_half_last]

```
/* CSS Generado */
.puma-icon {
  background-image: url("/images/puma.png"); }

.sea-slug-icon {
  background-image: url("/images/sea-slug.png"); }

.egret-icon {
  background-image: url("/images/egret.png"); }
```

[/one_half_last]

@while

Para definir la directiva @while debemos asociarle una expresión SassScript que devuelva un valor booleano, y mientras dicha expresión sea cierta continuará generando los estilos del bloque interno que hayamos definido. Veamos un ejemplo.

[one_half]

```
/* SCSS */
$i: 6;
@while $i &gt; 0 {
  .item-#{$i} { width: 2em * $i; }
  $i: $i - 2;
}
```

[/one_half][one_half_last]

```
.item-6 {
  width: 12em; }

.item-4 {
  width: 8em; }

.item-2 {
  width: 4em; }
```

[/one_half_last]

Directiva @extend

En muchas ocasiones nos encontramos con la situación en la que una clase tiene todos los estilos de otra clase, además de los suyos propios.

En esos casos podremos hacer uso de @extend, el cual nos permite indicar una especie de herencia entre clases. Su funcionamiento quedará muy claro en el siguiente ejemplo:

[one_half]

```
/* SCSS */
.alerta {
  background: orange;
  display: block;
  font-weight: bold;
}
.alertaCritica{
  @extend .alerta;
  background: red;
}
```

[/one_half]
[one_half_last]

```
/* CSS generado */
.alerta, .alertaCritica {
  background: orange;
  display: block;
  font-weight: bold; }

.alertaCritica {
  background: red; }
```

[/one_half_last]

Como vemos, las dos clases del ejemplo comparen el mismo conjunto de estilos, y posteriormente se incluyen las personalizaciones de la segunda clase.

El ejemplo que acabamos de ver es de los más sencillos ya que únicamente intervienen dos clases, pero el concepto de extender se puede extender a cualquier tipo de selector siempre que este incluya un único componente, como por ejemplo a:hover, .link.disabled o a.user[href^=»https://»]. Es decir, no podremos extender selectores del tipo «a .disabled» o «.clase1 + .clase2».

Otra funcionalidad que nos ofrece Sass es la posibilidad de crear «plantillas» destinadas exclusivamente a ser extendidas, de forma que si no las utilizamos, no se generará ningún CSS asociado. Estas plantillas se definen incluyendo un selector ficticio que empieza por % y que utilizaremos como identificador de la plantilla, que será sustituido por la clase que está extendiendo a la plantilla. Veamos un ejemplo:

[one_half]

```
/* SCSS
```

```
 Esta regla no generará CSS
 por sí misma
*/
#cuerpo a%plantilla{
  font-weight: bold;
  font-size: 2em;
}

/* Usamos la plantilla */
.alerta {
  @extend %plantilla;
}
```

[/one_half]
[one_half_last]

```
/* CSS generado */
#cuerpo a.alerta {
  font-weight: bold;
  font-size: 2em; }
```

[/one_half_last]

Directiva @media

@media es una directiva CSS bastante conocida ya que se introdujo en CSS2 para definir distintos tipos de estilos según el medio (display, print, etc.), y en CSS3 se ha enriquecido con las denominadas *media queries*, las cuales permiten aplicar estilos en base determinados valores del medio, como por ejemplo el tamaño de pantalla del dispositivo.

Esta última característica ha dado paso al denominado *responsive design*, el cual permite que nuestro contenido se adapte a distintos tamaños de pantalla, dado que podremos aplicar estilos específicos a cada tamaño.

A partir de Sass 3.1 se ha introducido el concepto de **@media bubbling**, el cual nos permite definir *media queries* como si de selectores se tratase, pudiendo anidarlas dentro de nuestras estructuras de selectores.

Sass extraerá el contenido del bloque asociado y generará una condición @media donde se asociará dicho contenido al selector formado según la ruta de anidamiento que correspondiese. Veamos un ejemplo para aclarar este funcionamiento:

[one_half]

```
.sidebar {
```

```
  width: 300px;
  @media screen and (orientation: landscape) {
    width: 500px;
  }
}
```

[/one_half]
[one_half_last]

```
/* CSS generado */
.sidebar {
  width: 300px; }
  @media screen and (orientation: landscape) {
    .sidebar {
      width: 500px; }
```

[/one_half_last]

Vemos que Sass forma un bloque @media aplicando los estilos asociados (width: 500px) al selector padre (.sidebar).

Además, también podremos utilizar variables e interpolación en cualquier parte de la definición de la *media query*:

[one_half]

```
$media: screen;
$feature: -webkit-min-device-pixel-ratio;
$value: 1.5;

@media #{$media} and ($feature: $value) {
  .sidebar {
    width: 500px;
  }
}
```

[/one_half]
[one_half_last]

```
/* CSS generado */
@media screen and (-webkit-min-device-pixel-ratio: 1.5) {
  .sidebar {
    width: 500px; } }
```

[/one_half_last]

IV. Ejemplos de uso y recomendaciones

Ahora que conocemos la potencia de Sass, vamos a ver algunos casos prácticos donde podemos ver cómo podemos hacer uso de Sass a la hora de abordar operaciones cotidianas de maquetación CSS. También veremos alguna recomendación a la hora de utilizar Sass.

Ejemplos de uso

Prefijos de navegadores en propiedades CSS3

Al utilizar algunas propiedades CSS3 es conveniente incluir la definición de las versiones con prefijos de cada navegador. Una forma fácil de trabar con ellos sería definirnos un mixin en Sass para su incorporación:

[one_half]

```
@mixin vendor-prefix($name, $argument) {
  #{$name}: $argument;
  -webkit-#{$name}: $argument;
  -ms-#{$name}: $argument;
  -moz-#{$name}: $argument;
  -o-#{$name}: $argument;
}
.redondo {
  @include vendor-prefix(border-radius, 1px 1px 1px 1px)
}
```

[/one_half] [one_half_last]

```
.redondo {
  border-radius: 1px 1px 1px 1px;
  -webkit-border-radius: 1px 1px 1px 1px;
  -ms-border-radius: 1px 1px 1px 1px;
  -moz-border-radius: 1px 1px 1px 1px;
  -o-border-radius: 1px 1px 1px 1px; }
```

[/one_half_last]

Fuentes @font-face

Si tenemos un conjunto de fuentes que queramos definir a través de @font-face y las tenemos todas situadas en el mismo sitio, podemos hacer uso del siguiente bucle @each para definirlas todas a la vez:

```
@each $font-face in font-1, font-2, ...  {
```

```
    @font-face {
          font-family: $font-face; font-style: normal; font-weight:
normal;
          src: url('/path/to/font/#{$font-face}.eot'),
          src: url('/path/to/font/#{$font-face}.eot?') format('eot'),
               url('/path/to/font/#{$font-face}.woff') format('woff'),
               url('/path/to/font/#{$font-face}.ttf')  format('truetype');
    }
}
```

Cross-browser opacity

Para definir el nivel de visibilidad de un elemento utilizamos la propiedad opacity indicando el nivel de visibilidad entre 0 y 1, pero esta propiedad no está disponible en IE, donde se hace una implementación basada en valores entre 0 y 100. Sería útil definir un mixin que nos asegure una definición multinavegador:

[one_half]

```
@mixin opacity($opacity) {
  filter: alpha(opacity=#{$opacity}); // IE 5-9+
  opacity: $opacity * 0.01;
}

.shadow {
  @include opacity(50);
}
```

[/one_half] [one_half_last]

```
.shadow {
  filter: alpha(opacity=50); opacity: 0.5; }
```

[/one_half_last]

Recomendaciones

Evitar el anidamiento excesivo

Como hemos visto en este tutorial, el anidamiento es una de sus mejores características de Sass pues, ya que nos permite definir reglas sobre selectores más específicos, teniendo un control más preciso sobre los estilos de nuestro proyecto.

El problema que podemos encontrarnos ante un uso excesivo de esta característica es que tendamos a reflejar la estructura de nuestro contenido en la estructura de nuestro CSS. Para comprender mejor este problema veamos un ejemplo sobre una página web en la que pudiésemos tener este contenido:

```
<div class="contenedor">
  <div class="contenido">
    <div class="articulo">
    <h1>...</h1>
      <div class="cuerpo">
      ...
      </div>
    </div>
  </div>
</div>
```

Una estructura típica que pudiésemos encontrar en un blog. Haciendo uso del anidamiento, empezamos a añadir estilos y a hacer uso del anidamiento según vamos necesitando definir propiedades sobre cada elemento, siendo fácil acabar con una estructura SCSS muy parecida al contenido HTML original:

```
body {
  div.contenedor {
    margin: auto;
    div.contenido {
      border:1px;
          padding: 2px;
        div.articulo{
          padding: 2px;
          h1 {color: blue}
        }
      }
    }
  }
}
```

El problema es que al compilar este fichero nos encontramos una cosa como esta:

```
body div.contenedor {
  margin: auto; }
  body div.contenedor div.contenido {
    border: 1px; }
    body div.contenedor div.contenido div.articulos {
      padding: 2px; }
      body div.contenedor div.contenido div.articulos div.articulo {
        padding: 2px; }
        body div.contenedor div.contenido div.articulos div.articulo h1 {
          color: blue; }
```

Crear selectores tan complejos puede afectar al rendimiento del navegador a la hora de dibujar el *layout* de nuestra página, aparte del considerable aumento del tamaño del fichero css generado.

Otro problema añadido es que haciendo uso intensivo del anidamiento estamos ligando excesivamente nuestros estilos a la estructura del documento, generando conjuntos de reglas poco reutilizables y difíciles de mantener.

En general, hay una regla no escrita en la que se sugiere no aplicar más de 4 niveles de anidamiento. Si en algún momento nos encontramos en esta situación, esto nos debería alertar de que algo estamos haciendo mal y estamos asociando excesivamente nuestros estilos al DOM de la página. Si bien es posible que en determinadas situaciones tengamos que sobrepasar 4 niveles de anidamiento, no es una cosa que deberíamos hacer frecuentemente.

Te aconsejo que leas el siguiente artículo en The Sass Way donde explican detenidamente este problema y donde aportan algunas recomendaciones para crear nuestras reglas sin llegar a necesitar más de 4 niveles.

Creación de módulos

Ya hemos visto que Sass nos permite separar en varios ficheros nuestro código llamados *partials* y posteriormente importarlos todos en un único fichero mediante la directiva @import, o al menos aquellos que nos interesen.

Esta funcionalidad nos invita a fragmentar nuestro código Sass en conjuntos funcionales que eventualmente podamos reutilizar en varios proyectos, es decir, intentar que cada fichero represente un módulo independiente que podamos importar a nuestro proyecto solo en caso de necesitarlo.

Es importante remarcar que el concepto de módulo que aquí se discute no es nada ofrecido por Sass, sino unas recomendaciones a la hora de trabajar con *partials* que nos permitirán una mejor organización de nuestro código. Estas recomendaciones están extraídas del artículo original A Standard Module Definition for Sass.

A la hora de crear estos módulos, se pueden seguir el siguiente conjunto de recomendaciones:

1. **Cada módulo en un *partial***: debemos separar nuestros módulos en *partials*, que como vimos en el capítulo 2 eran ficheros con prefijo _ destinados a ser importados, no a generar ficheros .css.
2. **Un módulo no debe generar código por sí mismo**: es decir, un módulo solo contendrá *mixins*, funciones y variables Sass, de forma que únicamente importando dicho módulo no generaremos ningún código CSS de salida, salvo que invoquemos a algún *mixin* contenido en el módulo.

3. **Cada módulo contendrá un *mixin principal***: este *mixin* estará situado al principio del fichero y hará un uso estándar de las funciones y *mixins* que incorpora el módulo, generando un conjunto estándar de estilos del módulo.

   Por ejemplo, supongamos que tenemos un módulo para crear botones, donde hemos incorporado un conjunto de *mixins* y funciones que nos permitirán generar distintos tipos de botones, con distintos colores, tamaños, etc.

   El *mixin principal* se encargaría de generar un conjunto estándar de botones, pongamos por ejemplo, botones rojo, verde y amarillo en tamaños pequeño, mediano y grande. De esta forma, sin necesidad de conocer el funcionamiento de los mixins y funciones incorporadas en el módulo, podremos tener una gama estándar de botones ofrecidas por el módulo.

   Además, examinando el funcionamiento de dicho mixin principal podremos obtener una mejor visión de cómo se utiliza dicho módulo, en caso de queramos generar nuestros propios botones con otros tamaños y colores.

4. **Todas las variables con !default**: como explicamos en el capítulo 2, incorporando la directiva !default al final de las asignaciones en nuestras variables estaremos indicando que dicho valor únicamente se aplicará si dicha variable no se ha definido con anterioridad. Si la incluimos en todas las definiciones de variables de nuestro módulo, esteremos permitiendo que se puedan personalizar sin necesidad de modificar el contenido del módulo.

Conclusión

Con este capítulo doy por finalizado el tutorial de Sass. Espero que te haya gustado y que hayas aprendido al menos tanto como lo he hecho yo. He de reconocer que una vez que empiezas a utilizar Sass ya no te sentirás tan cómodo escribiendo sucio CSS «a pelo».

Para terminar, dejarte un par de links donde podrás encontrar más información sobre este lenguaje:

- The Sass Way: blog con artículos relacionados con Sass
- Documentacion oficial: documento de referencia donde encontrarás todo lo que se puede hacer con Sass. Además, encontrarás la definición de todas las funciones ofrecidas para tratamiento de colores, cálculos matemáticos, etc.