Ciclo Formativo de Desarrollo de Aplicaciones Web

Diseño de Interfaces Web

Vicente Javier lópez Belmonte

Preprocesamiento CSS

Introducción a los pre-procesadores

Definición de pre-procesador

Con CSS es posible diseñar sitios web exactamente como el diseñador hubiera querido, pero a medida que los sitios se van complicando, empezamos a encontrar limitaciones al CSS puro, encontrándonos con hojas de estilo de miles de líneas realmente difíciles de mantener, modificar o refactorizar.

Un pre-procesador de CSS es un lenguaje de hoja de estilos, esto es, una extensión de CSS que nos permite añadir nuevas funcionalidades para hacer el trabajo de diseño y maquetación del sitio web mucho más sencillo.

Todos los navegadores actuales pueden trabajar con CSS puro sin problemas, pero no ocurre lo mismo con lenguajes de hoja de estilos, razón por la cual es necesario procesarlos antes de incorporarlos al sitio web.

Dicho de otro modo, el código del documento pre-procesado (SASS, SCSS, LESS) debe ser convertido a CSS puro antes de realizar el despliegue del sitio web.

SASS (Syntactically Awesome Style Sheet)

Cuando Tim Berners-Lee puso las bases del HTML a principios de los 90, el mundo de Internet como lo conocemos ahora era simplemente una visión. La complejidad de muchas de las aplicaciones actuales y la forma en cómo se ha extendido el uso a todos los ámbitos y sectores de la vida diaria, desde el blog amateur hasta las grandes aplicaciones bancarias, han hecho crecer las necesidades de los desarrolladores para adaptarse a mayores requerimientos.

Los CSS nacieron algo más tarde que el HTML, al ver que todos los estilos aplicados en las etiquetas hacían poco eficiente el desarrollo. Separar los colores y los tamaños de las fuentes fue un gran paso que permite hoy en día generar una consistencia a lo largo de las páginas, centralizando y separando el contenido de como se visualiza.

Actualmente, con las novedades de HTML5, con Javascript, que da gran parte de la vida a las millones de websites que se visitan cada día enriqueciendo la experiencia a los usuarios y facilitando el trabajo a los desarrolladores, como con el framework angularjs, los CSS necesitan de alguna forma reinventarse, y es verdad que lo está haciendo actualmente, vía CSS3, cuya propagación depende de los navegadores que interpretan y dan soporte a nuevas propiedades y posibilidades.

En todo este estado de efervescencia de las tecnologías web es en la que se enmarca el preprocesador SASS (con un nombre que lo dice todo, Syntactically Awesome Style Sheets), para dar una nueva esperanza. Con estos websites más complejos de la actualidad, el trabajo en CSS tiene el peligro de crearse un código con muchos trozos repetidos e inabarcable por el número de líneas que se generan. SASS y otros preprocesadores, como Less o Stylus, te ayudan de tal forma que, cuando los usas, ya no puedes volver atrás, es un enamoramiento de por vida para volver a disfrutar de la ardua tarea de las hojas en cascada.

¿Qué hace potente a SASS?

La principal ventaja de SASS es la posibilidad de convertir los CSS en algo dinámico. Permite trabajar mucho más rápido en la creación de código con la posibilidad de crear funciones que realicen ciertas operaciones matemáticas y reutilizar código gracias a los mixins, variables que nos permiten guardar valores. SASS, en definitiva, se convierte en tu mejor ayudante.

Y algo siempre importante cuando te decantas por una herramienta con alternativas, dispone de una gran comunidad que la hace progresar, por lo que se le augura un gran futuro por delante.

El formato #SASS hace que los #CSS sean mucho más divertidos, potentes y reusables.

¿Cómo funciona?

SASS dispone de dos formatos diferentes para la sintaxis, lo que hace se traduce en dos extensiones de fichero diferentes: .sass y .scss

El primero en salir fue .sass y se caracteriza, al igual que Stylus y coffeescript para el lenguaje de programación Javascript, en no hacer uso de llaves ni punto y coma final, en busca de la simplicidad y eliminación de ruido.

```
body

background: #000

font-size: 62.5%
```

Los .scss salieron con la versión 3 de preprocesador y permiten utilizar llaves e incorporar código de CSS clásico.

```
body {
    background: #000;
    font-size: 62.5%;
}
```

Tanto la sintaxis de .sass como la de .scss no puede ser interpretada directamente por los navegadores, por lo se debe compilar para traducir nuestro archivo SASS en un clásico fichero CSS.

El uso SASS o cualquier otro pre-procesador no es obligatorio, pero como hemos adelantado antes, nos provee de diferentes funcionalidades no incluidas en CSS puro y que harán más sencillo el diseño de nuestro sitio web, así como la reutilización de código en diferentes proyectos:

- Variables
- Funciones matemáticas
- Bucles
- Condicionales
- Estructuras lógicas y de control
- Mixins o plantillas
- Indentaciones
- Anidamientos
- Herencias
- Ficheros parciales
- Etc.

El uso de SASS requiere de un cambio en la forma de escribir nuestras hojas de estilo, ya que SASS es un lenguaje que trabaja mediante indentaciones y retornos de carro, en vez de utilizar llaves y puntos y comas.

Existe otra forma de utilizar SASS más orientada a la compatibilidad con el estándar CSS llamada SCSS, que evita dichas indentaciones y retornos de carro y las sustituye por el estándar CSS.

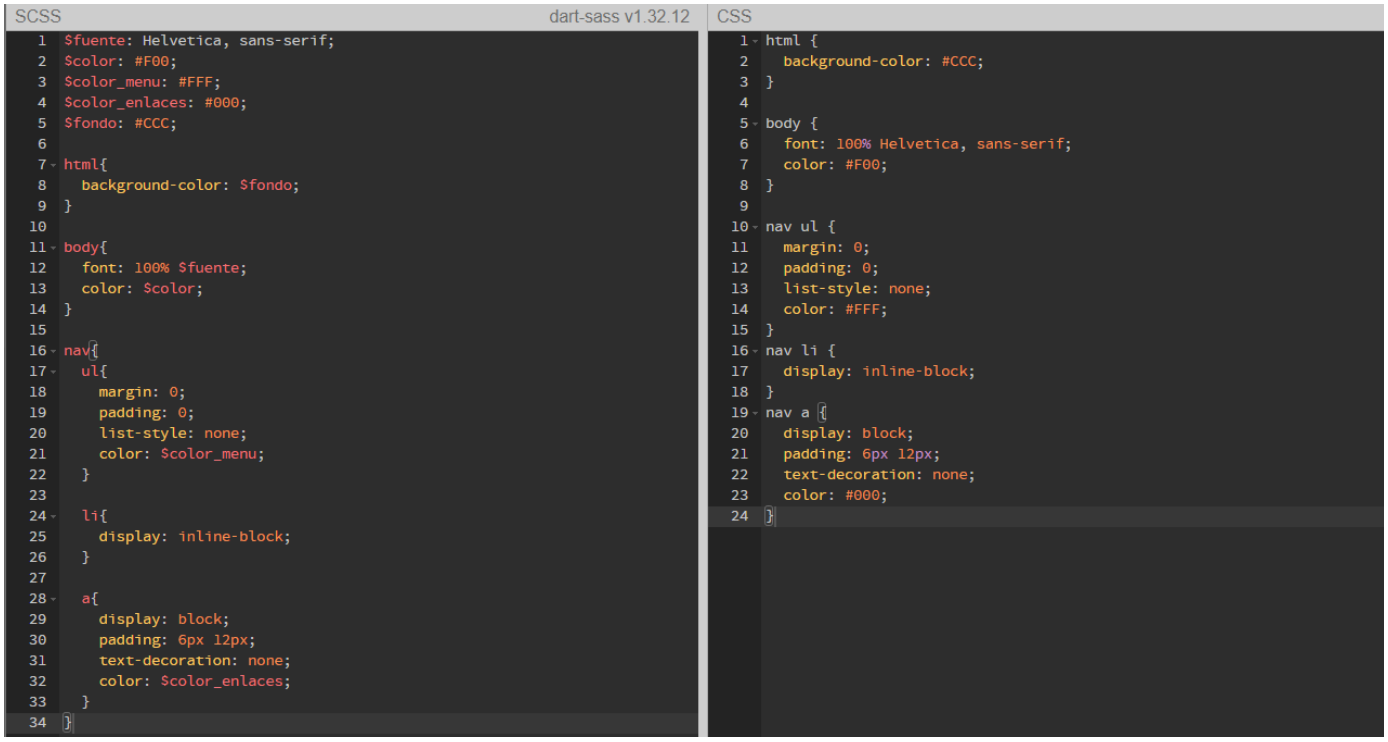
Podemos utilizar cualquiera de las dos, pero no podemos mezclarlas en el mismo proyecto, ya que los ficheros tendrán extensión *.sass o *.scss.

Ejemplo de sintaxis en SASS

```
$fuente: Helvetica, sans-serif
$color: #F00
$color_menu: #FFF
$color_enlaces: #000
$fondo: #CCC

html
    background-color: $fondo


body
    font: 100% $fuente
    color: $color

nav
    ul
        margin: 0
        padding: 0
        list-style: none
```

```
        color: $color_menu

    li
        display: inline-block

    a
        display: block
        padding: 6px 12px
        text-decoration: none
        color: $color_enlaces
```

Sintaxis en SCSS y el equivalente a CSS



SCSS (dart-sass v1.32.12)

```scss
$fuente: Helvetica, sans-serif;
$color: #F00;
$color_menu: #FFF;
$color_enlaces: #000;
$fondo: #CCC;

html{
  background-color: $fondo;
}

body{
  font: 100% $fuente;
  color: $color;
}

nav{
  ul{
    margin: 0;
    padding: 0;
    list-style: none;
    color: $color_menu;
  }

  li{
    display: inline-block;
  }

  a{
    display: block;
    padding: 6px 12px;
    text-decoration: none;
    color: $color_enlaces;
  }
}
```

CSS

```css
html {
  background-color: #CCC;
}

body {
  font: 100% Helvetica, sans-serif;
  color: #F00;
}

nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
  color: #FFF;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
  color: #000;
}
```

Instalación

Aunque existen varias maneras de llevar a cabo la instalación, nos vamos a centrar en dos formas sencillas que nos pueden ayudar.

Mediante el uso de un plug-in “en vivo” para Visual Studio Code.

En el caso concreto de Visual Studio Code, es posible utilizar un plug-in gratuito que, sin mayor configuración, permite trabajar con archivos *.sass y *.scss y procesarlos en tiempo real.

Para conseguirlo, debemos acceder a la página del plug-in Live Sass Compiler y pulsar el botón Install, o bien acceder en Visual Studio Code a la gestión de extensiones (Ctrl + P) e introducir ext install live-sass:

Configuración en Visual Studio Code.

En este caso, una vez instalado el plug-in, tan solo debemos abrir Visual Studio Code y, desde la barra de estado (View > Appearance > Show Status Bar) y, con un archivo con extensión *.sass o *.scss, hacer clic en el botón Watch Sass, de modo que empiece a procesar los cambios en nuestro archivo y volcarlos en otro de css puro que tendrá el mismo nombre que el original, pero con extensión *.css:

Para dejar de procesar, tan solo debemos volver a pulsar el botón.

Otra opción para conseguir el mismo resultado, es abrir una Command Palette (View > Command Palette... o Ctrl + Shft + P) y teclear Live Sass: Watch Sass para empezar a procesar en tiempo real, o Live Sass: Stop Watching Sass para dejar de hacerlo.

Por último, también es posible teclear Live Sass: Compile Sass - Without Watch Mode, de modo que solo haga la compilación en ese momento (primero se debe guardar el archivo).

Mediante el uso de frameworks de desarrollo.

Algunos frameworks de desarrollo, incorporan funcionalidades para optimizar el desarrollo con SASS, como es el caso de Webpack en el caso de Laravel.

Sintaxis y Reglas

En este apartado introduciremos los siguientes conceptos:

- Anidamiento (Nesting) de estilos y propiedades
- Referencia a selectores padre
- Comentarios
- Selectores placeholder
- Funciones especiales

Anidamiento (Nesting) de estilos y propiedades

Ya sabemos que las reglas de estilo son la base de CSS, y lo mismo ocurre con Sass y SCSS, de modo que la declaración siguiente es válida tanto en CSS como en SCSS:

```
.button {
    padding: 3px 10px;
    font-size: 12px;
    border-radius: 3px;
    border: 1px solid #e1e4e8;
  }
```

Sin embargo, con Sass/SCSS podemos mejorar mucho nuestras hojas de estilo si introducimos el concepto de anidamiento o nesting, que básicamente nos va a permitir anidar los estilos que correspondan a elementos hijos dentro de los estilos de sus padres:

```
nav {
    ul {
      margin: 0;
      padding: 0;
      list-style: none;
    }

    li { display: inline-block; }

    a {
      display: block;
      padding: 6px 12px;
      text-decoration: none;
    }
```

```
}
```

Correspondería al CSS:

```
nav ul {
    margin: 0;
    padding: 0;
    list-style: none;
}
nav li {
    display: inline-block;
}
nav a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
}
```

No obstante debemos tener en cuenta que el anidamiento excesivo de estilos se convertirá en hojas de estilo cada vez más complejas y poco manejables para el navegador, por lo que debemos realizar el anidamiento con precaución.

También es posible utilizar el anidamiento para las listas de selectores, esto es, selectores separados por comas, ya que cada uno de estos selectores será anidado de manera separada y luego combinado de nuevo en la lista de selectores resultante:

SCSS:

```
.alert, .warning {
    ul, p {
      margin-right: 0;
      margin-left: 0;
      padding-bottom: 0;
    }
}
```

CSS:

```
.alert ul, .alert p, .warning ul, .warning p {
    margin-right: 0;
    margin-left: 0;
    padding-bottom: 0;
}
```

Referencia a selectores padre

Otra utilidad interesante para combinar con el anidamiento es el uso del selector de padre o parent selector, &, utilizado para referenciar al selector exterior del anidamiento:

SCSS:

```scss
.alert {
   /* Para añadir pseudo-clases al selector exterior */
   &:hover {
     font-weight: bold;
   }
      /* Para añadir un estilo al selector exterior en un determinado contexto (por ejemplo, un body con idioma de derecha a izquierda) */
   [dir=rtl] & {
     margin-left: 0;
     margin-right: 10px;
   }
    /* Como argumento para selectores de pseudo-clases */
   :not(&) {
     opacity: 0.8;
   }
    /* Con sufijos */
   &__copy {
     display: none;
   }
 }
```

CSS:

```css
.alert:hover {
 font-weight: bold;
}
[dir=rtl] .alert {
 margin-left: 0;
 margin-right: 10px;
}
:not(.alert) {
 opacity: 0.8;
}
.alert__copy {
 display: none;
}
```

Comentarios

Los comentarios en SCSS funcionan de forma similar a los de JavaScript, esto es, los comentarios de una línea empiezan con //, van hasta el final de la línea y no son procesados en el CSS de salida (son llamados silent comments por este motivo).

Los comentarios multi-línea empiezan con /* y terminan con */. Por defecto, el comentario aparecerá en CSS, salvo en el caso de que éste sea comprimido. Por eso se llaman loud comments.

Si el comentario empieza con /*! aparecerá incluso si está comprimido.

Además, si el comentario está en un lugar donde podría ir una sentencia (expresión a evaluar) o contiene interpolación (resultado de una expresión SassScript), siempre aparecerá:

SCSS:

```
// Este comentario no aparecerá en CSS.

/* Este comentario sí, salvo que el CSS esté comprimido. */

/* Puede contener interpolación:
* 1 + 1 = #{1 + 1} */

/*! Este comentario siempre aparecerá. */

p /* Los comentarios en línea pueden ir en cualquier lugar
* donde esté permitido un espacio en blanco. */ .sans {
font: Helvetica, // Al igual que los comentarios de una línea.
    sans-serif;
}
```

CSS:

```
/* Este comentario sí, salvo que el CSS esté comprimido. */
/* Puede contener interpolación:
* 1 + 1 = 2 */
/*! Este comentario siempre aparecerá. */
p .sans {
 font: Helvetica, sans-serif;
}
```

Selectores placeholder

Existe un tipo de selector específico en Sass y SCSS llamado placeholder, que parece y actúa como un selector de clase estándar pero empieza con % y no se incluirá luego en el CSS de salida:

SCSS:

```
.alert:hover, %strong-alert {
   font-weight: bold;
 }
  %strong-alert:hover {
   color: red;
 }
```

CSS:

```
.alert:hover {
 font-weight: bold;
}
```

La pregunta es obvia: **¿Para qué queremos un selector que luego no aparece en el CSS de salida?**

Básicamente porque aún no lo podemos extender. Todavía no hemos hablado de la directiva @extend, pero básicamente nos permitirá extender cualquier estilo (heredar de él, vaya).

De este modo, estos placeholder selectors nos permitirían crear una plantilla de estilos que podríamos extender en un momento dado (como una librería en un archivo aparte) y sólo aparecerán en el CSS de salida si los hemos utilizado:

SCSS:

```
%toolbelt {
   box-sizing: border-box;
   border-top: 1px rgba(#000, .12) solid;
   padding: 16px 0;
   width: 100%;
    &:hover { border: 2px rgba(#000, .5) solid; }
 }
  .action-buttons {
   @extend %toolbelt;
   color: #4285f4;
 }
  .reset-buttons {
   @extend %toolbelt;
   color: #cddc39;
 }
```

CSS:

```
.reset-buttons, .action-buttons {
 box-sizing: border-box;
 border-top: 1px rgba(0, 0, 0, 0.12) solid;
 padding: 16px 0;
 width: 100%;
}
.reset-buttons:hover, .action-buttons:hover {
 border: 2px rgba(0, 0, 0, 0.5) solid;
}

.action-buttons {
 color: #4285f4;
}

.reset-buttons {
 color: #cddc39;
}
```

Variables

El uso de variables en Sass/SCSS es bastante simple: asignamos un valor a un nombre que empiece por $ (como en PHP) y luego podemos referirnos a él en cualquier lugar en vez de referirnos al valor.

Además de la evidentemente reducción de repeticiones y facilidad de refactorización, las variables nos van a permitir realizar operaciones matemáticas, configurar librerías y mucho más.

Las declaraciones de variable son como las declaraciones de propiedades, esto es, :, pero al contrario que las propiedades, las variables se pueden definir en cualquier lugar, sea dentro o fuera de las reglas.

Es importante saber que los nombres de variables son tratados como el resto de identificadores Sass, de modo que los guiones bajos y medios son tratados como iguales, y por tanto $font_size y $font-size harán referencia a la misma variable.

SCSS:

```
$base-color: #c6538c;
$border-dark: rgba($base-color, 0.88);

.alert {
 border: 1px solid $border-dark;
}
```

CSS:

```
.alert {
 border: 1px solid rgba(198, 83, 140, 0.88);
}
```

Importante: No hay que confundir las variables de Sass/SCSS con las variables de CSS. Éstas últimas pertenecen a CSS puro pero, como es sabido, tienen muchos problemas de compatibilidad entre navegadores, mientras que las variables de Sass/SCSS se trasladan a CSS puro mediante compilación, pero no aparecen en el archivo de salida, con lo que podemos utilizarlas sin problemas de compatibilidad.

Operadores

Como en la mayoría de los lenguajes, podemos utilizar diferentes tipos de operadores, tanto matemáticos como relacionales o de comparación, el funcionamiento es bastante similar:

- **==** y **!=** se usan para comprobar si dos valores son iguales o no
- **+**, **-**, *****, **/** y **%** efectúan la correspondiente operación matemática entre los valores
- **<**, **<=**, **>** y **>=** son los operadores de comparación
- **and**, **or** y **not** tienen el comportamiento usual de los operadores booleanos
- **+**, **-** y **/** también se pueden usar para encadenar strings, si bien **-** y **/** funcionan como operadores unarios y están en desuso.

Por último, es importante saber cuál es el orden de mayor a menor prioridad en el que se efectúan las operaciones (y aplicar paréntesis de forma matemática en nuestras expresiones si fuera necesario):

1. Los operadores unarios not, **+**, **-** y **/**

2. Los operadores matemáticos *****, **/** y **%**
3. Los operadores matemáticos **+** y **-**
4. Los operadores de comparación **<**, **<=**, **>** y **>=**
5. Los operadores de igualdad **==** y **!=**
6. El operador booleano **and**
7. El operador booleano **or**
8. El operador **=** (en desuso, antes se usaba en funciones para mantener la compatibilidad con Internet Explorer)

Tipos de datos

Sass/SCSS permiten trabajar con muchos tipos de datos, muchos de los cuales provienen directamente de CSS:

- **Números**, que pueden tener unidades o no, como 12 o 200px.
- **Strings**, que pueden tener comillas o no, como "Helvetica Neue" o bold.
- **Colores**, que pueden indicarse en hexadecimal (#F00), por el nombre (red), o como funciones (rgb(255,0,0) o hsl(0, 100%, 50%))
- **Listas de valores**, que pueden estar separadas por espacios (1.5em 1em 0 2em), comas (Helvetica, Arial) o corchetes ([col1-start]).

y otros son más específicos de Sass/SCSS, siendo los más comunes:

- **Mapas** que permiten asociar valores con claves, como ("background": red, "foreground": pink).
- **Booleanos**, cuyos posibles valores serán true o false.
- **Null** o ausencia de valor, muy útil para el retorno de funciones.

Mapas

Los mapas contienen pares de índices y valores, y hacen más sencillo asignar un valor a partir de su correspondiente índice.

Es posible añadir elementos al mapa mediante la función map-merge(), pero hay que tener en cuenta que los mapas son inmutables, por lo que en ningún caso estaremos modificando la lista original, a no ser que asignemos el nuevo mapa a otra variable.

```
$font-weights: ("regular": 400, "medium": 500, "bold": 700);

@debug map-get($font-weights, "medium"); // 500
@debug map-get($font-weights, "extra-bold"); // null
```

```
$light-weights: ("lightest": 100, "light": 300);
$heavy-weights: ("medium": 500, "bold": 700);

@debug map-merge($light-weights, $heavy-weights);
// (
//   "lightest": 100,
//   "light": 300,
//   "medium": 500,
//   "bold": 700
// )

$font-weights: ("regular": 400, "medium": 500, "bold": 700);

@debug map-merge($font-weights, ("extra-bold": 900));
// ("regular": 400, "medium": 500, "bold": 700, "extra-bold": 900)

$font-weights: ("regular": 400, "medium": 500, "bold": 700);

@debug map-merge($font-weights, ("medium": 600));
// ("regular": 400, "medium": 600, "bold": 700)

$prefixes-by-browser: ("firefox": moz, "safari": webkit, "ie": ms);

@mixin add-browser-prefix($browser, $prefix) {
 $prefixes: map-merge($prefixes-by-browser, ($browser: $prefix));
}

@include add-browser-prefix("opera", o);
@debug $prefixes-by-browser;
```

Además, podemos trabajar con el mapa de modo que hagamos algo para cada par valor e índice:

SCSS:

```
@each $name, $glyph in $icons {
 .icon-#{$name}:before {
   display: inline-block;
   font-family: "Icon Font";
   content: $glyph;
 }
}
```

CSS:

```
.icon-eye:before {
 display: inline-block;
 font-family: "Icon Font";
 content: "\f112";
}

.icon-start:before {
 display: inline-block;
 font-family: "Icon Font";
 content: "\f12e";
}

.icon-stop:before {
 display: inline-block;
 font-family: "Icon Font";
```

```
 content: "\f12f";
}
```

Importante: Al igual que las listas, los mapas de Sass/SCSS son inmutables, esto es, todas las funciones devolverán un nuevo mapa sin afectar al original. Para modificar el mapa original, debemos asignar a esa misma variable el valor modificado.

Funciones

Además de la posibilidad de definir nuestras propias funciones, que veremos más adelante, Sass/SCSS viene con gran cantidad de funciones predefinidas a las que podremos llamar utilizando la sintaxis estándar de CSS y, en algunos casos, una sintaxis especial de Sass.

Estas funciones predefinidas están divididas en categorías que se corresponden en la mayoría de los casos con los tipos de variables definidos anteriormente:

- Funciones de CSS, aquéllas que no son reconocidas por Sass como propias y que se compilarán como funciones de CSS estándar (var(), calc(), url(), etc.)
- Funciones de Números, normalmente de carácter matemático
- Funciones de Strings, para crearlos, combinarlos, dividirlos y hacer búsquedas
- Funciones de Colores, para generarlos, mezclarlos o modificarlos
- Funciones de Listas, para acceder a ellas o modificarlas
- Funciones de Mapas, para trabajar con ellos
- Funciones de Selectores, para acceder al motor de selectores de Sass
- Funciones de Introspección, para mostrar detalles de la forma interna de trabajar de Sass

Para conocer todas las funciones incluidas puedes consultar la documentación oficial.

Interpolado

La interpolación es algo que podemos utilizar casi en cualquier lugar de una hoja de Sass/SCSS para incrustar el resultado de una expresión SassScript en CSS, si bien su uso más común es inyectar valores en Strings. Para ello, debemos encerrar dicha expresión mediante #{ } en cualquiera de los siguientes lugares:

- Selectores en reglas de estilo
- Nombres de propiedades en declaraciones
- Valores de propiedades personalizadas
- Directivas (@)

- Herencias (@extends)
- @imports de CSS
- Strings con o sin comillas
- Funciones especiales
- Funciones de CSS
- Comentarios de tipo loud

Un ejemplo de uso en selectores:

SCSS:

```scss
@mixin corner-icon($name, $top-or-bottom, $left-or-right) {
   .icon-#{$name} {
     background-image: url("/icons/#{$name}.svg");
     position: absolute;
     #{$top-or-bottom}: 0;
     #{$left-or-right}: 0;
   }
 }
  @include corner-icon("mail", top, left);
```

CSS:

```css
.icon-mail {
 background-image: url("/icons/mail.svg");
 position: absolute;
 top: 0;
 left: 0;
}
```

Reglas y directivas básicas

En este apartado introduciremos las reglas y directivas básicas, también conocidas como at-rules:

- Importación (@import) y Parciales (Partials)
- Mixins (@mixin) e Includes (@include)
- Funciones (@function)
- Herencia y extensión (@extend)

Importación (@import) y Parciales (Partials)

Sass/SCSS extiende la regla @import de CSS añadiendo la capacidad de importar tanto hojas de estilo CSS como hojas de Sass/CSS, permitiendo así el acceso a mixins, funciones y variables de dichas hojas y la combinación de todas ellas en una sola.

Además, Sass/SCSS realiza toda la importación durante la compilación, en vez de obligar al navegador a hacer una petición HTTP por cada @import como hace CSS.

La sintaxis es similar a la de CSS, salvo que en este caso podremos realizar varios @import a la vez, separando las rutas de los archivos por comas:

```
// archivo foundation/_code.scss

code {
   padding: .25em;
   line-height: 0;
 }
// archivo foundation/_lists.scss
 ul, ol {
   text-align: left;
    & & {
     padding: {
       bottom: 0;
       left: 0;
     }
   }
 }
  // archivo style.scss
 @import 'foundation/code', 'foundation/lists';
```

Nota: Las rutas de los archivos se escriben en formato url, usando / incluso en Windows. Por otra parte, no es necesario el uso de ./ para @import relativos, pues estos siempre están disponibles.

Como convención, los ficheros Sass/SCSS pensados para ser importados y no compilados directamente suelen empezar su nombre por _, como el anterior _code.scss. Este tipo de archivos reciben el nombre de **partials** y ese _ le dice a las herramientas de Sass que no lo compilen por sí mismo. No obstante, como hemos visto en el ejemplo anterior, no es necesario poner el _ para importar el partial.

Si creamos un archivo _index.scss o _index.sass en un directorio, cuando importamos dicho directorio dicho fichero será cargado en su lugar, lo cual puede resultar util para mejorar la estructura de nuestras hojas y partials.

Si bien los **@import** están pensados para ser utilizados al principio de la hoja, es posible anidarlos dentro de reglas de estilo u otras directivas.

Mixins (@mixin) e Includes (@include)

Los mixins nos permiten definir estilos para ser re-utilizables en todas las hojas de estilo o ser distribuidos en forma de librerías.

La sintaxis para declararlos es de tipo **@mixin <nombre> { ... }** o de tipo **@mixin nombre(<argumentos...>) { ... }**, mientras que la sintaxis para llamarlos será utilizando **@include <nombre>** o bien **@include <nombre>(<argumentos...>)**

SCSS:

```
@mixin reset-list {
   margin: 0;
   padding: 0;
   list-style: none;
 }
@mixin horizontal-list {
   @include reset-list;
    li {
     display: inline-block;
     margin: {
       left: -2px;
       right: 2em;
     }
   }
 }
  nav ul {
   @include horizontal-list;
 }
```

CSS:

```
nav ul {
 margin: 0;
 padding: 0;
 list-style: none;
}
nav ul li {
 display: inline-block;
 margin-left: -2px;
 margin-right: 2em;
}
```

Funciones (@function)

Las funciones nos van a permitir definir operaciones complejas que podremos abstraer de una manera sencilla y utilizar en cualquier otra parte de la hoja.

Se definen mediante la directiva **@function <nombre>(<argumentos...>) { ... }** y solo puede contener sentencias universales (variables, directivas de control o directivas @error, @warn y @debug).

Los argumentos funcionan de manera similar a los mixins, tal y como muestran los siguientes ejemplos:

```
 //Argumentos obligatorios
@function pow($base, $exponent) {
   $result: 1;
   @for $_ from 1 through $exponent {
     $result: $result * $base;
   }
```

```
  @return $result;
}
 .sidebar {
  float: left;
  margin-left: pow(4, 3) * 1px;
}
 //Argumentos opcionales
@function invert($color, $amount: 100%) {
  $inverse: change-color($color, $hue: hue($color) + 180);
  @return mix($inverse, $color, $amount);
}
 $primary-color: #036;
.header {
  background-color: invert($primary-color, 80%);
}
 //Argumentos con keywords
$primary-color: #036;
.banner {
  background-color: $primary-color;
  color: scale-color($primary-color, $lightness: +40%);
}
 //Argumentos como listas
@function min($numbers...) {
  $min: null;
  @each $number in $numbers {
    @if $min == null or $number < $min {
      $min: $number;
    }
  }
  @return $min;
}
 .micro {
  width: min(50px, 30px, 100px);
}
```

CSS:

```
.sidebar {
 float: left;
 margin-left: 64px;
}

.header {
 background-color: #523314;
}

.banner {
 background-color: #036;
 color: #0a85ff;
}

.micro {
 width: 30px;
}
```

Importante: La directiva @return que hemos visto en algunos de los ejemplos anteriores indica, como en otros lenguajes, que le ejecución finaliza y se devuelve el resultado. Esta directiva es obligatoria y solo se puede utilizar dentro de @function.

Herencia y extensión (@extend)

Hay muchos casos en los que nos encontramos en la necesidad de que una clase tenga todos los estilos de otra, además de algo específico para ella. Para estos casos, utilizamos la directiva **@extend** mediante la sintaxis **@extend <selector>**.

Esto nos permitirá modificar reglas de una forma más sencilla y óptima que con un **mixin**:

```
//Ejemplo 1
.error {
 border: 1px #f00;
 background-color: #fdd;

 &--serious {
   @extend .error;
   border-width: 3px;
 }
}

//Ejemplo 2
.error:hover {
 background-color: #fee;
}

.error--serious {
 @extend .error;
 border-width: 3px;
}
```

CSS:

```
.error, .error--serious {
 border: 1px #f00;
 background-color: #fdd;
}
.error--serious {
 border-width: 3px;
}

.error:hover, .error--serious:hover {
 background-color: #fee;
}

.error--serious {
 border-width: 3px;
}
```

Directivas de control y heredadas

En este apartado introduciremos las directivas heredadas de CSS puro y las estructuras de control, básicas para gestionar el comportamiento de las reglas en distintos ámbitos y también incluidas dentro de las at-rules :

- Condicionales (@if y @else)
- Bucles (@each, @for y @while)
- Directivas heredadas (@namespace, @font-face, @media, @support)

Condicionales (@if y @else)

La directiva **@if** permite comprobar si un bloque es evaluado o no, y como en cualquier otro lenguaje, opcionalmente puede incorporar la directiva **@else** o la directiva **@elseif.**

La sintaxis es del tipo **@if <expression> { ... }.**

```
//Ejemplo @if
@mixin avatar($size, $circle: false) {
   width: $size;
   height: $size;
    @if $circle {
     border-radius: $size / 2;
   }
 }
 .square-av { @include avatar(100px, $circle: false); }
 .circle-av { @include avatar(100px, $circle: true); }

  //Ejemplo @else
 $light-background: #f2ece4;
 $light-text: #036;
 $dark-background: #6b717f;
 $dark-text: #d2e1dd;
 @mixin theme-colors($light-theme: true) {
   @if $light-theme {
     background-color: $light-background;
     color: $light-text;
   } @else {
     background-color: $dark-background;
     color: $dark-text;
   }
 }
 .banner {
   @include theme-colors($light-theme: true);
   body.dark & {
     @include theme-colors($light-theme: false);
   }
 }
```

```
 //Ejemplo @elseif
@mixin triangle($size, $color, $direction) {
  height: 0;
  width: 0;
   border-color: transparent;
  border-style: solid;
  border-width: $size / 2;
   @if $direction == up {
    border-bottom-color: $color;
  } @else if $direction == right {
    border-left-color: $color;
  } @else if $direction == down {
    border-top-color: $color;
  } @else if $direction == left {
    border-right-color: $color;
  } @else {
    @error "Unknown direction #{$direction}.";
  }
}
  .next {
   @include triangle(5px, black, right);
}
```

Bucles (@each, @for y @while)

La directiva **@each** permite recorrer una lista y emitir estilos o evaluar código para cada uno de sus elementos. La sintaxis será de tipo **@each <variable> in <expresión> { ... }**.

```
$sizes: 40px, 50px, 80px;

@each $size in $sizes {
 .icon-#{$size} {
   font-size: $size;
   height: $size;
   width: $size;
 }
}
```

CSS:

```
.icon-40px {
   font-size: 40px;
   height: 40px;
   width: 40px;
 }

  .icon-50px {
   font-size: 50px;
   height: 50px;
   width: 50px;
 }

  .icon-80px {
   font-size: 80px;
   height: 80px;
```

```
   width: 80px;
 }
```

La directiva **@for** permite contar de un número superior a otro inferior (o viceversa) y realizar una acción en cada pasada. La sintaxis será de tipo **@for <variable> from <expresión> to <expresión> { ... }** si queremos excluir el valor final, o bien **@for <variable> from <expresión> through <expresión> { ... }** si queremos incluirlo.

SCSS:

```
$base-color: #036;
@for $i from 1 through 3 {
 ul:nth-child(3n + #{$i}) {
   background-color: lighten($base-color, $i * 5%);
 }
}
```

CSS:

```
ul:nth-child(3n+1) {
 background-color: #004080;
}

ul:nth-child(3n+2) {
 background-color: #004d99;
}

ul:nth-child(3n+3) {
 background-color: #0059b3;
}
```

La directiva **@while** evalúa una expresión y ejecuta acciones mientras sea cierta. La sintaxis será de tipo **@while <expresión> { ... }**.

SCSS:

```
/// Dividir `$value` entre `$ratio` hasta que sea menor que `$base`.
@function scale-below($value, $base, $ratio: 1.618) {
 @while $value > $base {
   $value: $value / $ratio;
 }
 @return $value;
}

$normal-font-size: 16px;
sup {
 font-size: scale-below(20px, 16px);
}
```

CSS:

```
sup {
 font-size: 12.3609394314px;
}
```

Nota: Como en cualquier otro lenguaje, para evitar bucles infinitos y porque además son más rápidas, es mejor utilizar **@for o @each** frente a **@while.**

```
//Ejemplo @namespace
@namespace svg url(http://www.w3.org/2000/svg);

//Ejemplo @font-face
@font-face {
 font-family: "Open Sans";
 src: url("/fonts/OpenSans-Regular-webfont.woff2") format("woff2");
}

//Ejemplo @media sin argumentos
.print-only {
 display: none;

 @media print { display: block; }
}

//Ejemplo @media con argumentos
$layout-breakpoint-small: 960px;

@media (min-width: $layout-breakpoint-small) {
 .hide-extra-small {
   display: none;
 }
}

//Ejemplo @support (para compatibilidad de navegadores)
@mixin sticky-position {
 position: fixed;
 @supports (position: sticky) {
   position: sticky;
 }
}

.banner {
 @include sticky-position;
}
```

CSS:

```
@namespace svg url(http://www.w3.org/2000/svg);
@font-face {
 font-family: "Open Sans";
 src: url("/fonts/OpenSans-Regular-webfont.woff2") format("woff2");
}
.print-only {
 display: none;
}
@media print {
 .print-only {
   display: block;
 }
}

@media (min-width: 960px) {
 .hide-extra-small {
   display: none;
 }
```

```
}
.banner {
 position: fixed;
}
@supports ((position: -webkit-sticky) or (position: sticky)) {
 .banner {
   position: -webkit-sticky;
   position: sticky;
 }
}
```

Bibliografía

https://jairogarciarincon.com/clase/sass-y-scss

https://uniwebsidad.com/libros/sass

https://www.ondho.com/que-es-sass-y-por-que-los-css-pueden-volver-a-divertirnos/