SASS

Índice:

1. Sintaxis

- Sintaxis SCSS: Es una sintaxis bastante similar a CSS, de hecho el código CSS es código SCSS válido. Se podría decir que SCSS es codigo CSS con algunas cosas extras como: variables, nesting, mixins, y herencia de los selectores.

```
$button-height: 1.5em;

@mixin button-base() {
  @include typography(button);

  display: inline-flex;
  position: relative;
  height: $button-height;
  border: none;

  &:hover { cursor: pointer; }
}
```

- Sintaxis SASS: Esta sintaxis es un poco diferente al CSS estándar. Evita poner **;** al final de los valores de las propiedades, evita usar **{ }** para definir las reglas, tan solo usa indentaciones *(como python)*.

```
@mixin button-base()
  @include typography(button)

  display: inline-flex
  position: relative
  height: $button-height
  border: none

  &:hover
    cursor: pointer
```

.

2. Variables

Recientemente se han agregado a CSS variables, aunque no todos los navegadores son compatibles, todavía, con esta nueva función. Por este motivo es muy util seguirse apoyando en los preprocesadores. Las variables son capaces de almacenar un dato que luego podremos usar todas las veces que nos haga falta a lo largo de nuestro código *Sass*.

Así se definen variables en Sass

```
$primary-color: #55A;

$normal-space: 10px;

$large-text-size: 2em;

$general-typography: arial, verdana, sans-serif;
$alternative-typography: 'Times New Roman', Times, serif;
```

Así se usan las variables en Sass

```
.title {
  color: $primary-color;
  font-family: $alternative-typography;
  font-size: $large-text-size;
}
```

3. Anidación *(nesting)*

El código Sass soporta una organización en forma de árbol, como el *HTML*. A la hora de producir código anidado en Sass hay una técnica que se usa bastante y es referenciar al selector padre usando el caracter *"&"* con el fin de no repetir código innecesario y aligerar nuestra maquetación.

Ahora vamos a ver un ejemplo práctico:

```
<div class="container">
  <header>Este es el encabezado
    <a href="#" class="container__link">Enlace encabezado</a>
  </header>
</div>
```

En css quedaría de la siguiente manera:

```
.container {
  background-color: red;
}

.container header {
  background-color: black;
  color: #fff;
}

.container header a {
  color: #ff6;
}
.container header a:hover {
  color: #ff6;
}
```

En SCSS quedaría así:

```
.container {
  background-color: red;

  header {
    background-color: black;
    color: #fff;
  }

  &__link {    //referencia a .container
    color: #ff6;

    &:hover {   //referencia a .container__link
      color: #6ff
    }
  }
}
```

4. Modularización

Dividir el código CSS de tu aplicación en múltiples ficheros, facilitará enormemente el mantenimiento. Vamos a llamar *"módulos"* a los archivos que creas para importar desde otros archivos scss. Para que el compilador de Sass no compile un módulo por separado, tiene que saber qué archivo es un módulo. Para ello simplemente vamos a nombrar el módulo comenzando por un guión bajo "*_*". *Imagina que quieres crear un módulo con una serie de definiciones de variables. Entonces ese módulo lo podrías llamar "_variables.scss". Todos los archivos que comiencen por un "_"* no generarán nuevos archivos .css,

Cómo importar un módulo Sass:

En el archivo scss principal, iremos realizando todas las importaciones de los módulos, para agrupar todos los archivos que teníamos por separado. Esto se consigue con la sentencia **@import**, a continuación del

nombre del módulo a importar, **sin el guión bajo y sin la extensión.** Este import debe figurar en el código scss antes que comiences a usar las variables que se están declarando.

Ejemplo del archivo "_variables.scss":

```
@import 'variables';

body {
  font-size: $f-normal;
  font-family: $tipo-general;
}
```

Cómo organizar tus archivos de CSS:

Existen diversas aproximaciones a arquitecturas CSS publicadas por desarrolladores de renombre, que ofrecen guías "estandarizadas" para organizar el código CSS. Una bastante conocida es ITCSS, que puede servirte de ayuda para llegar a tu propio modelo de organización del código.

Será aconsejable tener definiciones en archivos por separado de cosas como:

- Colores
- Fuentes
- Espaciados
- Clases
- Componentes

En otro PDF hablaré de manera más extendida de la arquitectura ITCSS.

5. Mixins

Serían algo así como lo que conocemos como funciones en un lenguaje de programación.

Podemos escribir un mixin e invocarlo tantas veces como sea necesario, produciendo una salida CSS. Un mixin además tiene la habilidad de recibir parámetros, como las funciones, de modo que se produzca la salida a partir del valor de los parámetros recibidos en su invocación.

Vamos a ver un ejemplo para invertir los colores:

> Esto facilita el mantenimiento porque, si más adelante queremos modificar el ese "color invertido", solo habría que cambiarlo en el mixin, una vez, y se traspasará a todos los lugares donde lo estamos usando.

Definir un mixin:

Tenemos que comenzar a declararlos con la cadena "@mixin", seguido por el nombre del mixin que estamos generando y unas llaves para englobar su código.

```
@mixin color-invertido {
  background-color: #111;
  color: #eee;
}
```

Invocar un mixin:

Es tan sencillo como escribir **"@include"** seguido del nombre del mixin que queremos invocar.

```
blockquote {
  padding: 20px;
  text-align: center;
  @include color-invertido;
}
```

Paso de parámetros en un mixin Sass:

Nuestros mixins pueden producir salida diferente, simplemente mandando valores de parámetros distintos.

Un claro ejemplo de esta utilidad es la inserción automática de los vendor prefixes de CSS, es decir, los prefijos que necesitan algunos navegadores para entender ciertas propiedades CSS que todavía no están en el estándar definitivo. Una de ellas es "transform".

```
@mixin transformar($propiedad) {
  -webkit-transform: $propiedad;
  -ms-transform: $propiedad;
  transform: $propiedad;
}
```

Ahora podemos aplicar ese mixin en cualquier elemento que queramos transformar. Seguiremos usando el método de antes con @include y después del nombre del mixin debemos enviar el valor del parámetro:

```
.escalada {
  @include transformar(scale(2, 3))
}

h1 {
  @include transformar(rotate(22deg))
}
```

Esto producirá como salida el siguiente CSS procesado.

```
.escalada {
  -webkit-transform: scale(2, 3);
  -ms-transform: scale(2, 3);
  transform: scale(2, 3);
}

h1 {
  -webkit-transform: rotate(22deg);
  -ms-transform: rotate(22deg);
  transform: rotate(22deg);
}
```

También puede ser util para la generación de las @mediaqueries.

```
@mixin encabezados($tamano) {
  h1 {
    font-size: $tamano;
  }
  h2 {
    font-size: $tamano - 0.2;
  }
  h3 {
    font-size: $tamano - 0.5;
  }
}

@media(min-width: 800px) {
  @include encabezados(2em);
}
@media(min-width: 1200px) {
  @include encabezados(2.5em);
}
```

6. Herencia

La herencia es un mecanismo por el cual un selector puede recibir estilos CSS que nos llegan de declaraciones realizadas con anterioridad. Conseguir este objetivo es sencillo gracias a la directiva @extend y las denominadas "placeholder class", que son una construcción de Sass que no tiene representación en el CSS hasta que no la usemos. Vamos a ver algunos ejemplos.

Clases placeholder:

Este tipo de clases CSS **son específicas de Sass**. Básicamente son declaraciones CSS que podemos realizar agrupando diversas reglas de estilo. No tienen una representación directa en el código CSS compilado. Sólo se escribirán en el CSS resultante cuando sean usadas.

Creación:

La sintaxis para crearlas consiste en anteponerle un símbolo "%".

```
%heading {
  background-color: blanchedalmond;
  color: brown;
}
```

Esta declaración indica un estilo de base para nuestros encabezados y nos sirve para poder agregarla en donde la necesitemos. Sin embargo, si la dejamos sin usar en el código Sass, no serviría para nada. Al compilarse simplemente se eliminaría sin producir salida alguna.

> Se podría considerar como una función que nunca se ha invocado.

La directiva @extend nos sirve para extender el código de cierta declaración de CSS con nuevos estilos. Las reglas de estilo con las que podremos extender las declaraciones serán tomadas directamente de las clases placeholder.

Por ejemplo, podemos tener varios encabezados en nuestra página que se extienden mediante el placeholder class creado en el punto anterior. Para ello usamos la sintaxis @extend, seguido del nombre de la clase placeholder que queremos usar.

```
h1 {
  @extend %heading;
  font-size: 2em;
}

h2 {
  @extend %heading;
  font-size: 1.5em;
}
```

Es una funcionalidad similar a los mixins, pero realmente los @extend son más sencillos y en la mayoría de las ocasiones es suficiente con ellos.

> Si más adelante se quieren cambiar los estilos es suficiente con editar la placeholder class una vez.

Se compila de la siguiente manera:

```
h1, h2 {
  background-color: blanchedalmond;
  color: brown;
}

h1 {
  font-size: 1.5em;
}

h2 {
```

```
  font-size: 1.5em;
}
```

Por lo que podemos apreciar que sigue unas buenas prácticas en CSS y no repite código de manera innecesaria.

7. Programación (bucles y condicionales):

Bucles:

1. for
2. each
3. while

1. Bucle @for:

@for es el loop estándar en programación. En **SASS**, esta directiva se presenta en dos variantes diferentes: o bien el último ciclo se ejecuta una vez más cuando se alcanza el objetivo o se abandona el bucle antes con la palabra reservada **through.**

Seguidamente se escribe, bien entre corchetes o con sangría, lo que debe suceder. En nuestro ejemplo, cada vuelta hace aumentar la información de tamaño.

```
@for $i from 1 through 4 {  //mientras que $i sea <= que 4
  .width-#{$i} { width: 10em + $i; }
}

@for $i from 1 to 4 { //mientras que $i sea < que 4
  .height-#{$i} { height: 25em * $i; }
}
```

> En SASS #{} es una interpolación. Con ella se puede concatenar a una variable con un identificador que se ha asignado a sí mismo. El equivalente en js sería `${lorem}`

1. Bucle @while:

Tiene una mecánica muy similar a la de @for pero, mientras que este último tiene puntos fijos de inicio y final, un bucle @while contiene una consulta de tipo lógico.

```
$i: 1;
@while $i < 5 {
  .width-#{$i} { width: 10em + $i; }
  $i: $i + 1;
}
```

.

1. Bucle @each:

Funciona de manera diferente. Este bucle se basa en una lista de datos especificada por el usuario que el bucle recorrerá en cada vuelta. Para cada entrada, @each hace una repetición diferente. La verdadera ventaja de este bucle, es que también puede introducirse otra información en la lista además de valores numéricos (con @each, por ejemplo, se insertan varias imágenes diferentes en el diseño). Puedes introducir los datos directamente en la directiva o introducir la lista en una variable y luego llamarla.

```
$list: dog cat bird dolphin;
@each $i in $list {
  .image-#{$i} { background-image: url('/images/#{$i}.png'); }
}
```

Resultado CSS:

```
.image-dog {
background-image: url("/images/dog.png");
}
.image-cat {
background-image: url("/images/cat.png");
}
.image-bird {
background-image: url("/images/bird.png");
}
.image-dolphin {
background-image: url("/images/dolphin.png");
}
```

Condicionales

Con la función if, una orden solo se ejecuta si tiene lugar un cierto evento; en caso contrario, se ejecuta otro comando.

```
$black: #000000;
$white: #ffffff;
$text-color: $black;

body {
    background-color: if($text-color == $black, $white, $black);
}
```

Las funciones tienen la particularidad de devolver un solo valor. Para requisitos más complejos, conviene emplear la directiva @if que permite anidar tantos if/else como se quiera.

.

```
@if ($color == $black) {
background-color: $white;
} @else if ($color == $white) {
background-color: $black;
} @else if ($color == $lightgrey) {
background-color: $black;
} @else {
background-color: $white;
}
```

8. At-Rules:

Gran parte de la funcionalidad extra que tiene SASS en comparación a CSS viene dado por las At-Rules

1. @use
2. @forward
3. @import
4. @mixin and @include
5. @function
6. @extend
7. @error
8. @warn
9. @debug
10. @at-root

1. @use

Esta regla carga mizins, funciones y variables de otras hojas de estilo de SASS. Las hojas de estilo cargadas por @use se denominan "módulos".

> La regla @use debe preceder a cualquier otra regla excepto @forward

Cargando otras hojas de estilo:

En SASS existen los namespaces, se usan de la siguiente manera:

```
// src/_corners.scss
$radius: 3px;

@mixin rounded {
  border-radius: $radius;
}

// style.scss
@use "src/corners";

.button {
  @include corners.rounded;
```

```
  padding: 5px + corners.$radius;
}
```

Si se queda demasiado largo,también se pueden poner namespaces personalizados de la siguiente manera:

```
// src/_corners.scss
$radius: 3px;

@mixin rounded {
  border-radius: $radius;
}

// style.scss
@use "src/corners" as c;

.button {
  @include c.rounded;
  padding: 5px + c.$radius;
}
```

```
//Se renderizarían ambos casos en CSS de la siguiente manera:
.button {
  border-radius: 3px;
  padding: 8px;
}
```

Privacidad:

A veces puede que no quieras que tus paquetes sean usables fuera de la hoja de estilos en la que están definidos. Sass facilita la definición de un miembro privado al comenzar su nombre con _ o -

```
// src/_corners.scss
$-radius: 3px;

@mixin rounded {
  border-radius: $-radius;
}
// style.scss
@use "src/corners";

.button {
  @include corners.rounded;

  // Esto da un error porque la variable $-redaiud no es visible fuera de
'_corners.scss'.
  padding: 5px + corners.$-radius;
}
```

Sobreescribir variables:

Después de cargar un módulo, puede reasignar sus variables.

```
// _library.scss
$color: red;

// _override.scss
@use 'library';
library.$color: blue;

// style.scss
@use 'library';
@use 'override';
@debug library.$color;  //=> blue
```

Cargar un módulo:

En Sass no hace falta poner la extensión de una hoja de estilos para cargarla en un módulo.

> Para garantizar que las hojas de estilo funcionen en cualquier SO, Sass las carga mediante URL no por ruta de archivo. Por eso es importante que se usen las barras diagonales / y no las inversas \.

> Sass no requiere que se use ./ para importaciones relativas. Las importaciones relativas están directamente disponibles.

2. @forward

Esta regla permite que las funciones, mixins y variables estén disponibles cuando se usa la regla @use. Se asemejaría a un tipo primitivo de herencia.

```
// src/_list.scss
@mixin list-reset {
  margin: 0;
  padding: 0;
  list-style: none;
}

// bootstrap.scss
@forward "src/list";

// styles.scss
@use "bootstrap";

li {
  @include bootstrap.list-reset;
}
```

Prefijos:

Normalmente los módulos se usan con un namespace, y es muy posible que los namespaces no tengan sentido fuera del módulo en el que están definidos. Es por esto que la regla @forward tiene la opción de agregar un prefijo adicional a los miembros que reenvía.

> @forward "url/miarchivo.scss" as prefix-*;

```
// src/_list.scss
@mixin reset {
  margin: 0;
}

// bootstrap.scss

@forward "src/list" as list-*;
// styles.scss
@use "bootstrap";

li {
  @include bootstrap.list-reset;
}
```

Visibilidad:

Es posible que desees mantener algunos miembros privados para que solo su paquete pueda usarlos, o puede solicitar a sus usuarios que carguen algunos miembros de una manera diferente. Puede controlar exactamente qué miembros se pueden usar desde fuera de su archivo escribiendo @forward "url/url.scss" hide members;

```
// src/_list.scss
$horizontal-list-gap: 2em;  //esta variable pasaría a ser privada en
bootstrap.scss

@mixin list-reset {  //este mixin pasaría a ser privado en bootstrap.scss
  margin: 0;
  padding: 0;
  list-style: none;
}

@mixin list-horizontal {
  @include reset;

  li {
    display: inline-block;
    margin: {
      left: -2px;
      right: $horizontal-list-gap;
```

```
    }
  }
}

// bootstrap.scss
@forward "src/list" hide list-reset, $horizontal-list-gap;
```

Configuración de módulos:

Esta regla también puede cargar un módulo de configuración, funciona igual que @use con una nueva funcionalidad, con @forward se puede usar el flag **!default**, esto permite que un sólo módulo cambie el valor predeterminado para todo el fichero.

```
// _library.scss
$black: #000 !default;
$border-radius: 0.25rem !default;
$box-shadow: 0 0.5rem 1rem rgba($black, 0.15) !default;

code {
  border-radius: $border-radius;
  box-shadow: $box-shadow;
}

// _opinionated.scss
@forward 'library' with (
  $black: #222 !default,  //se sobreescribe la variable $black
  $border-radius: 0.1rem !default  //se sobreescribe la variable $border-
radius
);

// style.scss
@use 'opinionated' with ($black: #333);   //se sobreescribe la variable
$black
```

3. @import

Sass eliminará la regla @import del lenguaje por completo en los próximos años.

> Se desaconseja el uso del import por lo siguiente:
> Hace que todas las variables, mixins y funciones sean accesibles globalmente.
> No hay forma de definir miembros privados o selectores de marcadores de posición que fueran inaccesibles para las hojas de estilo posteriores.
> Cada vez que aparece un @import, Sass lo carga; lo que aumenta el tiempo de compilación y produce una salida inflada.

El nuevo sistema de módulos y la regla @use abordan todos estos problemas.

4. @mixin and @include

Los mixins le permiten definir estilos que se pueden reutilizar en toda su hoja de estilo. Facilitan evitar el uso de clases no semánticas como .float-left, y distribuir colecciones de estilos en bibliotecas.

> Los mixins se definen usando la @mixin Los mixins se incluyen en el contexto actual usando la @include

```
@mixin reset-list {   // Se crea mixin
  margin: 0;
  padding: 0;
  list-style: none;
}

@mixin horizontal-list {   // Se crea otro mixin y se incluye uno ya creado
  @include reset-list;     // además de un estilo propio
  li {
    font-size: 20px;
  }
}

nav ul {
  @include horizontal-list;
}
```

Los mixins también pueden aceptar argumentos

```
@mixin rtl($property, $ltr-value) {
  #{$property}: $ltr-value;
}

.sidebar {
  @include rtl(float, left);
}
```

5. @function

6. @extend

7. @error

8. @warn

9. @debug

10. @at-root