Sass Guidelines

Una guía de estilo subjetiva para escribir Sass sano, sostenible y escalable.

Sobre El Autor

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_author.md)

Mi nombre es Hugo Giraudel *(http://hugogiraudel.com)*, soy un desarrollador front-end nacido en Francia hace 26 años y que vive Berlín (Alemania) desde 2015, actualmente trabajo en N26 *(https://n26.com)*.

Llevo usando Sass desde hace varios años y soy el autor de muchos proyectos relacionados con Sass como por ejemplo SassDoc *(http://sassdoc.com)*, SitePoint Sass Reference *(http://sitepoint.com/sass-reference/)* y Sass-Compatibility *(http://sass-compatibility.github.io)*. Si sientes más curiosidad acerca de mis aportaciones a esta comunidad, echa un vistazo a esta lista *(http://github.com/HugoGiraudel/awesome-sass)*.

También soy el autor de un libro sobre CSS (en francés) titulado CSS3 Pratique du Design Web *(http://css3-pratique.fr/)* (*Eyrolles* editions), al igual que de otro sobre Sass (en inglés) llamado Jump Start Sass *(https://learnable.com/books/jump-start-sass)* (*Learnable* editions).

Contribuir

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_contributing.md)

Esta guía de Sass, es un proyecto abierto que mantengo en mi tiempo libre. Sobra decir que lleva una cantidad enorme de trabajo mantenerlo todo vigente, documentado y relevante. Afortunadamente, cuento con la ayuda de muchos colaboradores que hacen un gran trabajo manteniendo las docenas de traducciones diferentes que hay hoy en día. ¡Asegúrate de darles las gracias!

Aún asi, si sientes que te apetece contribuir, que sepas que *Twiteando* al respecto, difundiéndolo en la red, o incluso, arreglando algún pequeño error ortográfico. Abrir un ticket o a hacer un *pull-request* en el repositorio de GitHub *(https://github.com/HugoGiraudel/sass-guidelines)* sería de gran ayuda.

Por último, antes de empezar: si te ha gustado este documento, o si ha sido útil para ti o para tu equipo, por favor, considera hacer una donación para que pueda seguir trabajando en ello.

Acerca De Sass

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_sass.md)

Así se describe Sass *(http://sass-lang.com)* en su documentación *(http://sass-lang.com/documentation/file.SASS_REFERENCE.html)*:

> *Sass es una extensión de CSS que añade potencia y elegancia al lenguaje básico.*

El objetivo principal de Sass es el de corregir los defectos de CSS. Como todos sabemos, CSS no es el mejor lenguaje del mundo [cita requerida] y aunque es sencillo de aprender, puede convertirse en un auténtico desorden, especialmente en proyectos grandes.

Es aquí donde Sass cobra sentido, ya que actúa como un meta-lenguaje para mejorar la sintaxis de CSS con el fin de proporcionar características extra y herramientas útiles. Entretanto, Sass quiere ser conservador con respecto al lenguaje CSS.

La idea no es convertir CSS en un lenguaje de programación totalmente funcional; Sass solo quiere mejorar aspectos donde CSS falla. Por ello, empezar con Sass no es más difícil que aprender CSS: Sass simplemente agrega un par de características adicionales *(http://sitepoint.com/sass-reference/)*.

Dicho esto, hay muchas formas de utilizar estas funciones. Algunas buenas, algunas malas y otras, poco comunes. Esta guía tiene como finalidad darte un enfoque coherente y documentado para escribir Sass.

Ruby Sass O LibSass

El primer commit de Sass *(https://github.com/hcatlin/sass/commit/fa5048ba405619273e474a50400c7243fbff54fe)* se remonta a finales de 2006, hace más de 10 años. Obviamente se ha recorrido un largo camino desde entonces. A la primera versión desarrollado en Ruby, le sucedieron distintas variaciones; la más popular, LibSass *(https://github.com/sass/libsass)* (Escrita en C/C++) está ahora muy cerca de ser totalmente compatible con la versión original en Ruby.

En 2014, los equipos de Ruby Sass y LibSass decidieron esperar a que ambas versiones se sicronozaran antes de seguir adelante *(https://github.com/sass/libsass/wiki/The-LibSass-Compatibility-Plan)*. Desde entonces, LibSass ha continuado publicando nuevas versiones para igualar sus características con las de su hermano mayor. Así que he reunido y listado las últimas inconsistencias entre ambas, en el proyecto Sass-

Compatibility *(http://sass-compatibility.github.io)*. Si conoces otras incompatibilidades entre estas dos versiones y no aparecen listadas, por favor, házmelo saber abriendo un *issue*.

Volviendo a la tarea de elegir un compilador; en realidad, dependerá de tu proyecto. Si es un proyecto basado en ruby on Rails, es mejor utilizar Ruby Sass que es perfectamente adecuado para este caso. Además, ten en cuenta que Ruby Sass siempre será la implementación de referencia y por tanto liderará las funciones de LibSass. Pero si lo que estás buscando es cambiar de Ruby Sass a LibSass *(http://www.sitepoint.com/switching-ruby-sass-lib-sass/)*, este artículo es para tí.

Para proyectos que no están en Ruby que necesitan una integración de flujo de trabajo, LibSass es probablemente una mejor idea, ya que está dedicado principalmente a ser envuelto por cada lenguaje. Así que si quieres utilizar, por ejemplo, Node.js, node-sass *(https://github.com/sass/node-sass)* sería tu solución.

Sass O SCSS

Hay una gran confusión con respecto a la semántica del nombre *Sass*, y por una buena razón: Sass nombra tanto al preprocesador como a su propia sintaxis. No muy adecuado ¿verdad?

Verás, inicialmente Sass describe una sintaxis en la que su característica principal era la de ser sensible a las tabulaciones. Rápidamente, quienes mantienen Sass decidieron cerrar la brecha entre Sass y CSS, proporcionando una sintaxis amigable con CSS llamada *SCSS* también conocida como *Sassy CSS*. Su lema es: si es CSS válido, es SCSS válido.

Desde entonces, Sass (el preprocesador) ha estado proporcionando dos sintaxis diferentes *(http://www.sitepoint.com/whats-difference-sass-scss/)*: Sass (no todas en mayúsculas, por favor *(http://sassnotsass.com)*), también conocida como *sintaxis con tabulación*, y SCSS. La elección de cuál usar, depende mucho de tus preferencias, ya que ambas tienen características totalmente equivalentes. Llegados a este punto, es simplemente una cuestión de estética.

La sintaxis sensible a los espacios en blanco de Sass se basa en las tabulaciones para eliminar las llaves, los punto y coma y otros signos de puntuación, dando como resultado una sintaxis más corta y ligera. Mientras tanto, SCSS es más fácil de aprender, ya que en conjunto, se trata de añadir unos cuantos elementos adicionales a CSS.

Personalmente, prefiero SCSS a Sass ya que es mucho más parecido a CSS y más amigable para la mayoría de desarrolladores. Debido a esto, la usaré como sintaxis por defecto a lo largo de esta guía. Puedes cambiar a la sintaxis con tabulaciones de Sass pulsando en el panel de opciones.

Otros Preprocesadores

Sass es un preprocesador entre otros tantos. Su competidor más serio tiene que ser LESS *(http://lesscss.org/)*, un preprocesador basado en Node.js que se ha hecho muy popular gracias a que el famoso *framework* CSS Bootstrap *(http://getbootstrap.com/)* lo utilizaba hasta la versión 4. También está Stylus *(http://learnboost.github.io/stylus/)*, un preprocesador muy permisivo y flexible que si embargo es un poco más complicado de aprender y tiene una comunidad más pequeña.

¿Por qué preferir Sass a cualquier otro preprocesador? Esta sigue siendo una pregunta válida hoy en día. No hace mucho tiempo, soliamos recomendar el uso de Sass para los proyectos basados en Ruby, puesto que se desarrolló inicialmente en Ruby y por tanto funcionaba bien con Ruby on Rails. Ahora que LibSass se ha puesto al día (en su mayoría) con el Sass original, ya no es un consejo demasiado relevante.

Lo que me gusta de Sass es su enfoque conservador respecto a CSS. El diseño de Sass se basa en fuertes principios: gran parte del enfoque del diseño resultaba natural a los equipos desarrolladores principales, quienes creían que: a) añadir características adicionales tiene un coste de complejidad que debe ser justificado por su utilidad y b) debe ser fácil comprender lo que un bloque determinado de estilos está haciendo con solo observar dicho bloque. Por otra parte, Sass se centra con mucho más cuidado en los detalles que otros preprocesadores. Por mi parte, puedo decir que los desarrolladores principales se preocupan profundamente por mantener todos los pequeños detalles y casos de compatibilidad con CSS, asegurándose que cada comportamiento general sea consistente. En otras

palabras, Sass es un software orientado a solucionar problemas reales; ayudando a proporcionar funcionalidades útiles para CSS, justo donde CSS se queda corto.

Preprocesadores aparte, también debemos mencionar herramientas como PostCSS *(https://github.com/postcss/postcss)* y cssnext *(https://cssnext.-github.io/)* que han recibido una exposición significativas en los últimos meses.

PostCSS es comunmente (e incorrectamente) referido como un “post-procesador”. La suposición, combinada con el desafortunado nombre, es que PostCSS analiza sobre CSS lo que ya ha sido procesado por un preprocesador. Si bien puede funcionar de esta manera, no es un requisito para PostCSS, lo que lo convierte en realidad en sólo un “procesador”.

Te permite acceder a “tokens” de tus hojas de estilo (como selectores, propiedades y valores), los procesa con JavaScript para realizar operaciones de cualquier tipo y compila los resultados a CSS. Por ejemplo, la popular librería de prefijos Autoprefixer *(https://github.com/postcss/autoprefixer)* está creada con PostCss. Analiza todas las reglas para comprobar si estas necesitan algún prefijo específico de proveedor (*vendor prefixes*) haciendo referencia a la herramienta de soporte de navegadores CanIUse *(http://caniu-se.com)* y, a continuación, elimina o añade los prefijos necesarios.

Esto es increíblemente potente y genial para la construcción de librerías que funcionan con cualquier preprocesador (así como vanilla CSS), pero PostCSS no es particularmente fácil de usar todavía. Tienes que saber un poco de JavaScript para construir cualquier cosa con él, y su API puede ser confusa algunas veces. Mientras que Sass sólo proporciona un conjunto de características que son útiles para escribir CSS, PostCSS proporciona acceso directo al AST de CSS (*árbol de sintaxis abstracto*) y a JavaScript.

En resumen, Sass algo fácil y resolverá la mayoría de tus problemas. Por otra parte, PostCSS puede ser un poco complicado (si no se te da bien JavaScript), pero resulta ser increíblemente potente. No hay ninguna razón por la que no puedas usar ambos. De echo, PostCSS ofrece un analizador oficial de SCSS solo para esto.

Nota — Gracias a Cory Simmons *(https://github.com/corysimmons)* por su ayuda experta en esta sección.

Introducción

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_introduction.md)

Por Qué Una Guía De Estilo

Una guía de estilo no es sólo un documento agradable de leer mientras te imaginas tu código perfecto. Se trata de un documento clave en la vida de un proyecto, describe cómo y por qué debe escribirse el código. Puede parecer una exageración para pequeños proyectos, pero es de gran ayuda para mantener el código limpio, escalable y fácil de mantener.

Sobra decir, que cuantos más desarrolladores participen en el proyecto, más necesarias son las guías de estilo. En la misma línea, cuanto más grande es el proyecto, la guía de estilo se convierte en una necesidad.

Harry Roberts *(http://csswizardry.com)* Lo explica muy bien en su CSS Guidelines *(http://cssguidelin.es/#the-importance-of-a-styleguide)*:

Una guía de estilo (nota, no es una guía de estilo visual) es una valiosa herramienta para los equipos que:

- *construyen y mantienen productos durante un tiempo razonablemente largo;*
- *tienen desarrolladores con diferentes habilidades y especialidades;*
- *tienen un número variable de desarrolladores que trabajan en el proyecto en un momento determinado;*
- *hay una constante contratación de personal nuevo;*
- *tienen una serie de código base en el que los desarrolladores entran y salen constantemente.*

Renuncia De Responsabilidad

Primero lo primero: **esta no es una guía de CSS**. Este documento no tratará sobre convenciones de nomenclatura para clases CSS, ni sobre los patrones modulares de CSS, ni mucho menos, sobre la cuestión de los IDs en el mundo CSS. Esta guía solo busca estudiar el contenido relacionado específicamente con Sass.

Además, esta guía está elaborada por mi y por tanto es **subjetiva y personal**. Piensa en ella como una colección de metodologías y consejos que he reunido y pulido durante los últimos años. También me da la oportunidad de recoger un puñado de recursos interesantes, así que asegurate de echarle un vistazo a las *lecturas adicionales*.

Obviamente, esta no es la única forma de hacer las cosas, y puede que encaje o no en tu proyecto. Siéntete libre de elegir lo que te interese y de adaptarlo a tus necesidades. Como decimos, *tu experiencia puede variar*.

Principios Clave

Al final del día, si hay una cosa que me gustaría que aprendieses de toda esta guía, es que **Sass debe mantenerse tan simple como sea posible** *(http://www.sitepoint.com/keep-sass-simple/)*.

Gracias a mis tontos experimentos, como por ejemplo: Operadores bit a bit *(https://github.com/HugoGiraudel/SassyBitwise)*, iteradores y generadores *(https://github.com/HugoGiraudel/SassyIteratorsGenerators)* y analizador JSON *(https://github.com/HugoGiraudel/SassyJSON)* en Sass, podemos ser conscientes de lo que se puede hacer con este preprocesador.

Mientras CSS es un lenguaje sencillo. Sass, destinado a escribir CSS, no debería ser mucho más complejo que el CSS normal. El principio KISS *(http://es.wikipedia.org/wiki/Principio_KISS)* (Keep It Simple Stupid) es la clave aquí, e incluso puede tener prioridad respecto al principio DRY *(http://es.wikipedia.org/wiki/No_te_repitas)* (Don't Repeat Yourself) en algunas circunstancias.

Algunas veces es mejor repetirse un poco para que el proyecto sea fácil de mantener, antes que construir un sistema demasiado pesado, inabarcable e innecesariamente complicado que genere un código imposible de mantener debido a su complejidad.

También, si me permites que cite a Harry Roberts *(https://csswizardry.com)* una vez más, **el pragmatismo prevalece sobre la perfección**. En algún momento, es probable que te des cuenta que estás yendo en contra de las reglas descritas aquí. Si tiene sentido, si crees que está bien, hazlo. El código solo es un medio, no un fin.

Ampliación De La Guía

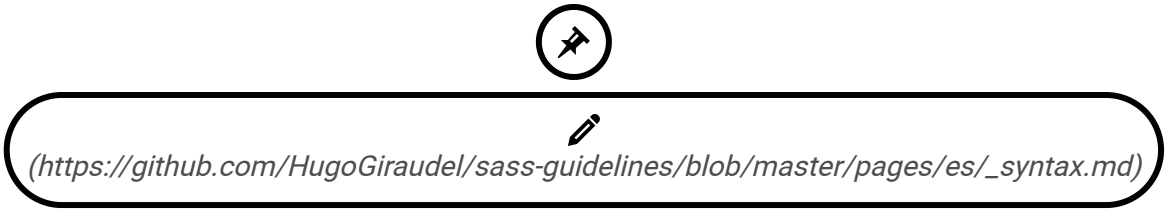
Una gran parte de esta guía es claramente subjetiva. Llevo leyendo y escribiendo Sass bastantes años, hasta el punto en el que tengo muchos principios en lo que se refiere a escribir una hoja de estilo limpia. Entiendo que esto no pueda complacer o encajar con todo el mundo, lo cual es perfectamente normal.

Aunque, creo que las guías se hacen para ser constantemente extendidas. Extender las directrices de Sass podría ser tan simple como tener un documento que indica que el código está siguiendo estas guías de estilo a excepción de algunas normas; en cuyo caso, se deberían especificar estas normas a continuación.

Un ejemplo de extensión de una guía de estilo puede encontrarse en el repositorio de SassDoc *(https://github.com/SassDoc/sassdoc/blob/master/GUIDELINES.md)*:

> *Esta es una extensión de la guía de estilo de Node (https://github.com/felixge/node-style-guide) de Felix Geisendörfer. Cualquier cosa de este documento anula lo que podría decirse en la Guía de Estilo de Node.*

Sintaxis Y Formato



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_syntax.md)

En mi opinión, la primera cosa que debe hacer una guía de estilo es describir la forma en que queremos que luzca nuestro código.

Cuando varios desarrolladores están involucrados en la escritura CSS dentro del mismo proyecto o proyectos, es sólo cuestión de tiempo antes de que uno de ellos empiece a hacer las cosas a su manera. Las guías de estilo de código que fomentan la coherencia no solo previenen esto, sino que también ayudan a la hora de leer y actualizar el código.

A grandes rasgos, lo que queremos (humildemente inspirados en CSS Guidelines *(http://cssguidelin.es/#syntax-and-formatting)* es):

- dos (2) espacios en blanco, en lugar de tabulaciones;
- idealmente, líneas de 80 caracteres;
- reglas CSS multilínea correctamente escritas;
- buen uso de los espacios en blanco.

```
// Yep
.foo {
  display: block;
  overflow: hidden;
  padding: 0 1em;
}

// Nope
.foo {
    display: block; overflow: hidden;

    padding: 0 1em;
}
```

Cadenas

Lo creas o no, las cadenas (*strings*) juegan un papel importante en los ecosistemas de CSS y Sass. La mayoría de los valores CSS suelen ser longitudes o identificadores, así que es bastante crucial que cumplas ciertas pautas cuando se trabaja con cadenas en Sass.

CODIFICACIÓN

Para evitar cualquier posible problema con la codificación de caracteres, es muy recomendable forzar la codificación UTF-8 *(http://es.wikipedia.org/wiki/UTF-8)* en la hoja de estilo principal usando la directiva `@charset`. Asegúrate que es el primer elemento de la hoja de estilo y que no hay ningún otro caracter de ningún tipo antes.

```
@charset 'utf-8';
```

COMILLAS

En CSS, las cadenas (*strings*) no tienen por qué estar entre comillas, ni siquiera aquellas que contienen espacios. Toma como ejemplo, los nombres de las tipografías: no importa si las pones entre comillas o no para que el analizador sintáctico CSS las entienda.

Debido a esto, Sass *tampoco* necesita que las cadenas estén entre comillas. Incluso mejor (y con *suerte* me darás la razón) una cadena entre comillas es equivalente a su versión gemela sin comillas (por ejemplo, `'abc'` es estrictamente igual a `abc`).

Dicho esto, los lenguajes que no requieren que las cadenas estén entre comillas, son definitivamente una minoría, y es por lo que **las cadenas siempre deben ir entre comillas simples** (`'`) en Sass (las comillas simples son mucho más fáciles de escribir en los teclados *qwerty*). Además de la coherencia con otros lenguajes, entre ellos el primo de CSS, Javascript, hay otras razones para esta elección:

- los nombres de los colores son tratados directamente como colores cuando no están entre comillas, lo que provoca serios problemas;
- muchos resaltadores de sintaxis se volverían locos sin las comillas;
- ayuda a la legibilidad;
- no hay una razón válida para no entrecomillar las cadenas.

```
// Yep
$direction: 'left';

// Nope
$direction: left;
```

Nota — Según las especificaciones CSS, la directiva `@charset` debe estar declarada con comillas dobles para ser considerada válida *(http://www.w3.org/TR/css3-syntax/#charset-rule)*. Sin embargo, Sass se preocupa de esto cuando compila el CSS para que su creación no tenga ningún impacto con el resultado final. Puedes usar sin ningún problema las comillas simples, incluso para `@charset`.

CADENAS COMO VALORES CSS

Los valores CSS específicos (identificadores) como `initial` o `sans-serif` no necesitan llevar comillas. De hecho, la declaración `font-family: 'sans-serif'` fallará porque CSS está esperando un identificador, no una cadena entre comillas. Por ello, no pondremos nunca estos valores con comillas.

```
// Yep
$font-type: sans-serif;

// Nope
$font-type: 'sans-serif';

// Okay I guess
$font-type: unquote('sans-serif');
```

Por lo tanto, podemos hacer una distinción entre las cadenas destinadas a ser utilizadas como valores CSS (identificadores CSS) como en el ejemplo anterior, y las cadenas que son un tipo de datos Sass, por ejemplo, las claves de mapa.

No ponemos entre comillas la primera, pero si envolvemos la segunda con comillas simples.

CADENAS QUE CONTIENEN COMILLAS

Si una cadena contiene una o varias comillas simples, se podría considerar envolver la cadena con comillas dobles (`"`), con el fin de evitar escapar demasiados caracteres dentro de la cadena.

```
// Okay
@warn 'You can\'t do that.';

// Okay
@warn "You can't do that.";
```

URLS

Las URLs deben ir entre comillas por las mismas razones que se explican anteriormente:

```
// Yep
.foo {
  background-image: url('/images/kittens.jpg');
}

// Nope
.foo {
  background-image: url(/images/kittens.jpg);
}
```

Números

En Sass, número es un tipo de datos que incluye de todo, desde números sin unidades a longitudes, pasando por duraciones, frecuencias y ángulos, entre otros. Esto permite que se ejecuten cálculos sobre tales medidas.

CEROS

Siempre se deben mostrar los ceros a la izquierda antes de un valor decimal menor que uno. Nunca mostrar los ceros finales.

```
// Yep
.foo {
  padding: 2em;
  opacity: 0.5;
}

// Nope
.foo {
  padding: 2.0em;
  opacity: .5;
}
```

Nota — Sublime Text y otros editores proporcionan una búsqueda que te permite reemplazar este valor usando una expresión regular, es muy fácil añadir un cero a la izquierda a (casi todos) los número flotantes. Solamente reemplaza `\s+\.(\d+)` con `\ 0.$1`. No olvides el espacio antes del `0`.

UNIDADES

Cuando se trata de longitudes, el `0` nunca debe llevar el nombre de la unidad.

```
// Yep
$length: 0;

// Nope
$length: 0em;
```

Nota — Ten cuidado, esta práctica solo debe limitarse a las longitudes. No se permite tener un cero sin unidades en propiedades como `transition-delay`. Teóricamente, si un cero sin unidades es especificado para una duración, la declaración es marcada como inválida y debería quedar descartada. No todos los navegadores son tan estrictos, pero algunos si. En resumen, solo se omiten las unidades para las longitudes

El error más común que se me ocurre respecto a los números en Sass, es el de pensar que las unidades son solo una serie de cadenas que se añaden a un número. Aunque parezca verdad, esta sin duda, no es la forma en la que funcionan las unidades. Piensa en las unidades como símbolos algebráicos. Por ejemplo, en el mundo real, multiplicar 5 metros por 5 metros, da como resultado 25 metros cuadrados. La misma lógica se aplica a Sass.

Para agregar una unidad a un número, hay que multiplicar este número por *1 unidad*.

```
$value: 42;

// Yep
$length: $value * 1px;

// Nope
$length: $value + px;
```

Ten en cuenta que sumando un *valor de 0 de unidad* también funciona, pero prefiero recomendar el método antes mencionado, ya que sumar una *unidad 0* puede resultar confuso. De hecho, cuando se trata de convertir un número

a otra unidad compatible, emplear el truco del 0, no funcionará. Puedes leer más sobre esto en este artículo *(http://css-tricks.com/snippets/sass/correctly-adding-unit-number/)*.

```
$value: 42 + 0px;
// -> 42px

$value: 1in + 0px;
// -> 1in

$value: 0px + 1in;
// -> 96px
```

Al final, siempre depende lo que estés tratando de conseguir. Solo ten en cuenta que añadir la unidad como una cadena de caracteres no es una buena manera de proceder.

Para eliminar la unidad de un valor, hay que dividirlo por *1 unidad de su mismo tipo*.

```
$length: 42px;

// Yep
$value: $length / 1px;

// Nope
$value: str-slice($length + unquote(''), 1, 2);
```

Al añadir a un número una unidad en forma de cadena, el resultado es una cadena, impidiendo cualquier operación adicional con dicho valor. Separando la parte numérica de un número con una unidad también devolverá una cadena. Usa longitudes, no cadenas *(http://hugogiraudel.com/2013/09/03/use-lengths-not-strings/)*.

CÁLCULOS

Los cálculos numéricos de nivel superior deben ir siempre entre paréntesis lo cual, no solo mejorará drásticamente la legibilidad, sino que también evitará algunos casos extremos al forzar a Sass a evaluar los contenidos de los paréntesis.

```
// Yep
.foo {
  width: (100% / 3);
}

// Nope
.foo {
  width: 100% / 3;
}
```

NÚMEROS MÁGICOS

Un “número mágico” es un término de programación de la vieja escuela *(http://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants)* para *las constantes numéricas sin nombre*. Básicamente, es solo un número aleatorio que *simplemente funciona* y sin embargo no está ligado con ninguna explicación lógica.

No hace falta decir que **los números mágicos son una plaga y que se deben evitar a toda costa**. Cuando no se puede encontrar una explicación razonable a por qué un número funciona, añade un comentario claro y completo explicando como has llegado hasta allí y por qué crees que funciona. Admitir que no sabes por qué algo funciona es mucho más útil para el siguiente desarrollador que dejarle tener que averiguar lo que está pasando desde el principio.

```
/**
 * 1. Número mágico. Este valor es el más bajo que puc
 * `.foo` con su padre. Idealmente, deberíamos arregla
 */
.foo {
  top: 0.327em; /* 1 */
}
```

Relacionado con esto, CSS-Tricks tiene un excelente artículo *(http://css-tricks.com/magic-numbers-in-css/)* acerca de los números mágicos en CSS que te animo a que leas.

Colores

Los colores ocupan un lugar importante en el lenguaje CSS. Naturalmente, Sass termina siendo un valioso aliado cuando se trata de la manipulación de colores, especialmente, al proporcionar un puñado de funciones potentes *(http://sass-lang.com/documentation/Sass/Script/Functions.html)*.

Sass es tan útil cuando se trata de manipular los colores, que han florecido artículos por todo Internet sobre este tema. Te recomiendo algunas lecturas:

- ¿Cómo ir programáticamente de un color a otro? - En inglés *(http://the-sassway.com/advanced/how-to-programtically-go-from-one-color-to-another-in-sass)*
- Usando Sass para construir paletas de color - En inglés *(http://www.sitepoint.com/using-sass-build-color-palettes/)*
- Tratando con esquemas de color en Sass - En inglés *(http://www.sitepoint.com/dealing-color-schemes-sass/)*

FORMATOS DE COLOR

Con el objetivo de que hacer colores sean tan simple como sea posible, mi consejo es que respetes el siguiente orden de preferencia de los formatos de color:

1. Notación HSL *(http://es.wikipedia.org/wiki/Modelo_de_color_HSV)*;
2. Notación RGB *(http://es.wikipedia.org/wiki/RGB)*;
3. Notación hexadecimal. Preferiblemente en minúsculas y en formato corto.

Las palabras clave para denominar colores en CSS no deben ser usadas, a menos que se utilicen para prototipos rápidos. De hecho, son palabras inglesas y algunas de ellas hacen un trabajo bastante malo al describir el color que representan, especialmente para hablantes no nativos. Además de eso, las palabras clave no son perfectamente semánticas; Por ejemplo `grey` es en realidad más oscuro que `darkgrey`, y la confusión entre `grey` y `gray` puede llevar a usos inconsistentes de este color.

La representación HSL no solo es la más sencilla de comprender para el cerebro humano[cita requerida], sino que también facilita a los autores de las hojas de estilo modificar el color al ajustar el tono, la saturación y la luminosidad de forma individual.

RGB todavía tiene la ventaja de mostrar de inmediato si el color tiene más de azul, o de verde o de rojo. Por lo tanto, podría ser mejor que HSL en algunas situaciones, especialmente cuando se describe un rojo, verde o azul puro. Pero esto no significa que sea fácil construir un color con estas tres partes.

Por último, el formato hexadecimal está cerca de ser indescifrable para la mente humana. Úsalo como último recurso, si tienes que hacerlo.

```
// Yep
.foo {
  color: hsl(0, 100%, 50%);
}

// Also yep
.foo {
  color: rgb(255, 0, 0);
}

// Meh
.foo {
  color: #f00;
}

// Nope
.foo {
  color: #FF0000;
}

// Nope
.foo {
  color: red;
}
```

Al usar HSL o notación RGB, añade siempre un espacio simple después de la coma (`,`) y ningún espacio entre el paréntesis (`(`, `)`) y el contenido.

```
// Yep
.foo {
  color: rgba(0, 0, 0, 0.1);
  background: hsl(300, 100%, 100%);
}

// Nope
.foo {
  color: rgba(0,0,0,0.1);
  background: hsl( 300, 100%, 100% );
}
```

COLORES Y VARIABLES

Cuando se utiliza un color más de una vez, guárdala en una variable con un nombre significativo que represente al color.

```
$sass-pink: hsl(330, 50%, 60%);
```

Ahora puedes usar esta variable en cualquier lugar. Sin embargo, si su uso está fuertemente ligado a un tema, te desaconsejaría usar esta variable como tal. En su lugar, guárdala en otra variable con un nombre que explica como se debe utilizar.

```
$main-theme-color: $sass-pink;
```

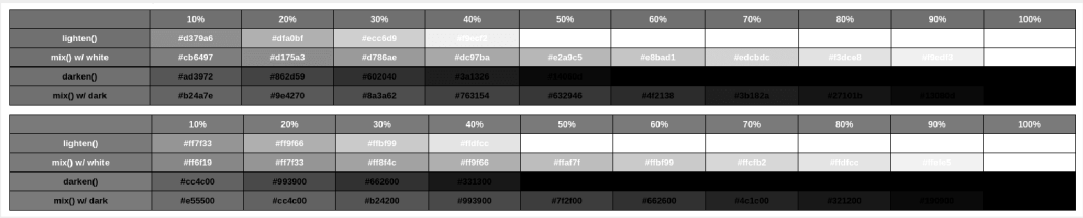
Al hacer esto, evitarás que algún cambio en el tema resulte en algo como `$sass-pink: blue`. Este artículo *(http://davidwalsh.name/sass-color-variables-dont-suck)* hace un buen trabajo explicando por qué pensar el nombre de tus variables de color es importante.

ACLARANDO Y OSCURECIENDO COLORES

Tanto las funciones `lighten` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#lighten-instance_method)* como `darken` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#darken-instance_method)* manipulan la luminosidad de un color en el espacio HSL añadiendo o restando a la luminosidad de dicho espacio. Básicamente, son alias para el parámetro `$lightness` de la función `adjust-color` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#adjust_color-instance_method)*

La cuestión es que esas funciones a menudo no proporcionan el resultado esperado. Por otro lado, la función `mix` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#mix-instance_method)* es una buena manera para aclarar u oscurecer un color al mezclarlo con `blanco` or `negro`.

La ventaja de usar `mix` en lugar de una de las funciones anteriormente mencionadas, es que con `mix` irá progresivamente a negro (o blanco) a medida que se disminuye la proporción de color, mientras que `darken` y `lighten` apagará rápidamente el color hasta llegar a negro o blanco.



	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
lighten()	#d379a6	#dfa0bf	#ecc6d9	#f5ecf2						
mix() w/ white	#cb6497	#d175a3	#d786ae	#dc97ba	#e2a9c5	#e8bad1	#edcbdc	#f3dce8	#f8edf4	
darken()	#ad3972	#862d59	#602040	#3a1326	#150509					
mix() w/ dark	#b24a7e	#9e4270	#8a3a62	#763154	#632946	#4f2138	#3b182a	#27101b	#140809	

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
lighten()	#ff7f33	#ff9f66	#ffbf99	#ffdfcc						
mix() w/ white	#ff6f19	#ff7f33	#ff8f4c	#ff9f66	#ffaf7f	#ffbf99	#ffcfb2	#ffdfcc	#ffefe5	
darken()	#cc4c00	#993900	#662600	#331300						
mix() w/ dark	#e55500	#cc4c00	#b24200	#993900	#7f2f00	#662600	#4c1c00	#331200	#190900	

Ilustración de la diferencia entre `lighten/darken` y `mix` por KatieK (http://codepen.io/KatieK2/pen/tejhz/)

Si no quieres escribir la función `mix` cada vez que quieras usarla, puedes crear dos funciones muy sencillas de utilizar, `tint` y `shade` (que también son parte de Compass *(http://compass-style.org/reference/compass/helpers/colors/#shade)*) para conseguir el mismo resultado:

```
/// Aclarar ligeramente un color
/// @access public
/// @param {Color} $color - color a matizar
/// @param {Number} $percentage - porcentaje del `$col
/// @return {Color}
@function tint($color, $percentage) {
  @return mix(white, $color, $percentage);
}

/// Oscurecer ligeramente un color
/// @access public
/// @param {Color} $color - color a oscurecer
/// @param {Number} $percentage - porcentaje de `$colc
/// @return {Color}
@function shade($color, $percentage) {
  @return mix(black, $color, $percentage);
}
```

Nota — La función `scale-color` *(http://sass-lang.com/documentation/Sass/Script/Functions.html#scale_color-instance_method)* (escala de color) está diseñada para escalar las propiedades de una forma más fluida, al tener en cuenta qué tan altas o bajas son en el momento inicial. Debe proporcionar resultados tan agradables como los de `mix` pero con una nomenclatura mucho más clara. Sin embargo, el factor de escalado no es exactamente el mismo.

Listas

Las listas son el equivalente en Sass a los arreglos (*Arrays*). Una lista es una estructura de datos plana (a diferencia de los Mapas) destinada a almacenar valores de cualquier tipo (incluyendo listas, lo que conduce a las listas anidadas).

Las listas deberían respetar las siguientes pautas:

- pueden ser en una línea o multilínea
- deben usar múltiples líneas si su longitud es mayor a 80 caracteres;

- a menos que se utilice para CSS, siempre usará la coma como separador;
- siempre debe ir entre paréntesis;
- Usar coma final si hay múltiples líneas, pero no cuando es una sola.

```
// Yep
$font-stack: ('Helvetica', 'Arial', sans-serif);

// Yep
$font-stack: (
  'Helvetica',
  'Arial',
  sans-serif,
);

// Nope
$font-stack: 'Helvetica' 'Arial' sans-serif;

// Nope
$font-stack: 'Helvetica', 'Arial', sans-serif;

// Nope
$font-stack: ('Helvetica', 'Arial', sans-serif,);
```

Al añadir nuevos elementos a una lista, utiliza siempre la API proporcionada. No trates de añadir nuevos elementos manualmente.

```
$shadows: (0 42px 13.37px hotpink);

// Yep
$shadows: append($shadows, $shadow, comma);

// Nope
$shadows: $shadows, $shadow;
```

En este artículo *(http://hugogiraudel.com/2013/07/15/understanding-sass-lists/)*, explico un montón de trucos y consejos para manejar y manipular las listas correctamente en Sass.

Mapas

Con Sass, se pueden definir mapas - el término en Sass para los arreglos asociativos, *hashes* o incluso objetos JavaScript. Un mapa es una estructura de datos que asocia claves a valores. Tanto las claves como los valores, pueden ser de cualquier tipo de dato, incluyendo mapas, aunque yo no recomendaría usar tipos de datos tan complejos como claves de un mapa, solo por salud mental.

Los mapas deberían estar escritos de la siguiente manera:

- poner un espacio después de los dos puntos (`:`);
- la llave de apertura (`(`) debe ir en la misma línea que los dos puntos (`:`);
- poner las **claves entre comillas** si son cadenas (lo que representa el 99% de los casos);
- cada par clave/valor debe ir en su propia línea;
- poner coma (`,`) al final de cada par clave/valor;
- poner **coma final** (`,`) en el último elemento para que sea más sencillo añadir, eliminar o reordenarlos;
- la llave de cierre (`)`) debe ir en una línea nueva;
- no poner espacio o línea nueva entre la llave de cierre (`)`) y el punto y coma (`;`).

Ejemplo:

```
// Yep
$breakpoints: (
  'small': 767px,
  'medium': 992px,
  'large': 1200px,
);

// Nope
$breakpoints: ( small: 767px, medium: 992px, large: 12
```

Los escritos acerca de los mapas de Sass son muchos dado cuánto se anhelaba esta característica. Aquí hay 3 que recomiendo: Usando Mapas en Sass - En inglés *(http://www.sitepoint.com/using-sass-maps/)*, Funciones

adicionales de mapas en Sass - En inglés *(http://www.sitepoint.com/extra-map-functions-sass/)*, Sass real, mapas reales - En inglés *(http://blog.grayg-hostvisuals.com/sass/real-sass-real-maps/)*.

Conjunto De Reglas CSS

Llegados a este punto, esto es básicamente una revisión de lo que todo el mundo ya sabe, pero esta es la forma en la que un conjunto de reglas CSS deben estar escritas (por lo menos según la mayoría de las guías, incluyendo la Guía de estilo de CSS *(http://cssguidelin.es/#anatomy-of-a-ruleset)*):

- los selectores relacionados deben ir en la misma línea; los selectores no vinculados en líneas nuevas;
- la llave de apertura (`{`) debe separarse del selector usando un espacio;
- cada declaración debe ir en una línea nueva;
- añadir un espacio después de los dos puntos (`:`);
- poner punto y coma (`;`) al final de todas las declaraciones;
- la llave de cierre (`}`) debe ir en una línea nueva;
- añadir una línea después de la llave de cierre (`}`).

Ejemplo:

```
// Yep
.foo, .foo-bar,
.baz {
  display: block;
  overflow: hidden;
  margin: 0 auto;
}

// Nope
.foo,
.foo-bar, .baz {
    display: block;
    overflow: hidden;
    margin: 0 auto }
```

Adicionalmente a esas guías CSS, queremos prestar especial atención a las siguientes pautas:

- las variables locales se declaran antes que cualquier otra y están espaciadas por un salto de línea;
- las llamadas a los *mixin* sin `@content` deben ir antes de cualquier declaración;
- los selectores anidados van siempre después de un salto de línea;
- las llamadas a los *mixin* con `@content` deben ir después de cualquier selector anidado;
- no usar un salto de línea antes de una llave de cierre (`}`).

Ejemplo:

```
.foo, .foo-bar,
.baz {
  $length: 42em;

  @include ellipsis;
  @include size($length);
  display: block;
  overflow: hidden;
  margin: 0 auto;

  &:hover {
    color: red;
  }

  @include respond-to('small') {
    overflow: visible;
  }
}
```

Clasificación De Declaraciones

No se me ocurren muchos temas donde las opiniones estén tan divididas como en lo que respecta a la clasificación de las declaraciones en CSS. En

concreto, hay dos bandos aquí:

- mantener un estricto orden alfabético;
- ordenar las declaraciones por tipo (posición, *display*, colores, tipografía, varios…).

Hay pros y contras para estas dos posturas. Por una parte, el orden alfabético es universal (por lo menos para los idiomas que usan el alfabeto latino) así que no hay discusión posible a la hora de poner una propiedad antes que otra. Sin embargo, me parece extremadamente raro ver que propiedades como, `bottom` y `top` no estén la una justo después de la otra. ¿Por qué las animaciones deben aparecer antes que el tipo de display? Hay muchas singularidades con respecto al orden alfabético.

```
.foo {
  background: black;
  bottom: 0;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
  height: 100px;
  overflow: hidden;
  position: absolute;
  right: 0;
  width: 100px;
}
```

Por otra parte, ordenar las propiedades por tipo tiene mucho sentido. Cada declaración relacionada está reunida, `top` y `bottom` están una debajo de la otra y leer un conjunto de reglas parece como leer un cuento corto. Pero a menos que mantengas algunas convenciones como las que se exponen en CSS idiomático *(https://github.com/necolas/idiomatic-css)*, hay mucho espacio para la interpretación en esta forma de escribir CSS. ¿Dónde debería ir `white-space`: en tipografía o en display?, ¿Dónde pertenece `overflow` exactamente?, ¿Cuál es el orden de una propiedad dentro de un grupo? (podría ser en orden alfabético, ¡oh! ironía)

```
.foo {
  height: 100px;
  width: 100px;
  overflow: hidden;
  position: absolute;
  bottom: 0;
  right: 0;
  background: black;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
}
```

También hay otra rama interesante en cuanto al orden de propiedades, llamado CSS concéntrico *(https://github.com/brandon-rhodes/Concentric-CSS)*, que parece ser bastante popular. Básicamente el CSS concéntrico se basa en el modelo de caja para definir un orden: se empieza fuera y se mueve hacia dentro.

```
.foo {
  width: 100px;
  height: 100px;
  position: absolute;
  right: 0;
  bottom: 0;
  background: black;
  overflow: hidden;
  color: white;
  font-weight: bold;
  font-size: 1.5em;
}
```

Debo decir que ni yo mismo puedo decidirme. Una encuesta reciente en CSS-Tricks *(http://css-tricks.com/poll-results-how-do-you-order-your-css-properties/)* determinó que más de un 45% de los desarrolladores ordenan sus declaraciones por tipo, frente el 14% que lo hace alfabéticamente. También hay un 39% que lo hace completamente al azar, incluido yo mismo.

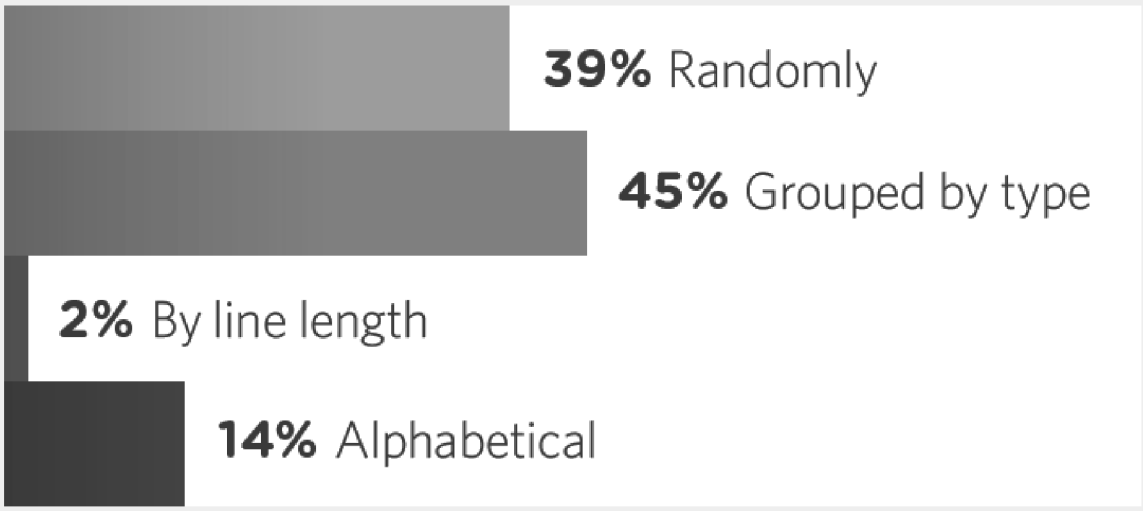


Gráfico que muestra cómo los desarrolladores ordenan sus declaraciones CSS

Debido a esto, no voy a imponer una opción en concreto en esta guía de estilo. Elige la que más te guste, siempre y cuando sea coherente con el resto de tus hojas de estilo (es decir, no la opción *al azar*).

Nota — Un estudio reciente *(http://peteschuster.com/2014/12/reduce-file-size-css-sorting/)* muestra que usando CSS Comb *(https://github.-com/csscomb/csscomb.js)* (que ordena por tipo *(https://github.com/csscomb/csscomb.js/blob/master/config/css-comb.json)*) para clasificar las declaraciones CSS termina acortando el tamaño promedio de los archivos bajo compresión Gzip en un 2.7% frente al 1.3% cuando se ordenan alfabéticamente.

Anidamiento De Selectores

Una característica particular que aporta Sass y que está siendo muy mal utilizada por muchos desarrolladores es el *anidamiento de selectores*. El anidamiento de selectores ofrece una forma para que los autores de las hojas de estilo puedan calcular selectores largos anidando selectores más cortos dentro de ellos.

REGLA GENERAL

Por ejemplo, el siguiente anidamiento Sass:

```
.foo {
  .bar {
    &:hover {
      color: red;
    }
  }
}
```

…generará el siguiente CSS:

```
.foo .bar:hover {
  color: red;
}
```

En la misma línea, desde Sass 3.3 es posible usar la referencia al selector actual (`&`) para generar selectores avanzados. Por ejemplo:

```
.foo {
  &-bar {
    color: red;
  }
}
```

…generará el siguiente CSS:

```
.foo-bar {
  color: red;
}
```

Este método se utiliza a menudo junto con las convenciones de nombrado BEM *(http://csswizardry.com/2013/01/mindbemding-getting-your-head-round-bem-syntax/)* para generar los selectores `.block__element` y `.block--modifier` basados en los selectores originales (por ejemplo, `.block` en este caso).

> **Nota —** Aunque podría ser anecdótico, generar nuevos selectores a partir de la referencia del selector actual (`&`) hace que dichos selectores no se puedan buscar en la base del código ya que no existen per se.

El problema con la anidación de selectores es que en última instancia hace que el código sea más difícil de leer. Se debe calcular mentalmente el selector resultante de los distintos niveles de sangría; no siempre resulta obvio conocer el código resultante en CSS.

Esta afirmación se vuelve más verdadera en cuanto los selectores se hacen más largos y las referencias al selector actual (`&`) más frecuentes. En algún punto, el riesgo de perder la pista y no poder entender lo que está pasando es tan alto que no merece la pena.

Para evitar estas situaciónes, hemos hablado mucho sobre la regla de Orgien - En inglés *(http://thesassway.com/beginner/the-inception-rule)* desde hace algunos años. Recomendaba no anidar los selectores más allá de 3 niveles de profundidad, como referencia a la película *Inception* de Christopher Nolan. Yo sería mucho más drástico y recomendaría **evitar la anidación de selectores tanto como sea posible**.

Sin embargo, es evidente que hay algunas excepciones a esta regla como se verá en la siguiente sección, esta opinión es bastante popular y puedes leer más sobre ella en Ten cuidado con la anidación de selectores - En inglés *(http://www.sitepoint.com/beware-selector-nesting-sass/)* y Evita la anidación de selectores para obtener un CSS más modular - En inglés *(http://thesassway.com/intermediate/avoid-nested-selectors-for-more-modular-css)*.

EXCEPCIONES

Para los principiantes, se permite e incluso se recomienda anidar pseudo-clases y pseudo-elementos dentro del selector inicial.

```
.foo {
  color: red;

  &:hover {
    color: green;
  }

  &::before {
    content: 'pseudo-element';
  }
}
```

Usar la anidación para las pseudo-clases y los pseudo-elementos no solo tiene sentido (porque trata con selectores relacionados), sino que también ayuda a mantener todo lo relacionado con un componente en un mismo lugar.

Además, cuando se utilizan clases de estado independientes del dispositivo como `.is-active`, está perfectamente bien anidar dentro del selector del componente para mantenerlo todo ordenado.

```
.foo {
  // …

  &.is-active {
    font-weight: bold;
  }
}
```

Por último, pero no menos importante, cuando se da estilo a un elemento, este pasa a estar contenido en otro elemento específico, también está bien utilizar la anidación para mantener todo lo relacionado con el componente en el mismo lugar.

```
.foo {
  // …

  .no-opacity & {
    display: none;
  }
}
```

Como en todo, los detalles son algo irrelevante, la coherencia es la clave. Si te sientes completamente confiado con la anidaciones de selectores, entonces utilizala. Sólo asegurate de que todo tu equipo está de acuerdo con ello.

Convenciones De Nomenclatura

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_naming.md)

En esta sección, no trataremos sobre cuáles son las mejores convenciones de nomenclatura en CSS para mejorar el mantenimiento y la escalabilidad; esto no solo depende de tí, sino que también está fuera del alcance de una guía de estilo. Sugiero las recomendaciones que aparecen en la guía de estilo de CSS - En inglés *(http://cssguidelin.es/#naming-conventions)*.

Hay algunas cosas a las que se les puede asignar un nombre en Sass, y es importante que tengan un nombre adecuado, así todo tu código será coherente y fácil de leer:

- variables;
- funciones;
- mixins.

Los *placeholders* han sido omitidos deliberadamente de esta lista ya que pueden ser considerados como selectores regulares CSS y por lo tanto, se

sigue el mismo patrón de nomenclatura que se utiliza con las clases.

En cuanto a las variables, las funciones y los *mixins*, utilizaremos algo muy *CSSable*: **minúsculas-delimitadas-con-guiones** y especialmente, nombres que tengan un significado claro.

```
$vertical-rhythm-baseline: 1.5rem;

@mixin size($width, $height: $width) {
  // …
}

@function opposite-direction($direction) {
  // …
}
```

Constantes

Si resultas ser un desarrollador de *frameworks* o de librerías, puede que te encuentres con variables que no van a ser actualizadas bajo ninguna circunstancia: las constantes. Desafortunadamente (o ¿afortunadamente?), Sass no proporciona ninguna forma para definir este tipo de entidades, por lo que tenemos que ser muy estrictos con las nomenclaturas para mantener nuestro objetivo.

Como con muchos lenguajes, sugiero que se utilice la opción todo-mayúsculas cuando se trata de constantes. No solo es una convención muy antigua, sino que también contrasta bien con las típicas variables minúsculas separadas con guión.

```
// Yep
$CSS_POSITIONS: (top, right, bottom, left, center);

// Nope
$css-positions: (top, right, bottom, left, center);
```

Si realmente quieres jugar con la idea de las constantes en Sass, deberías leer este dedicado artículo *(http://www.sitepoint.com/dealing-constants-sass/)*.

Espacio De Nombres

Si tienes la intención de distribuir tu código Sass, como por ejemplo, en el caso de una librería, un *framework*, un sistema de retícula o lo que sea, es posible que quieras considerar crear un espacio de nombres (*namespace*) para tus variables, funciones, *mixins* y *placeholders* para que no entren en conflicto con el código de ninguna otra persona.

Por ejemplo, si trabajas en un proyecto llamado *Sassy Unicorn* que está pensado para ser distribuido, podrías considerar utilizar `su-` como espacio de nombres (*namespace*). Es lo suficientemente específico para evitar cualquier colisión de nombres y lo suficientemente corto como para no ser un martirio a la hora de escribirlo.

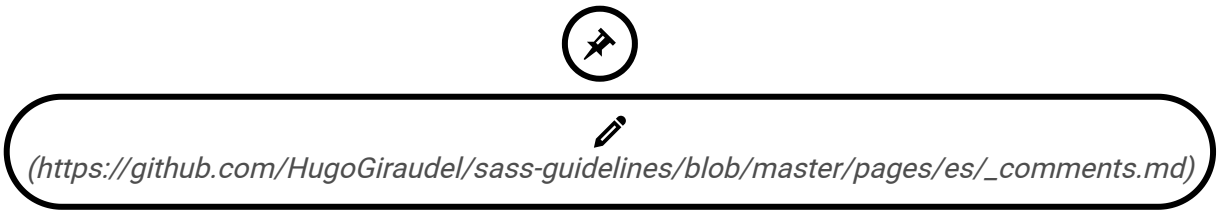
```
$su-configuration: ( … );

@function su-rainbow($unicorn) {
  // …
}
```

Kaelig *(http://kaelig.fr)* tiene un artículo muy revelador acerca del espacio de nombres global de CSS -En inglés *(http://blog.kaelig.fr/post/44554267597/please-respect-the-global-css-namespace)*, en caso de que este tema te interese.

Nota — Ten en cuenta que los espacios de nombres automáticos son sin duda un objetivo de diseño para `@import` en la nueva versión de Sass 4.0. Cuanto más se aproxima la solución, se volverá cada vez menos útil hacerlo de manera manual; eventualmente, las librerías nombradas manualmente pueden ser más difíciles de utilizar.

Comentarios



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_comments.md)

CSS es un lenguajes complicado, lleno de *hacks* y rarezas. Debido a esto, debería de tener muchos comentarios, especialmente si tú o alguien más tiene la intención de leer y actualizar el código dentro de 6 meses o 1 año. No dejes que ni tú, ni nadie se encuentre en la situación de: *yo-no-escribí-esto-oh-dios-mio-por-qué*.

Tan simple como pueda resultar CSS, aún se pueden escribir muchos comentarios. Estos podrían explicar:

- la estructura y/o la función de un archivo;
- el objetivo de un conjunto de reglas;
- la idea detrás de un número mágico;
- la razón detrás de una determinada declaración CSS;
- el orden de las declaraciones CSS;
- el proceso de pensamiento detrás de una manera de hacer las cosas.

Y probablemente haya olvidado muchas otras razones para realizar comentarios. Comentar lleva muy poco tiempo cuando se realiza a la vez que escribes el código, así que hazlo en el momento correcto. Volver atrás y comentar un trozo de código antiguo, no solo es completamente irreal, sino que también es extremadamente molesto.

Escribiendo Comentarios

Idealmente, *cualquier* conjunto de reglas CSS debería ir precedida por un comentario estilo-C explicando el objetivo del bloque CSS. Este comentario también debe dar una explicación numerada respecto a partes específicas del conjunto de reglas. Por ejemplo:

```
/**
 * Clase que corta y añade puntos suspensivos para que
 * en una sola línea
 * 1. Forzar a que el contenido quepa en una sola líne
 * 2. Añadir puntos supensivos al final de la línea.
 */
.ellipsis {
  white-space: nowrap; /* 1 */
  text-overflow: ellipsis; /* 2 */
  overflow: hidden;
}
```

Básicamente, todo lo que no es evidente a primera vista debería de comentarse. No existe tal cosa como demasiada documentación. Recuerda que no se puede *comentar demasiado*, así que ponte manos a la obra y escribe comentarios para todo lo que merezca la pena.

Cuando se comenta una sección específica de Sass, utiliza los comentarios de línea de Sass en lugar de los bloques estilo-C. Esto hace que el comentario sea invisible en la salida, incluso en modo expandido durante el desarrollo.

```
// Añadir el módulo actual a la lista de módulos impor
// se requiere un indicador `!global` para que pueda a
$imported-modules: append($imported-modules, $module)
```

Ten en cuenta que esta forma de comentar el código, es compatible con la guía de CSS en su sección de comentarios *(http://cssguidelin.es/#commenting)*.

Documentación

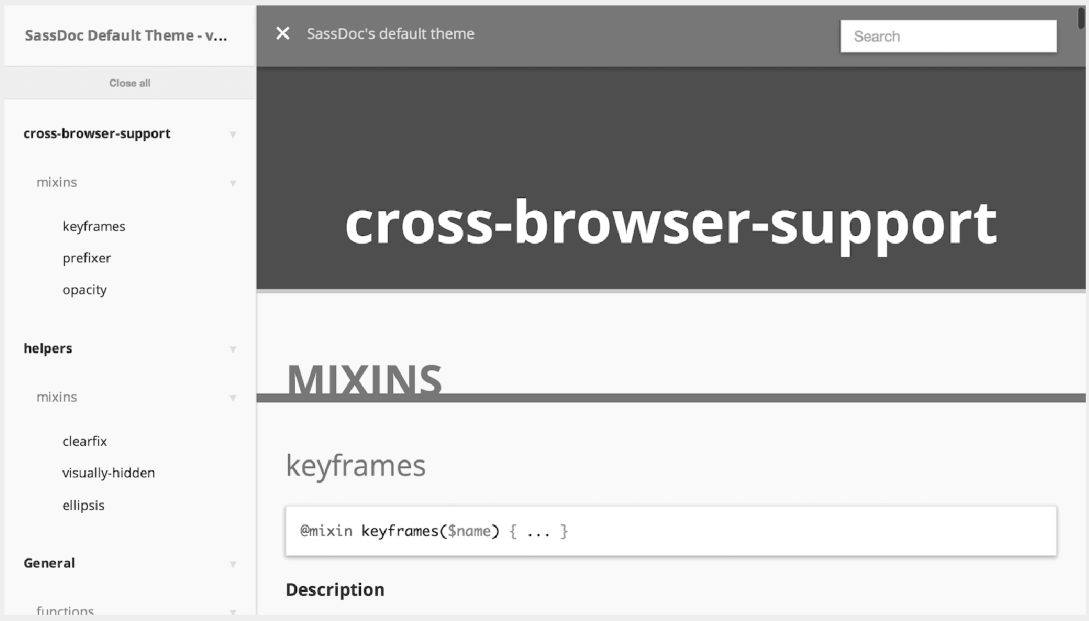
Cada variable, función, *mixin* y *placeholder* que tiene como objetivo ser reutilizado en todo el código, debería estar documentado como parte de la API global usando SassDoc *(http://sassdoc.com)*.

```
/// Ritmo vertical de la línea base que se utiliza en
/// @type Length
$vertical-rhythm-baseline: 1.5rem;
```

Nota — Se requieren tres barras diagonales (`/`).

SassDoc tiene dos funciones principales:

- forzar el uso de comentarios estandarizados basados en un sistema de anotación para todo lo que es parte de una API pública o privada;
- ser capaz de generar una versión HTML de la documentación de la API utilizando cualquiera de los *endpoints* de SassDoc (CLI tool, Grunt, Gulp, Broccoli, Node…).



Documentación generada por SassDoc

Este es un ejemplo de un *mixin* ampliamente documentado con SassDoc:

```
/// Mixin que ayuda a definir tanto `width` como `heig
///
/// @author Hugo Giraudel
///
/// @access public
///
/// @param {Length} $width - `width` del elemento
/// @param {Length} $height [$width] - `height` del el
///
/// @example scss - Usage
///   .foo {
///     @include size(10em);
///   }
///
///   .bar {
///     @include size(100%, 10em);
///   }
///
/// @example css - CSS output
///   .foo {
///     width: 10em;
///     height: 10em;
///   }
///
///   .bar {
///     width: 100%;
///     height: 10em;
///   }
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
```

Arquitectura

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_architecture.md)

Establecer la arquitectura CSS es probablemente una de las cosas más difíciles que tendrás que hacer en la vida de un proyecto. Mantener esa arquitectura consistente y significativa es aún más difícil.

Afortunadamente, uno de los principales beneficios de utilizar un preprocesador CSS es tener la capacidad de dividir el código en varios archivos sin afectar al rendimiento (como haría la directiva `@import` en CSS). Gracias a la directiva `@import` de Sass, es perfectamente seguro (y de hecho recomendable) usar tantos archivos como sean necesarios en el desarrollo, compilándolo todo en una sola hoja de estilo cuando vaya a producción.

Además, también quiero hacer hincapié en la necesidad de utilizar carpetas, incluso para los proyectos a pequeña escala. En casa, no guardas todos tus documentos en el mismo cajón sin más. Utilizas carpetas; una para la casa/apartamento, otra para el banco, otra para las facturas y así sucesivamente. No hay razón para hacer lo contrario cuando se estructura un proyecto CSS. Divide el código en carpetas significativas para que sea sencillo encontrar las cosas más tarde cuando tengas que volver al código de nuevo.

Hay una gran cantidad de arquitecturas populares - En inglés *(http://www.sitepoint.com/look-different-sass-architectures/)* para los proyectos CSS: OOCSS *(http://oocss.org/)*, Atomic Design *(http://bradfrost.com/blog/post/atomic-web-design/)*, Bootstrap *(http://getbootstrap.com/)*, Foundation *(http://foundation.zurb.com/)* o similare. Todas ellas tienen sus méritos, pros y contras.

Yo utilizo un método que resulta ser bastante similar a SMACSS *(https://smacss.com/)* de Jonathan Snook *(http://snook.ca/)*, que se centra en mantener las cosas simples y evidentes.

Nota — He aprendido que la arquitectura es, en la mayoría de los casos muy específica para cada proyecto. Siéntete libre de descartar o de

adaptar la solución propuesta hasta que encuentres un sistema que se adapte a tus necesidades.

Componentes

Hay una gran diferencia entre hacer que algo *funcione* y hacerlo *bien*. De nuevo, CSS es un lenguaje bastante desordenado [cita requerida]. Cuánto menos CSS tengamos, mejor. No queremos tener que trabajar con megabytes de código CSS. Para mantener las hojas de estilo cortas y eficientes —y esto no será ninguna sorpresa para tí— es una buena idea pensar en una interfaz como en una colección de componentes.

Los componentes pueden ser cualquier cosa, siempre y cuando:

- haga una cosa y sólo una cosa;
- sea reutilizable y se reutilice a lo largo del proyecto;
- sea independiente.

Por ejemplo, un formulario de búsqueda debería ser tratado como un componente. Debería ser reutilizable, en diferentes lugares, en diferentes páginas y en varias situaciones. No debe depender de su posición en el DOM (pie de página, barra lateral, contenido principal…).

La mayor parte de cualquier interfaz puede concebirse en forma de pequeños componentes y te recomiendo encarecidamente que lo hagas. Esto no solo reducirá la cantidad de CSS necesario para todo el proyecto, sino que también resultará más fácil de mantener que un desorden caótico donde todo está hecho un lío.

Estructura De Un Componente

Idealmente, los componentes deberían existir dentro de su propia partición Sass (en la carpeta `components/`, como se describe en el patrón 7-1), por ejemplo `components/_button.scss`. Los estilos descritos en cada archivo de componentes sólo deben estar relacionados con:

- el propio estilo del componente en sí;
- el estilo de las variantes, modificadores, y/o estados del componente;
- el estilo de los descendientes del componente (por ejemplo, los hijos), si es necesario.

Si quieres que tus componentes puedan ser personalizados externamente (por ejemplo, desde un tema de la carpeta `themes/`), limita las declaraciones a estilos estructurales, como dimensiones (width/height), padding, margins, alignment, entre otros. Evita los estilos del tipo color, sombra, tipografía, fondo, etc.

Un componente puede incluir variables específicas de componentes, *placeholders* e incluso *mixins* y funciones. Ten en cuenta, sin embargo, que debes evitar referenciar (es decir, `@import`-ar) archivos de componentes de otros archivos de componentes, ya que esto puede hacer que la dependencia dentro de tu proyecto sea un completo lío.

Este es un ejemplo del componente de un botón:

```
// Button-specific variables
$button-color: $secondary-color;

// … include any button-specific:
// - mixins
// - placeholders
// - functions

/**
 * Buttons
 */
.button {
  @include vertical-rhythm;
  display: block;
  padding: 1rem;
  color: $button-color;
  // … etc.

  /**
   * Inlined buttons on large screens
   */
  @include respond-to('medium') {
    display: inline-block;
  }
}

/**
 * Icons within buttons
 */
.button > svg {
  fill: currentcolor;
  // … etc.
}

/**
 * Inline button
 */
.button--inline {
  display: inline-block;
}
```

Nota — Gracias a David Khourshid *(https://twitter.com/davidkpiano)* por su ayuda y experiencia en esta sección.

El Patron 7-1

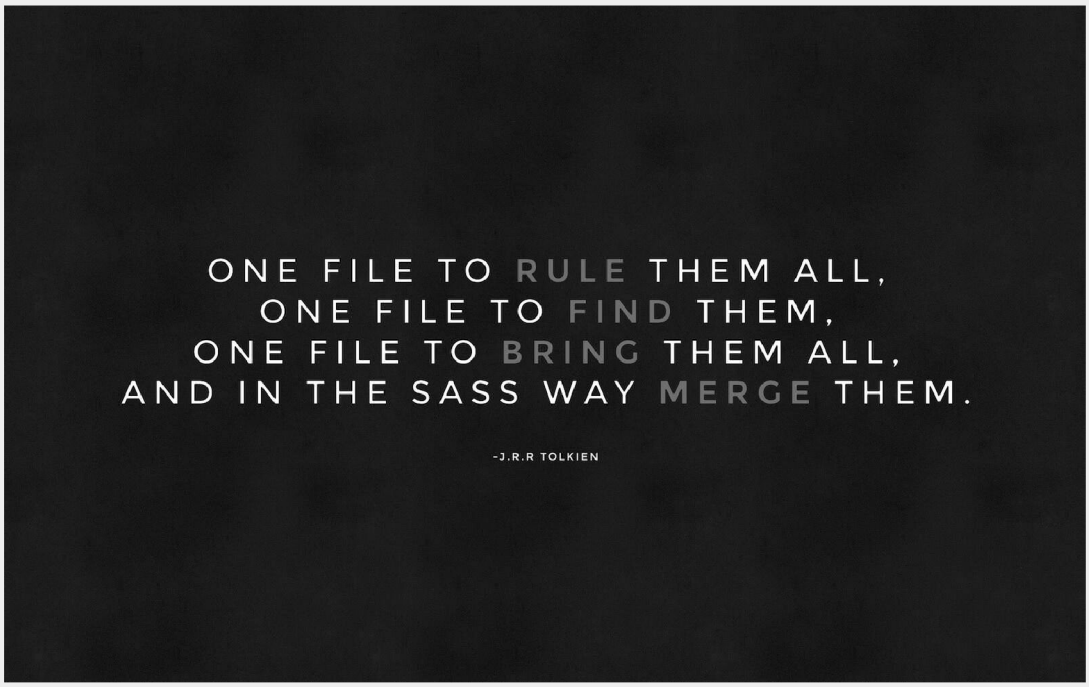
Volvamos a la arquitectura, ¿de acuerdo? Normalmente suelo trabajar con lo que yo llamo el *patrón 7-1*: 7 carpetas, 1 archivo. Basicamente, tienes todas las partes almacenadas en 7 carpetas diferentes, y un único archivo en el directorio raíz (normalmente llamado `main.scss`) y que importa todas estas partes para luego compilarlas en una hoja de estilo CSS.

- `base/`
- `components/`
- `layout/`
- `pages/`
- `themes/`
- `abstracts/`
- `vendors/`

Y por supuesto:

- `main.scss`

Nota — Si quieres usar el patrón 7-1, aquí hay una plantilla reutilizable *(https://github.com/HugoGiraudel/sass-boilerplate)* en GitHub. Debería contener todo lo que necesitas para empezar con esta arquitectura.



Fondo de pantalla realizado por Julen He (https://twitter.com/julien_he)

Idealmente, podemos llegar a algo como esto:

```
sass/
|
|- base/
|   |- _reset.scss       # Reset/normalize
|   |- _typography.scss  # Reglas tipográficas
|   …                    # Etc.
|
|- components/
|   |- _buttons.scss     # Botones
|   |- _carousel.scss    # Carousel
|   |- _cover.scss       # Cubierta
|   |- _dropdown.scss    # Dropdown
|   …                    # Etc.
|
|- layout/
|   |- _navigation.scss  # Navegación
|   |- _grid.scss        # Sistema de retícula
|   |- _header.scss      # Encabezamiento
|   |- _footer.scss      # Pie de página
|   |- _sidebar.scss     # Barra lateral
|   |- _forms.scss       # Formularios
|   …                    # Etc.
|
|- pages/
|   |- _home.scss        # Estilos específicos para la
|   |- _contact.scss     # Estilos específicos para la
|   …                    # Etc.
|
|- themes/
|   |- _theme.scss       # Tema por defecto
|   |- _admin.scss       # Tema del administrador
|   …                    # Etc.
|
|- utils/
|   |- _variables.scss   # Variables Sass
|   |- _functions.scss   # Funciones Sass
|   |- _mixins.scss      # Mixins Sass
|   |- _helpers.scss     # Clases & placeholders
|
|- vendors/
```

```
|   |- _bootstrap.scss   # Bootstrap
|   |- _jquery-ui.scss   # jQuery UI
|   …                    # Etc.
|
|
`- main.scss             # Archivo principal de Sass
```

Nota — Los archivos siguen las mismas convenciones de nomenclatura descritas anteriormente: están delimitadas por guiones.

CARPETA BASE

La carpeta `base/` contiene lo que podríamos llamar la plantilla del proyecto. Allí, puedes encontrar el archivo *reset* para reiniciar los estilos CSS, algunas reglas tipográficas y probablemente un archivo CSS que define algunos estilos estándares para los elementos HTML más comunes (y que me gusta llamar `_base.scss`).

- `_base.scss`
- `_reset.scss`
- `_typography.scss`

Nota — Si tu proyecto usa *muchas* animaciones CSS, podrías pensar en añadir un archivo `\_animations.scss` conteniendo las deficiones de `@keyframes` de todas tus animaciones. Si solo las usas esporádicamente, déjalas convivir con los selectores que las usan.

CARPETA LAYOUT

La carpeta `layout/` contiene todo lo que tiene que ver con la distribución del sitio o la aplicación. Esta carpeta puede contener hojas de estilo para las partes principales del sitio (header, footer, navigation, sidebar…), el sistema de retícula o incluso el estilo CSS para los formularios.

- `_grid.scss`
- `_header.scss`

- `_footer.scss`
- `_sidebar.scss`
- `_forms.scss`
- `_navigation.scss`

> **Nota —** La carpeta `layout/` también se puede llamar `partials/`, todo depende de lo que tu prefieras.

CARPETA COMPONENTES

Para componentes más pequeños, existe la carpeta `components/`. Mientras `layout/` tiene una visión *macro* (definiendo la estructura global), `components/` está mucho más centrado en los *widgets*. Contiene todo tipo de módulos específicos como por ejemplo, un *slider*, un *loader*, un *widget*, y básicamente todo lo que esté en esa misma línea. Por lo general hay un montón de archivos en `components/` ya que todo el sitio o la aplicación debería estar compuesto en su mayoría, de pequeños módulos.

- `_media.scss`
- `_carousel.scss`
- `_thumbnails.scss`

> **Nota —** La carpeta `components/` también se puede llamar `modules/`, todo depende de lo que tu prefieras.

CARPETA PÁGINAS

Si tienes estilos específicos para cada página, es mejor ponerlos en una carpeta `pages/`, dentro de un archivo con el nombre de la página. Por ejemplo, es común tener muchos estilos específicos para la página principal, por lo que existe la necesidad de tener un archivo `_home.scss` en la carpeta `pages/`.

- `_home.scss`
- `_contact.scss`

> **Nota —** Dependiendo de tu proceso de implementación, estos archivos podrían llamarse de manera independiente para evitar que sean fusionados con otros en la hoja de estilos resultante. Todo depende de ti.

CARPETA TEMAS

En sitios y aplicaciones grandes, no es raro tener diferentes temas. Es cierto que hay diferentes maneras de tratar con los temas, pero personalmente, me gusta tenerlos todos en la carpeta `themes/`.

- `_theme.scss`
- `_admin.scss`

> **Nota —** Esto es muy específico del proyecto y es probable que sea inexistente en muchos de ellos.

CARPETA ABSTRACTS

La carpeta `abstracts/` reúne todas las herramientas y *helpers* de Sass utilizados en todo el proyecto. Cada variable global, función, *mixin* y *placeholder* debería estar en esta carpeta.

La regla de oro para esta carpeta es que no debe generar ni una sola línea de CSS cuando se compila por si sola. Solo hay *helpers* de Sass.

- `_variables.scss`
- `_mixins.scss`
- `_functions.scss`
- `_placeholders.scss`

Cuando se trabaja en un proyecto muy grande, con muchas utilidades abstractas, podría ser interesante agruparlas por tema en vez de por tipo, por ejemplo tipografía (`_typography.scss`), temas (`_theming.scss`), etc. Cada archivo contiene todos los *helpers* relacionados: variables, funciones, *mixins*

y *placeholders*. Hacerlo provoca que que el código sea más fácil de navegar y mantener, especialmente cuando los archivos son muy largos.

> **Nota —** La carpeta `abstracts/` también se puede llamar `utilities/` o `helpers/`, dependiendo de tus preferencias.

CARPETA VENDORS

Y por último, pero no menos importante, la mayoría de los proyectos tienen una carpeta `vendors/` que contiene todos los archivos CSS procedentes de librerías externas y *frameworks* – Normalize, Bootstrap, jQueryUI, FancyCarouselSliderjQueryPowered, etc. Poner estos archivos en una misma carpeta, es una buena forma de decir "¡Hey! esto no lo he escrito yo, no es mi código y no es mi responsabilidad".

- `_normalize.scss`
- `_bootstrap.scss`
- `_jquery-ui.scss`
- `_select2.scss`

Si tienes que sobrescribir una sección de cualquier *vendor*, te recomiendo que tengas una octava carpeta llamada `vendors-extensions/` en la que puedes poner estos archivos usando el mismo nombre que le ha dado el desarrollador.

Por ejemplo, `vendors-extensions/_bootstrap.scss` es un archivo que contiene todas las reglas CSS que se volverán a declarar con respecto al CSS por defecto de Bootstrap. Esto se hace para evitar editar directamente los archivos del proveedor, lo que es en general una mala idea.

ARCHIVO PRINCIPAL

El archivo principal (normalmente llamado `main.scss`) debería ser el único archivo Sass de todo el código que no empieza con guión bajo. Este archivo no debería contener nada más que `@import` y comentarios.

Los archivos deben de importarse según la carpeta que los contiene, una después de la otra en el siguiente orden:

1. `abstracts/`
2. `vendors/`
3. `base/`
4. `layout/`
5. `components/`
6. `pages/`
7. `themes/`

Con el fin de mantener la legibilidad, el archivo principal debe respetar las siguientes pautas:

- un archivo para cada `@import`;
- un `@import` por línea;
- no dejar una línea en blanco entre dos archivos que pertenecen a la misma carpeta;
- dejar una línea en blanco después del último archivo de una carpeta;
- las extensiones de archivo y los guiones principales se omiten.

```
@import 'abstracts/variables';
@import 'abstracts/functions';
@import 'abstracts/mixins';
@import 'abstracts/placeholders';

@import 'vendors/bootstrap';
@import 'vendors/jquery-ui';

@import 'base/reset';
@import 'base/typography';

@import 'layout/navigation';
@import 'layout/grid';
@import 'layout/header';
@import 'layout/footer';
@import 'layout/sidebar';
@import 'layout/forms';

@import 'components/buttons';
@import 'components/carousel';
@import 'components/cover';
@import 'components/dropdown';

@import 'pages/home';
@import 'pages/contact';

@import 'themes/theme';
@import 'themes/admin';
```

Hay otra forma de importar las partes de un proyecto que también que me parece válida. La parte positiva, es que hace que el fichero sea más legible. Pero por otro lado, actualizar el contenido puede ser un poco más doloroso. De todas formas, te dejaré decidir qué es lo mejor, en realidad no importa demasiado. Para hacer las cosas de esta manera, el archivo principal debe respetar las siguientes pautas:

- un `@import` por carpeta;
- un salto de línea después de cada `@import`;
- cada archivo debe ir en su propia línea;

- dejar una línea en blanco después del último archivo de una carpeta;
- las extensiones de archivo y los guiones principales se omiten.

```
@import
  'abstracts/variables',
  'abstracts/functions',
  'abstracts/mixins',
  'abstracts/placeholders';

@import
  'vendors/bootstrap',
  'vendors/jquery-ui';

@import
  'base/reset',
  'base/typography';

@import
  'layout/navigation',
  'layout/grid',
  'layout/header',
  'layout/footer',
  'layout/sidebar',
  'layout/forms';

@import
  'components/buttons',
  'components/carousel',
  'components/cover',
  'components/dropdown';

@import
  'pages/home',
  'pages/contact';

@import
  'themes/theme',
  'themes/admin';
```

Acerca De Globbing

En programación, los patrones glob especifican un conjunto de nombres de fichero con caracteres comodín, como `*.scss`. En general, *globbing* busca hacer coincidir un conjunto de archivos basado en una expresión, en lugar de en una lista de nombres de archivo. Cuando esto se aplica a Sass, significa importar *partials* en el archivo main con un patrón glob en lugar de enumerarlos individualmente. Esto hará que el archivo principal tenga este aspecto:

```
@import 'abstracts/*';
@import 'vendors/*';
@import 'base/*';
@import 'layout/*';
@import 'components/*';
@import 'pages/*';
@import 'themes/*';
```

Sass no soporta el uso de patrones *glob* porque puede ser una característica peligrosa ya que sabes que en CSS es importante el orden. Al importar los archivos de manera dinámica (que suele ser en orden alfabético), no se controla el orden de los ficheros, lo que puede provocar efectos secundarios muy difíciles de depurar.

Dicho esto, en una estricta arquitectura basada en componentes, en la que se tiene especial cuidado de no filtrar ningún estilo de un *partial* a otro, el orden no debería ser importante, lo que permitiría el uso del patrón *glob* de importación. Esto significaría que es más fácil agregar o quitar *partials*, puesto que actualizar el archivo *main* no sería necesario.

Cuando se usa Ruby Sass, hay una gema de Ruby llamada sass-globbing *(https://github.com/chriseppstein/sass-globbing)* que permite tener este mismo comportamiento. Si se utiliza node-sass, se puede confiar en Node.js o en cualquier herramienta que se use para realizar la compilación (Gulp, Grunt, etc.).

El Archivo De La Verguenza

Hay un concepto interesante que ha popularizado Harry Roberts *(http://css-wizardry.com)*, Dave Rupert *(http://daverupert.com)* y Chris Coyier *(http://css-tricks.com)* y que consiste en poner todas las declaraciones CSS, *hacks* y cosas de las que no nos sentimos muy orgullosos en un archivo de la verguenza *(http://csswizardry.com/2013/04/shame-css-full-net-interview/)*. Este archivo, titulado dramáticamente `_shame.scss`, se importará después de los demás archivos, al final de la hoja de estilo.

```
/**
 * Arreglo específico para la navegación
 *
 * Alguien utilizó un ID en el código del *header* (`#
 * los selectores nav (`.site-nav a {}`). Usar !import
 * tenga tiempo de arreglarlo.
 */
.site-nav a {
    color: #BADA55 !important;
}
```

Responsive Web Design Y Puntos De Ruptura

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_rwd.md)

No creo que sea necesario tener que explicar lo que es el Responsive Web Design ahora que está en todas partes. Sin embargo es posible que te preguntes *¿por qué hay una sección sobre RWD en una guía de estilo de Sass?* En realidad hay algunas cuantas cosas que se puedan hacer para facilitar el trabajo con los puntos de ruptura o *breakpoints*, así que he pensado que no estaría mal listarlas aquí.

Nombrando Los Puntos De Ruptura

Creo que puedo decir con seguridad que las *media queries* no deberían estar vinculadas a dispositivos específicos. Por ejemplo, es una mala idea intentar trabajar con tamaños específicos para iPads o Blackberry. Las *media queries* deberían trabajar con un rango de tamaños de pantalla, justo hasta que el diseño se rompa y la siguiente *media query* haga su función.

Por las mismas razones, los puntos de ruptura no deberían llevar el nombre de los dispositivos, sino algo más general. Sobre todo porque algunos teléfonos son ahora más grandes que algunas tablets, algunas tablets más grandes que un ordenador pequeño y así sucesivamente…

```
// Yep
$breakpoints: (
  'medium': (min-width: 800px),
  'large': (min-width: 1000px),
  'huge': (min-width: 1200px),
);

// Nope
$breakpoints: (
  'tablet': (min-width: 800px),
  'computer': (min-width: 1000px),
  'tv': (min-width: 1200px),
);
```

Llegados a este punto, cualquier nomenclatura *(http://css-tricks.com/naming-media-queries/)* que deje claro que el diseño no está ligado a un dispositivo en concreto podría funcionar, siempre y cuando tenga un sentido de magnitud.

```
$breakpoints: (
  'seed': (min-width: 800px),
  'sprout': (min-width: 1000px),
  'plant': (min-width: 1200px),
);
```

Nota — Los ejemplos anteriores utilizan mapas anidados para definir los puntos de ruptura, sin embargo, esto depende de qué tipo de gestor de *breakpoints* utilices. Puedes optar por cadenas en lugar de mapas para una mayor flexibilidad (por ejemplo `'(min-width: 800px)'`).

Gestor De Puntos De Ruptura

Una vez que tus *breakpoints* tengan la nomenclatura deseada, necesitas una manera de utilizarlos en las *media queries* reales. Hay un montón de maneras para hacerlo, pero tengo que decir que soy un gran fan del mapa de puntos de ruptura (*breakpoint map*) leído por una función *getter*. Este sistema es tan simple como eficiente.

```
/// Gestor Responsive
/// @access public
/// @param {String} $breakpoint - Punto de ruptura
/// @requires $breakpoints
@mixin respond-to($breakpoint) {
  $raw-query: map-get($breakpoints, $breakpoint);

  @if $raw-query {
    $query: if(
      type-of($raw-query) == 'string',
      unquote($raw-query),
      inspect($raw-query)
    );

    @media #{$query} {
      @content;
    }
  } @else {
    @error 'No se ha encontrado un valor para `#{$brea
         + 'Por favor, asegúrate que está definido en
  }
}
```

Nota — Obviamente, este es un gestor de puntos de ruptura bastante simplista. Si necesitas un gestor ligeramente más permisivo, te recomiendo que no reinventes la rueda y utilices algo que ya esté probado y comprobado, como por ejemplo Sass-MQ *(https://github.-com/sass-mq/sass-mq)*, Breakpoint *(http://breakpoint-sass.com/)* o include-media *(https://github.com/eduardoboucas/include-media)*.

> Si estás buscando más información acerca de cómo enfocar las Media Queries en Sass, tanto SitePoint *(http://www.sitepoint.com/managing-responsive-breakpoints-sass/)* (de este servidor) como CSS-Tricks *(http://css-tricks.com/approaches-media-queries-sass/)* tienen muy buenos artículos al respecto.

Uso De Media Queries

No hace mucho tiempo, hubo un debate bastante acalorado acerca de dónde deberían estar las *medias queries*: ¿deberían estar dentro de los selectores (permitido por Sass) o deberían estar completamente separados de ellos? Debo decir que soy un ferviente defensor del sistema *media queries dentro del selector*, ya que creo que funciona bastante bien con la idea de *componentes*.

```
.foo {
  color: red;

  @include respond-to('medium') {
    color: blue;
  }
}
```

Resultaría el siguiente bloque CSS:

```
.foo {
  color: red;
}

@media (min-width: 800px) {
  .foo {
    color: blue;
  }
}
```

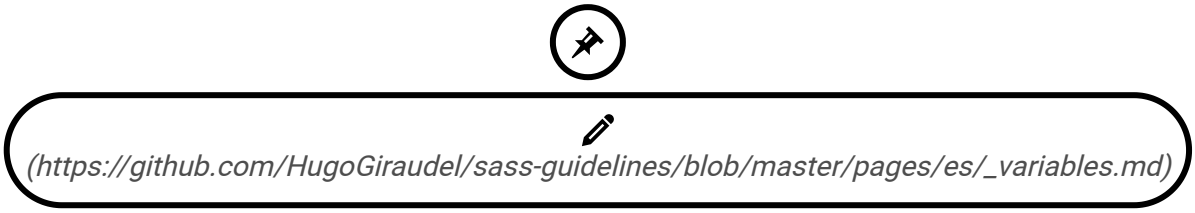
Es posible que escuches que esta convención dará como resultado en CSS bloques duplicados de *media queries*. Esto es definitivamente cierto. Sin embargo, se han realizado pruebas

(http://sasscast.tumblr.com/post/38673939456/sass-and-media-queries) y la conclusión es que dará igual una vez Gzip (o cualquier equivalente) haya hecho su trabajo:

> *… hemos discutido a fondo si hay consecuencias en el rendimiento, en cuanto a la combinación frente a la dispersión de Media Queries y se llegó a la conclusión de que la diferencia, aunque fea, es en el peor de los casos mínima y esencialmente inexistente en el mejor.*
> *— Sam Richards, en relación a los puntos de ruptura (http://sasscast.-tumblr.com/post/38673939456/sass-and-media-queries)*

Si realmente te preocupan las *media queries* duplicadas, puedes usar una herramienta para fusionarlas como por ejemplo esta gema *(https://github.-com/aaronjensen/sass-media_query_combiner)* sin embargo, siento que debo advertirte de los posibles efectos secundarios ocasionados al mover el código de lugar. Ya sabes que el orden del código es importante.

Variables



Las variables son la esencia de cualquier lenguaje de programación. Nos permiten reutilizar valores sin tener que copiarlos una y otra vez. Y lo más importante, permiten actualizar cualquier valor de manera sencilla. No más buscar y reemplazar valores de manera manual.

Sin embargo CSS no es más que una cesta enorme que contiene todos nuestros elementos. A diferencia de muchos lenguajes, no hay *scopes* reales en CSS. Debido a esto, debemos prestar especial atención cuando añadimos variables ya que pueden existir conflictos.

Mi consejo sería que solo crearas variables cuando tenga sentido hacerlo. No inicies nuevas variables solo por el gusto de hacerlo, no será de ayuda. Una nueva variable debe crearse solo cuando se cumplen los siguientes criterios:

- el valor se repite al menos dos veces;
- es probable que el valor se actualice al menos una vez;
- todas las ocurrencias del valor están vinculadas a la variable (es decir, no por casualidad).

Básicamente, no hay ningún sentido en declarar una variable que nunca se actualizará o que sólo se usará en un solo lugar.

Scoping

En Sass, el ámbito (*scoping*) de las variables ha cambiado a lo largo de los años. Hasta hace muy poco, las declaraciones de variables dentro de los conjuntos de reglas y otros ámbitos eran locales por defecto. Sin embargo cuando ya había una variable global con el mismo nombre, la asignación local cambiaría dicha variable global. Desde la versión 3.4, Sass aborda correctamente el concepto de ámbitos y crea una nueva variable local en su lugar.

Los documentos hablan de *ocultar o sombrear la variable global*. Cuando se declara una variable que ya existe en el marco global dentro de un ámbito interno (selector, función, *mixin*…), se dice que la variable local esta *sombreando* a la variable global. Básicamente, la sobrescribe solo en el ámbito local.

El siguiente fragmento de código explica el concepto de *sombreado de variable*:

```
// Inicializar una variable global a nivel raiz.
$variable: 'valor inicial';

// Crear un *mixin* que sobrescribe la variable global
@mixin global-variable-overriding {
  $variable: 'mixin value' !global;
}

.local-scope::before {
  // Crear una variable local que oculte la variable g
  $variable: 'local value';

  // Incluir el *mixin*: sobrescribe la variable globa
  @include global-variable-overriding;

  // Imprimir el valor de la variable.
  // Es la variable **local** puesto que sobrescribe l
  content: $variable;
}

// Imprime la variable en otro selector que no la está
// Es la variable **global**, como se esperaba.
.other-local-scope::before {
  content: $variable;
}
```

El Flag `!default`

Cuando se construye una librería, un *framework*, un sistema de retícula o cualquier bloque de Sass que está destinado a ser distribuido y usado por desarrolladores externos, todas las variables de configuración deben estar definidas con el flag `!default` para que puedan sobrescribirse.

```
$baseline: 1em !default;
```

Gracias a esto, cualquier desarrollador puede definir su propia variable `$baseline` *antes* de importar tu librería sin que su valor sea redefinido.

```
// La variable del desarrollador
$baseline: 2em;

// Tu librería declarando la variable `$baseline`
@import 'your-library';

// $baseline == 2em;
```

El Flag `!global`

El flag `!global` solo se debe utilizar cuando se sobrescribe una variable global desde un marco local. Cuando se define una variable a nivel raiz, el flag `!global` debe ser omitido.

```
// Yep
$baseline: 2em;

// Nope
$baseline: 2em !global;
```

Variables Múltiples O Mapas

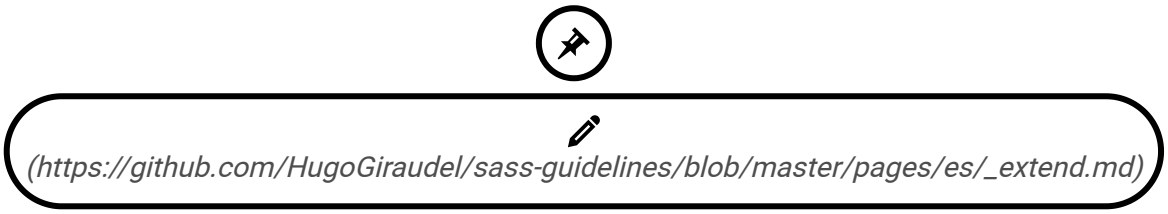
Existen varias ventajas al utilizar mapas en lugar de variables múltiples. La principal de ellas es la de poder iterar sobre un mapa, lo que no es posible con múltiples variables.

Otra ventaja de utilizar un mapa es la de tener la capacidad de crear una pequeña función *getter* para proporcionar una API más amigable. Por ejemplo, echa un vistazo al siguiente código Sass:

```
/// Mapa de Z-indexes, reuniendo todos las capas Z de
/// @access private
/// @type Map
/// @prop {String} key - Nombre de la capa
/// @prop {Number} value - Valor Z asignada a la clave
$z-indexes: (
  'modal': 5000,
  'dropdown': 4000,
  'default': 1,
  'below': -1,
);

/// Obtener todos los z-index a partir del nombre de u
/// @access public
/// @param {String} $layer - Nombre de la capa
/// @return {Number}
/// @require $z-indexes
@function z($layer) {
  @return map-get($z-indexes, $layer);
}
```

Extend



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_extend.md)

La directiva `@extend` es una característica muy potente que a menudo es malentendida. En general, hace posible decirle a Sass que use el estilo del selector A como si también correspondiera con el selector B. No hace falta decir, que puede ser un valioso aliado al escribir CSS modular.

Si embargo, el verdadero propósito de `@extend` es mantener las relaciones (restricciones) dentro de los selectores extendidos entre conjuntos de reglas. ¿Qué significa esto exactamente?

- Los selectores tienen *restricciones* (por ejemplo, `.bar` en `.foo > .bar` debe tener un padre `.foo`).
- Estas restricciones se *transfieren* al selector que se extiende (por ejemplo, `.baz { @extend .bar; }` producirá `.foo > .bar, .foo > .baz`);
- Las declaraciones del selector extendido serán compartidas con el selector que se está extendiendo.

Se puede ver con claridad que al extender selectores con restricciones indulgentes puede conducir a una explosión del selector. Si `.baz .qux` extiende `.foo .bar`, el selector resultante puede ser `.foo .baz .qux` o `.baz .foo .qux`, ya que tanto `.foo` como `.baz` son antepasados generales. Pueden ser padres, abuelos, etc.

Intenta definir las relaciones vía placeholders *(http://www.sitepoint.com/sass-reference/placeholders/)* y no con selectores directos. Esto te dará la libertad de usar (y cambiar) cualquier convención de nomenclatura que tengas para tus selectores, y ya que las relaciones solo se definen una vez dentro de los *placeholders*, es mucho menos probable tener selectores no desados.

Para heredar estilos, usa sólo `@extend` si el selector que se extiende `.class` o `%placeholder` *es un tipo del* selector extendido. Por ejemplo, un `.error` es un tipo de `.warning`, por lo que `.error` puede `@extend .warning`.

```
%button {
  display: inline-block;
  // … button styles

  // Relationship: a %button that is a child of a %moc
  %modal > & {
    display: block;
  }
}

.button {
  @extend %button;
}

// Yep
.modal {
  @extend %modal;
}

// Nope
.modal {
  @extend %modal;

  > .button {
    @extend %button;
  }
}
```

Hay muchos escenarios donde extender selectores pueden ser de gran ayuda e incluso, merece la pena. Sin embargo, ten en cuenta las siguientes reglas para `@extend`-er correctamente.

- Usa *extend* principalmente en `%placeholders`, no en selectores.
- Cuando extiendas clases, extiende una clase con otra, *nunca* con un selector complejo *(http://www.w3.org/TR/selectors4/#syntax)*.
- Intenta no extender directamente los `%placeholder`, hazlo tan pocas veces como sea posible.
- Evita extender los ancestros generales (por ejemplo, `.foo .bar`) o los selectores generales adyacentes (por ejemplo `.foo ~ .bar`). Esto es lo que

provoca la explosión del selector.

> **Nota —** A menudo se dice que `@extend` ayuda a disminuir el tamaño del archivo, ya que combina los selectores en lugar de duplicar las propiedades. Eso es cierto, sin embargo la diferencia es insignificante una vez que Gzip *(http://en.wikipedia.org/wiki/Gzip)* ha comprimido el archivo.
>
> Dicho esto, si no puedes usar Gzip (o cualquier equivalente), cambiar a un enfoque `@extend` puede resultar valioso, especialmente si el peso de tu hoja de estilo es el cuello de botella en el rendimiento de tu proyecto.

Extend Y Media Queries

Sólo debes extender los selectores dentro del mismo ámbito (directiva `@media`). Piensa en una media query como en otra restricción.

```
%foo {
  content: 'foo';
}

// Nope
@media print {
  .bar {
    // Esto no funciona. Peor: se bloquea
    @extend %foo;
  }
}

// Yep
@media print {
  .bar {
    @at-root (without: media) {
      @extend %foo;
    }
  }
}

// Yep
%foo {
  content: 'foo';

  &-print {
    @media print {
      content: 'foo print';
    }
  }
}

@media print {
  .bar {
    @extend %foo-print;
  }
}
```

Las opiniones parecen estar extremadamente divididas con respecto a los beneficios y los problemas de `@extend` hasta el punto en el que muchos

desarrolladores, incluido yo mismo, han estado abogando en contra de él, como se puede leer en los siguientes artículos:

- Lo que nadie te dijo acerca de Sass Extend - En inglés *(http://www.sitepoint.com/sass-extend-nobody-told-you/)*
- ¿Por qué debes evitar usar Extend? -En inglés *(http://www.sitepoint.com/avoid-sass-extend/)*
- No te "sobre extiendas" -En inglés *(http://pressupinc.com/blog/2014/11/dont-overextend-yourself-in-sass/)*

Dicho esto, y para resumir, te aconsejaría usar `@extend` solo para mantener la relación con los selectores. Si dos selectores son característicamente similares, es el caso de uso perfecto para `@extend`. Si no tienen ninguna relación, pero comparten algunas reglas, un `@mixin` puede ser la mejor opción. Para saber más sobre cómo elegir entre los dos, éste reportaje *(http://csswizardry.com/2014/11/when-to-use-extend-when-to-use-a-mixin/)* te puede ayudar.

> **Nota —** Gracias a David Khourshid *(https://twitter.com/davidkpiano)* por su ayuda y experiencia en esta sección.

Mixins

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_mixins.md)

Los *mixins* son una de las características más utilizadas dentro de todo el lenguaje Sass. Son la clave para la reutilización y los componentes DRY. Y por una buena razón: los *mixins* permiten a los autores definir estilos CSS que pueden volver a usarse a lo largo de toda la hoja de estilo sin necesidad de recurrir a las clases no semánticas, como por ejemplo `.float-left`.

Pueden contener reglas CSS completas y casi todo lo que se permite en cualquier parte de un documento Sass. Incluso pueden pasarse argumentos, al igual que en las funciones. Sobra decir que las posibilidades son infinitas.

Pero creo que debo advertirte contra el abuso del poder los *mixins*. De nuevo, la clave aquí es la *simplicidad*. Puede ser tentador contruir *mixins* extremadamente poderosos y con grandes cantidades de lógica. Esto se llama exceso de ingeniería y la mayoría de los desarrolladores la padecen. No pienses demasiado tu código y sobre todo haz que sea simple. Si un *mixin* ocupa mas o menos unas 20 líneas, debes dividirlo en partes más pequeñas o revisarlo completamente.

Fundamentos

Dicho esto, los *mixins* son extremadamente útiles y deberías estar usando algunos. La regla de oro es que si detectas un grupo de propiedades CSS que están siempre juntas por alguna razón (es decir, no es una coincidencia), puedes crear un *mixin* en su lugar. Por ejemplo, el hack micro-clearfix de Nicolas Gallagher *(http://nicolasgallagher.com/micro-clearfix-hack/)* merece ser puesto en un *mixin* sin argumentos.

```
/// Ayuda a limpiar floats internos
/// @author Nicolas Gallagher
/// @link http://nicolasgallagher.com/micro-clearfix-h
@mixin clearfix {
  &::after {
    content: '';
    display: table;
    clear: both;
  }
}
```

Otro ejemplo válido sería un *mixin* para para darle tamaño a un elemento, definiendo tanto `width` como `height` al mismo tiempo. No solo hace que el código sea más fácil de escribir, sino que también será más fácil de leer.

```
/// Asigna un tamaño a un elemento
/// @author Hugo Giraudel
/// @param {Length} $width
/// @param {Length} $height
@mixin size($width, $height: $width) {
  width: $width;
  height: $height;
}
```

Para ejemplos más complejos, echa un vistazo a éste mixin para generar triángulos CSS *(http://www.sitepoint.com/sass-mixin-css-triangles/)*, éste mixin para crear sombras largas *(http://www.sitepoint.com/ultimate-long-shadow-sass-mixin/)* o éste mixin para mantener gradientes CSS en navegadores antiguos (polyfill) *(http://www.sitepoint.com/building-linear-gradient-mixin-sass/)*.

Mixins Sin Argumentos

Algunas veces, los mixins se utilizan solo para evitar la repetición del mismo grupo de declaraciones una y otra vez, sin embargo, no necesitan ningun parámetro o tienen los suficientes argumentos por defecto como para tener que pasarle otros adicionales.

En estos casos, podemos omitir sin problema los paréntesis cuando invocamos el mixin. La palabra clave `@include` (o el signo `+` en la sintáxis con tabulación) actúa como un indicador de que la línea es una llamada a un mixin; no hay necesidad de paréntesis adicionales aquí.

```
// Yep
.foo {
  @include center;
}

// Nope
.foo {
  @include center();
}
```

Lista De Argumentos

Cuando se trabaja con un número indeterminado de argumentos en un *mixin*, utiliza siempre una `arglist` (lista de argumentos) en lugar de una lista. Piensa en un `arglist` como en el octavo tipo de dato no documentado y oculto de Sass que se usa implicitamente cuando se pasa un número arbitrario de argumentos a un *mixin* o una a función que contiene `...`.

```
@mixin shadows($shadows...) {
  // type-of($shadows) == 'arglist'
  // …
}
```

Ahora, cuando se contruye un *mixin* que acepta un puñado de argumentos (entiéndase 3 o más), piénsalo dos veces antes de fusionarlos en una lista o mapa pensando que será más sencillo que pasarlos uno a uno.

Sass es de hecho muy inteligente con los *mixins* y las declaraciones de funciones, tanto, que puedes pasarle una lista o un mapa como un *arglist* a una función o *mixin* para que sea tomado como una serie de argumentos.

```
@mixin dummy($a, $b, $c) {
  // …
}

// Yep
@include dummy(true, 42, 'kittens');

// Yep but nope
$params: (true, 42, 'kittens');
$value: dummy(nth($params, 1), nth($params, 2), nth($p

// Yep
$params: (true, 42, 'kittens');
@include dummy($params...);

// Yep
$params: (
  'c': 'kittens',
  'a': true,
  'b': 42,
);
@include dummy($params...);
```

Para más información acerca de cuál es la mejor opción entre usar múltiples argumentos, una lista o una lista de argumentos, SitePoint tiene una buen artículo sobre el tema *(http://www.sitepoint.com/sass-multiple-arguments-lists-or-arglist/)*.

Mixins Y Prefijos De Proveedores

Puede ser tentador definir *mixins* personalizados para manejar prefijos de *vendors* en propiedades CSS que no son compatibles o totalmente soportadas. Pero no queremos esto. Primero, si puedes usar Autoprefixer *(https://github.com/postcss/autoprefixer)*, usa Autoprefixer. Eliminará código Sass de tu proyecto, siempre estará al día y hará un trabajo mejor de lo que lo puedes hacer tu al poner prefijos a los atributos.

Desafortunadamente, Autoprefixer no siempre es una opción. Si usas Bourbon *(http://bourbon.io/)* o Compass *(http://compass-style.org/)*,

seguramente sabrás que ambos proporcinan una coleción de *mixins* que manejan los prefijos de los *vendors* por ti. Úsalos.

Si no puedes usar Autoprefixer, ni Bourbon, ni Compass, entonces y sólo entonces, puedes tener tu propio *mixin* para añadir prefijos a las propiedades CSS. Pero. Por favor, no construyas un *mixin* por propiedad, imprimiendo manualmente cada *vendor*.

```
// Nope
@mixin transform($value) {
  -webkit-transform: $value;
  -moz-transform: $value;
  transform: $value;
}
```

Hazlo de la manera inteligente.

```
/// Mixin para producir prefijos de proveedores
/// @access public
/// @author HugoGiraudel
/// @param {String} $property - propiedad CSS sin pref
/// @param {*} $value - Valor CSS en crudo
/// @param {List} $prefixes - Lista de prefijos a proc
@mixin prefix($property, $value, $prefixes: ()) {
  @each $prefix in $prefixes {
    -#{$prefix}-#{$property}: $value;
  }

  #{$property}: $value;
}
```

Entonces usar este *mixin* debería ser directo y sencillo:

```
.foo {
  @include prefix(transform, rotate(90deg), ('webkit',
}
```

Por favor, ten en cuenta que es una solución pobre. Por ejemplo, no puede trabajar con *polyfills* complejos como los necesarios para Flexbox. En ese sentido, utilizar Autoprefixer sería una solución mucho mejor.

Sentencias Condicionales

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_conditions.md)

Probablemente ya sabes que Sass proporciona instrucciones condicionales a través de las directivas `@if` y `@else`. A menos que tengas algún elemento con una lógica demasiado compleja en tu código, no hay necesidad de tener sentencias condicionales en tus hojas de estilo del día a día. En realidad, se usan principalmente para librerías y *frameworks*.

De todas formas, si algún día te encuentras con la necesidad de utilizarlas, por favor, respeta las siguientes pautas:

- No uses paréntesis a no ser que sea necesario;
- Deja siempre una línea en blanco antes del `@if`;
- Deja siempre un salto de línea después de una llave de apertura (`{`);
- La sentencia `@else` debe ir en la misma línea que la llave de cierre anterior (`}`).
- Deja siempre una línea en blanco después de la última llave de cierre (`}`) a menos que la siguiente línea sea otra llave de cierre (`}`).

```
// Yep
@if $support-legacy {
  // …
} @else {
  // …
}

// Nope
@if ($support-legacy == true) {
  // …
}
@else {
  // …
}
```

Cuando se evalúa un valor booleano falso, utiliza siempre la palabra clave `not` en lugar de evaluar contra `falso` o `null`

```
// Yep
@if not index($list, $item) {
  // …
}

// Nope
@if index($list, $item) == null {
  // …
}
```

Pon siempre la parte de la variable en la parte izquierda de la sentencia y el (in)esperado resultado en la derecha. Las sentencias condicionales invertidas son con frecuencia, más difíciles de leer, especialmente para desarrolladores inexpertos.

```
// Yep
@if $value == 42 {
  // …
}

// Nope
@if 42 == $value {
  // …
}
```

Cuando se utilizan sentencias condicionales dentro de una función para devolver un resultado diferente basado en alguna condición, siempre debes de asegurarte que la función tiene una declaracón `@return` fuera de cualquier bloque condicional.

```
// Yep
@function dummy($condition) {
  @if $condition {
    @return true;
  }

  @return false;
}

// Nope
@function dummy($condition) {
  @if $condition {
    @return true;
  } @else {
    @return false;
  }
}
```

Bucles

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_loops.md)

Puesto que Sass proporciona estructuras de datos complejas como por ejemplo listas y mapas, no es de extrañar que también proporcione una forma para que los autores puedan iterar sobre dichas entidades.

Sin embargo, la presencia de bucles generalmente implica una lógica moderadamente compleja que propablemente no pertenece a Sass. Antes de utilizar un bucle, asegúrate de que tiene sentido y que de hecho resuelve un problema.

Each

El bucle `@each` es definitivamente el más utilizado de los tres tipos de bucle que proporciona Sass. Sass ofrece una API limpia para iterar sobre una lista o mapa.

```
@each $theme in $themes {
  .section-#{$theme} {
    background-color: map-get($colors, $theme);
  }
}
```

Cuando se itera sobre un mapa utiliza siempre `$key` y `$value` como nombres de variables para mantener una coherencia.

```
@each $key, $value in $map {
  .section-#{$key} {
    background-color: $value;
  }
}
```

También asegúrate de respetar estas pautas para preservar la legibilidad:

- Deja siempre una línea en blanco antes del `@each`;

- Deja siempre una línea en blanco después de la llave de cierre (`}`) a no ser que la siguiente línea sea otra llave de cierre (`}`).

For

El bucle `@for` puede ser útil cuando se combina con las pseudo-clases CSS `:nth-*`. A excepción de estos escenarios, es preferible usar un bucle `@each` si *tienes que* iterar sobre algo.

```
@for $i from 1 through 10 {
  .foo:nth-of-type(#{$i}) {
    border-color: hsl($i * 36, 50%, 50%);
  }
}
```

Utiliza siempre `$i` como nombre de variable para mantener la convención habitual, a menos que tengas una muy buena razón para no hacerlo, no uses la palabra clave `to`, es mejor usar siempre `through`. Muchos desarrolladores no saben que Sass ofrece esta variación; usarla podría llevar a confusión.

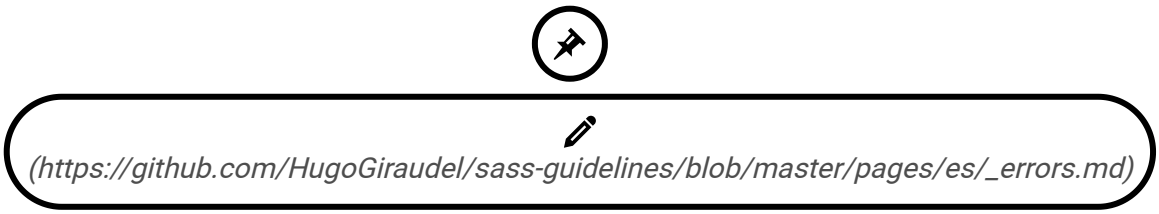
También asegúrate de respetar estas directrices para preservar la legibilidad:

- Deja siempre una línea en blanco antes del `@for`;
- Deja siempre una línea en blanco después de la llave de cierre (`}`) a no ser que la siguiente línea sea otra llave de cierre (`}`).

While

El ciclo `@while` no tiene ningún caso de uso en ningún proyecto real de Sass, puesto que no hay ninguna manera de romper un bucle desde el interior. **No lo utilices**.

Advertencias Y Errores



(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_errors.md)

Si hay una característica que a menudo es pasada por alto por los desarrolladores Sass, es la capacidad de generar advertencias e informes de errores de manera dinámica. De hecho, Sass viene con tres directivas personalizadas para imprimir contenido en el sistema de salida estándar (CLI, compiling app…):

- `@debug`;
- `@warn`;
- `@error`.

Dejemos `@debug` de lado, ya que claramente está pensado para depurar SassScript, lo que no es nuestro propósito aquí. Entonces nos quedamos con `@warn` y `@error` los cuales son notablemente idénticos, salvo porque uno detiene el compilador, mientras que el otro no. Te voy a dejar advinar cuál hace qué.

Ahora, siempre hay sitio en cualquier proyecto Sass para advertencias y errores. Básicamente cualquier mixin o función que espera un tipo o argumento específico podría lanzar un error si algo sale mal, o mostrar una advertencia cuando se hace una suposición.

Advertencias

Toma como ejemplo esta función de Sass-MQ *(https://github.com/sass-mq/sass-mq)* que intenta convertir un valor de `px` a `em`:

```
@function mq-px2em($px, $base-font-size: $mq-base-font
  @if unitless($px) {
    @warn 'Assuming #{$px} to be in pixels, attempting
    @return mq-px2em($px + 0px);
  } @else if unit($px) == em {
    @return $px;
  }
   @return ($px / $base-font-size) * 1em;
}
```

Si el valor no tiene unidades, la función asume que el valor estará expresado en píxeles. En este punto, hacer este tipo de hipótesis puede ser arriesgado, por lo que el usuario debe recibir una advertencia de que el software ha hecho algo que podría considerarse inesperado.

Errores

Los errores, a diferencia de las advertencias, impiden continuar al compilador. Básicamente paran la compilación y muestran un mensaje en la consola de salida, así como el rastro de seguimiento del error, lo que suele ser muy útil para la depuración. Debido a esto, los errores aparecerán cuando no hay manera de que el programa continúe funcionando. Cuando sea posible, trata de evitar el problema y mostrar una advertencia en su lugar.

Como ejemplo, digamos que construyes una función *getter* para acceder a los valores de un mapa específico. Se podría producir un error si la clave solicitada no existe en el mapa.

```
/// Z-indexes map, gathering all Z layers of the appli
/// @access private
/// @type Map
/// @prop {String} key - Layer's name
/// @prop {Number} value - Z value mapped to the key
$z-indexes: (
  'modal': 5000,
  'dropdown': 4000,
  'default': 1,
  'below': -1,
);

/// Get a z-index value from a layer name
/// @access public
/// @param {String} $layer - Layer's name
/// @return {Number}
/// @require $z-indexes
@function z($layer) {
  @if not map-has-key($z-indexes, $layer) {
    @error 'There is no layer named `#{$layer}` in $z-
         + 'Layer should be one of #{map-keys($z-index
  }

  @return map-get($z-indexes, $layer);
}
```

Para más información sobre cómo usar `@error` de manera eficiente, ésta introducción acerca del manejo de errores -En inglés *(http://webdesign.tuts-plus.com/tutorials/an-introduction-to-error-handling-in-sass--cms-19996)* puede serte de ayuda.

Herramientas

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_tools.md)

Lo bueno de un preprocesador CSS tan popular como Sass es que viene con un completo ecosistema de *frameworks*, *plugins*, librerías y herramientas. Después de 8 años de existencia, nos estamos acercando cada vez más a un punto donde todo lo que se puede escribir en Sass, ya ha sido escrito en Sass *(http://hugogiraudel.com/2014/10/27/rethinking-atwoods-law/)*.

Sin embargo, mi consejo es que debes disminuir al mínimo el número de dependencias. Manejarlas es casi un infierno del que no quieres formar parte. Además, hay pocas cosas que necesiten dependencias externas cuando se habla de Sass.

Compass

Compass *(http://compass-style.org/)* es el principal *framework* de Sass *(http://www.sitepoint.com/compass-or-bourbon-sass-frameworks/)* ahí fuera. Desarrollado por Chris Eppstein *(https://twitter.com/chriseppstein)*, uno de los diseñadores principales de Sass, y que en mi opinión, no creo que vaya a perder drásticamente su popularidad en un futuro cercano.

Aún asi, ya no uso Compass *(http://www.sitepoint.com/dont-use-compass-anymore/)*, la razón principal es que ralentiza mucho Sass. Ruby Sass es bastante lento por si solo, así que añadirle más Ruby y más Sass por encima no ayuda demasiado.

El punto es que usamos una parte muy pequeña de todo el *framework*. Compass es enorme. Los *mixins* de compatibilidad entre navegadores son solo la punta del iceberg. Funciones matemáticas, utilidades para imágenes, *spriting*… Hay muchas cosas que puedes hacer con este gran software.

Infortunadamente, todo esta azúcar no proporciona ninguna característica impresionante. Se puede hacer una excepción con el constructor de *sprites* que es *muy bueno*. Pero por ejemplo, Grunticon *(https://github.com/filament-group/grunticon)* y Grumpicon *(http://grumpicon.com/)* cumplen la misma función y además tienen la ventaja de poderse conectar durante el proceso de construcción.

De todas formas, no es que prohíba el uso de Compass aunque tampoco lo recomiendo, especialmente dado que no es compatible con LibSass (aunque

se han hecho muchos esfuerzos para ir en esa dirección). Si te sientes mejor utilizándolo, lo veo justo, pero no creas que obtendrás mucho al final del día.

> **Nota —** Ruby Sass tiene algunas optimizaciones importantes pendientes y que están específicamente orientadas a estilos de fuerte lógica con muchas funciones y *mixins*. Deberían mejorar drásticamente el rendimiento, hasta un punto en el que Compass u otros *frameworks* no volverán a ralentizar a Sass nunca más.

Sistemas De Retícula

No es una opción no utilizar un sistema de retícula ahora que el *Responsive Web Design* está por todas partes. Para que los diseños sean consistentes y coherentes en todos los tamaños, utilizamos algo similar a una cuadrícula para disponer los elementos. Para evitar tener que escribir el código de ésta retícula una y otra vez, algunas mentes brillantes han hecho la suya reutilizable.

Déjame ser directo: no soy muy fan de los sistemas de retícula. Por supuesto que veo su potencial, pero creo que muchos de ellos son completamente excesivos y en su mayoría sirven para dibujar columnas rojas sobre un fondo blanco en las presentaciones de algunos diseñadores listillos. ¿Cuándo fue la última vez que pensaste *Gracias-a-Dios-tengo-esta-herramienta-para-construir-retículas-de-2-5-3.2-π*?. Eso es, nunca. Porque en la mayoría de los casos, quieres usar la típica retícula de 12 columnas. Nada estrafalario.

Si estás usando un *framework* CSS para tu proyecto, como Bootstrap *(http://getbootstrap.com/)* o Foundation *(http://foundation.zurb.com/)*, hay muchas probabilidades de que ya incluya un sistema de retícula, en cuyo caso te recomendaría que usaras estos para evitar tener otra dependencia.

Si no trabajas con un sistema de retícula específico, te sentirás complacido al saber que hay dos motores de retícula de primera categoría para Sass: Susy *(http://susy.oddbird.net/)* y Singularity *(https://github.com/at-import/Singularity)*. Ambos hacen mucho más de lo que necesitas, así que puedes seleccionar el que prefieras entre estos dos y estar seguro/a de que todos los casos límite —incluso los más extremos— serán cubiertos. Si me

preguntas, Susy tiene una comunidad ligeramente mejor, pero esa es mi opinión.

O puedes trabajar con algo más casual, como csswizardry-grids *(https://github.com/csswizardry/csswizardry-grids)*. Con todo, la elección no tendrá mucho impacto en el estilo de tu código, así que todo depende de ti llegados a este punto.

SCSS-Lint

Limpiar el código es muy importante. Normalmente, seguir las pautas de una guía de estilo reduce la cantidad de errores en el código, pero nadie es perfecto y siempre hay cosas a mejorar. Asi que se puede decir que limpiar el código es tan importante como comentarlo.

SCSS-lint *(https://github.com/causes/scss-lint)* es una herramienta que te ayuda a mantener tus archivos SCSS limpios y legibles. Es completamente personalizable y fácil de integrar con tus propias herramientas.

Afortunadamente, las recomendaciones de SCSS-lint son muy similares a las descritas en este documento. Con el fin de configurar tu SCSS-lint según esta guía de Sass, te recomiendo seguir la siguiente configuración:

```
linters:

  BangFormat:
    enabled: true
    space_before_bang: true
    space_after_bang: false

  BemDepth:
    enabled: true
    max_elements: 1

  BorderZero:
    enabled: true
    convention: zero

  ChainedClasses:
    enabled: false

  ColorKeyword:
    enabled: true

  ColorVariable:
    enabled: false

  Comment:
    enabled: false

  DebugStatement:
    enabled: true

  DeclarationOrder:
    enabled: true

  DisableLinterReason:
    enabled: true

  DuplicateProperty:
    enabled: false

  ElsePlacement:
```

```
    enabled: true
    style: same_line

  EmptyLineBetweenBlocks:
    enabled: true
    ignore_single_line_blocks: true

  EmptyRule:
    enabled: true

  ExtendDirective:
    enabled: false

  FinalNewline:
    enabled: true
    present: true

  HexLength:
    enabled: true
    style: short

  HexNotation:
    enabled: true
    style: lowercase

  HexValidation:
    enabled: true

  IdSelector:
    enabled: true

  ImportantRule:
    enabled: false

  ImportPath:
    enabled: true
    leading_underscore: false
    filename_extension: false

  Indentation:
```

```
    enabled: true
    allow_non_nested_indentation: true
    character: space
    width: 2

  LeadingZero:
    enabled: true
    style: include_zero

  MergeableSelector:
    enabled: false
    force_nesting: false

  NameFormat:
    enabled: true
    convention: hyphenated_lowercase
    allow_leading_underscore: true

  NestingDepth:
    enabled: true
    max_depth: 1

  PlaceholderInExtend:
    enabled: true

  PrivateNamingConvention:
    enabled: true
    prefix: _

  PropertyCount:
    enabled: false

  PropertySortOrder:
    enabled: false

  PropertySpelling:
    enabled: true
    extra_properties: []

  PropertyUnits:
```

```
    enabled: false

  PseudoElement:
    enabled: true

  QualifyingElement:
    enabled: true
    allow_element_with_attribute: false
    allow_element_with_class: false
    allow_element_with_id: false

  SelectorDepth:
    enabled: true
    max_depth: 3

  SelectorFormat:
    enabled: true
    convention: hyphenated_lowercase
    class_convention: '^(?:u|is|has)\-[a-z][a-zA-Z0-9]

  Shorthand:
    enabled: true

  SingleLinePerProperty:
    enabled: true
    allow_single_line_rule_sets: false

  SingleLinePerSelector:
    enabled: true

  SpaceAfterComma:
    enabled: true

  SpaceAfterPropertyColon:
    enabled: true
    style: one_space

  SpaceAfterPropertyName:
    enabled: true
```

```
  SpaceAfterVariableColon:
    enabled: true
    style: at_least_one_space

  SpaceAfterVariableName:
    enabled: true

  SpaceAroundOperator:
    enabled: true
    style: one_space

  SpaceBeforeBrace:
    enabled: true
    style: space
    allow_single_line_padding: true

  SpaceBetweenParens:
    enabled: true
    spaces: 0

  StringQuotes:
    enabled: true
    style: single_quotes

  TrailingSemicolon:
    enabled: true

  TrailingZero:
    enabled: true

  TransitionAll:
    enabled: false

  UnnecessaryMantissa:
    enabled: true

  UnnecessaryParentReference:
    enabled: true

  UrlFormat:
```

```
    enabled: false

  UrlQuotes:
    enabled: true

  VariableForProperty:
    enabled: false

  VendorPrefixes:
    enabled: true
    identifier_list: base
    include: []
    exclude: []

  ZeroUnit:
    enabled: true
```

Si no estás convencido de la necesidad de usar SCSS-lint, te recomiendo que leas estos artículos: Limpia tu Sass con SCSS-lint -En inglés *(http://blog.martinhujer.cz/clean-up-your-sass-with-scss-lint/)*, Mejorando la calidad del código Sass en theguardian.com -En inglés *(http://www.theguardian.com/info/developer-blog/2014/may/13/improving-sass-code-quality-on-theguardiancom)* y Un guía de estilo SCSS auto-ejecutable -En inglés *(http://davidtheclark.com/scss-lint-styleguide/)*.

Nota — Si quieres usar SCSS-lint en el proceso de compilado de Grunt, estarás encantado de saber que hay un *plugin* de Grunt para esto, llamado grunt-scss-lint *(https://github.com/ahmednuaman/grunt-scss-lint)*.

También, si estás en la búsqueda de una aplicación limpia que funcione con SCSS-lint y similares, los chicos de Thoughtbot *(http://thoughtbot.com/)* (Bourbon, Neat…) están trabajando en Hound *(https://houndci.com/)*.

Too Long; Didn't Read

(https://github.com/HugoGiraudel/sass-guidelines/blob/master/pages/es/_tldr.md)

Esta guía es bastante larga y algunas veces es bueno tenerla resumida es una versión más corta. Tienes este resumen a continuación.

Principios Fundamentales

- Tener una guía de estilo es sobre todo tener **consistencia**. Si no estás de acuerdo con algunas reglas de Sass Guidelines, me parece justo, siempre y cuando seas consistente. ↩
- Sass debe permanecer siempre tan simple como se pueda. Evita utilizar sistemas complejos a no ser que sea absolutamente necesario. ↩
- Recuerda que algunas veces *KISS* (Keep It Simple, Stupid) es mejor que *DRY* (Don't Repeat Yourself). ↩

Sintaxis Y Formato

- La sangría debe hacerse con (2) espacios, no con tabulaciones. ↩
- Las líneas, deben ser, en la medida de lo posible, más cortas de 80 caracteres. Siéntete libre de dividirlas en disitintas líneas cuando sea necesario. ↩
- CSS debe escribirse adecuadamente, posiblemente siguiendo la guía CSS *(http://cssguidelin.es)* de Harry Roberts. ↩
- Los espacios en blanco son gratuitos, úsalos para separar elementos, reglas y declaraciones. No dudes en dejar líneas en blanco, no hacen daño. ↩

CADENAS

- Declarar la directiva `@charset` al inicio de tu hoja de estilo es altamente recomendable. ↩
- A menos que se apliquen como identificadores CSS, las cadenas deben ir entre comillas simples. Las URL también deben estar entre comillas. ↩

NÚMEROS

- Sass no tiene distinciones entre números, números enteros o decimales, por lo que los ceros finales (0) deben omitirse. Sin embargo, los ceros principales (0) ayudan a la lectura y deben agregarse. ↩
- Una longitud cero (0) no debe llevar su unidad. ↩
- La manipulación de unidades debe ser pensada como operaciones aritméticas, no como operaciones entre cadenas. ↩
- Para mejorar la legibilidad, los cálculos de nivel superior se deben incluir entre paréntesis. Además, las operaciones matemáticas complejas podrían dividirse en trozos más pequeños. ↩
- Los números mágicos resultan dramáticamente dañinos para la mantenibilidad del código y deben evitarse en todo momento. En caso de duda, explicar ampliamente el valor en cuestión. ↩

COLORES

- Los colores deben estar expresados en HSL siempre que sea posible, luego en RGB, luego en hexadecimal (en minúscula y en forma corta). Las palabras clave de color deben evitarse. ↩
- Prefiere `mix(..)` en lugar de `darken(..)` y `lighten(..)` al aclarar u oscurecer un color. ↩

LISTAS

- A menos que se utilice como una asignación directa a valores CSS separados con espacios, las listas deben separarse con comas. ↩
- Los paréntesis también se pueden usar para mejorar la legibilidad. ↩
- Las listas en una misma línea no deben tener una coma al final, las de varias líneas deben tenerlo. ↩

MAPAS

- Los mapas que contienen más de un par clave/valor deben escribirse en varias líneas. ↩
- Para ayudar a la mantenibilidad, el último par de un mapa debe tener una coma al final. ↩

- Las claves de mapa que resultan ser cadenas deben ir entre comillas como cualquier otra cadena. ↩

CLASIFICACIÓN DE DECLARACIONES

- El sistema usado para clasificar las declaraciones (alfabético, por tipo, etc.) no importa, siempre y cuando sea coherente. ↩

ANIDAMIENTO DE SELECTORES

- Evita anidar selectores cuando no sea necesario (lo que representa a la mayoría de los casos). ↩
- Usa el anidamiento de selectores para pseudo-clases y pseudo-elementos. ↩
- Las media queries también se pueden anidar dentro de su selector relevante. ↩

Convenciones De Nomenclatura

- Lo mejor es usar las mismas convenciones de nomenclatura CSS que están (excepto algunos errores) en minúsculas y delimitadas por guiones. ↩

Comentarios

- CSS es un lenguaje complicado; no dudes en escribir extensos comentarios sobre las cosas que parecen (o son) complejas. ↩
- Para variables, funciones, *mixins* y *placeholders* que establecen una API pública, usa los comentarios de SassDoc. ↩

Variables

- Usa el flag `!default` para cualquier variable de una API pública. Así puede ser cambiada sin generar problemas. ↩
- No uses el flag `!global` a nivel raiz ya que puede considerarse una violación de la sintaxis de Sass en el futuro. ↩

Extend

- Solo se extienden los *placeholders*, no los selectores existentes. ↩
- Extiende un *placeholder* tan pocas veces como sea posible con el finde de evitar efectos secundarios. ↩