Pequeña Guía Profunda de CSS

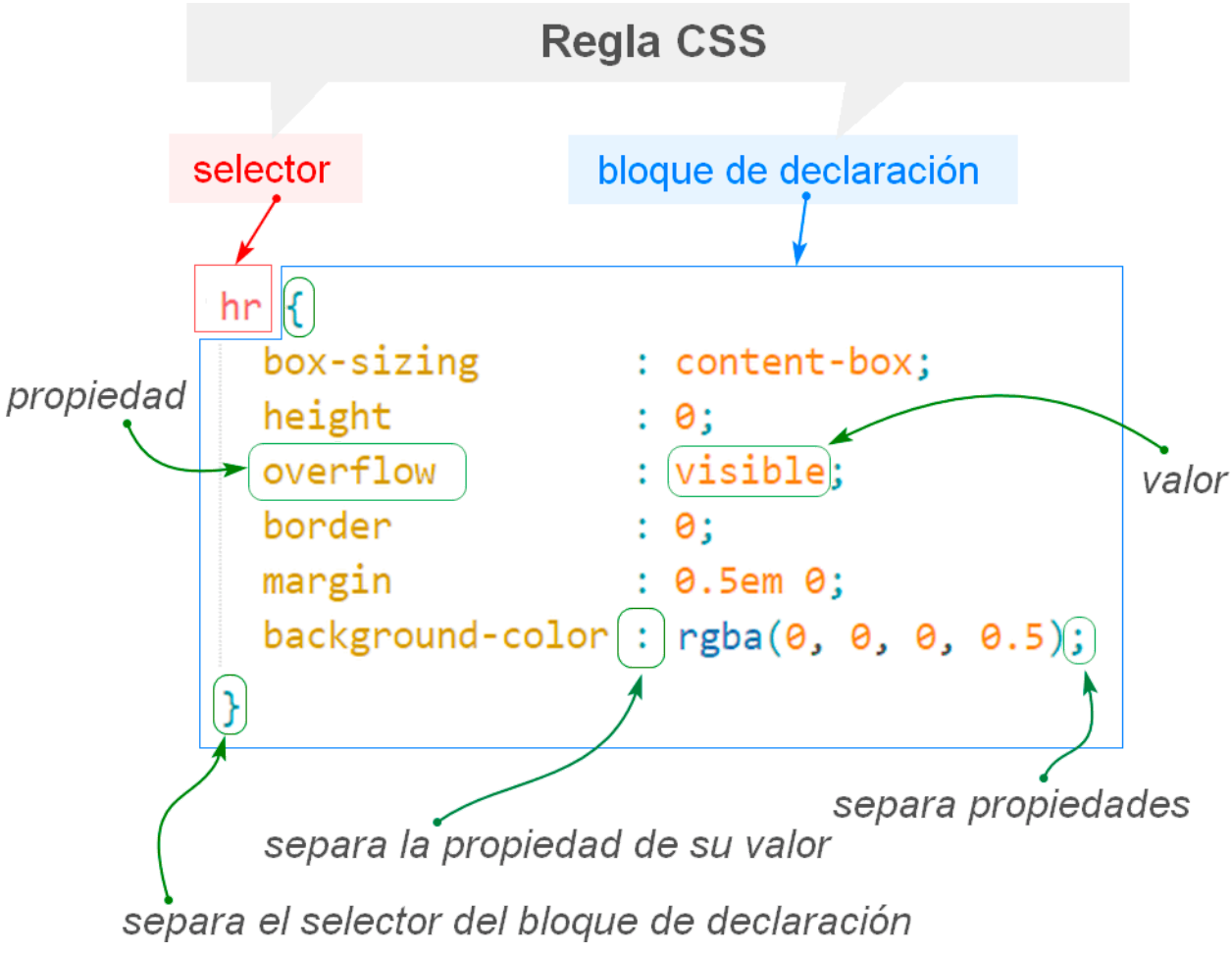
Primer Borrador: **08/10/2020**

Reglas CSS

En esta sección vamos a repasar algunos conceptos relativamente básicos de CSS, a hablar de metodologías de nombrado y pre-procesadores, y a dejar numerosas guías, artículos y enlaces para quien quiera profundizar más en algunos aspectos de CSS o simplemente para que puedan servir de ayuda y referencia en el futuro.

¿Qué es un selector?

Un **selector CSS** es la primera parte de una **regla CSS**. Es un patrón de elementos y otros términos que indican al navegador qué elementos HTML se seleccionan para aplicarles estilos. Los estilos se aplican por medio de **propiedades CSS**. A esta combinación de selector y propiedades se llama **regla CSS**.



Para ampliar:

- Learn CSS selectors (english) - developer.mozilla
- CSS Basics: The Syntax That Matters (english) - css-tricks
- Introducción a CSS (español) - uniwebsidad
- Learn CSS: The Complete Guide (english) - tuts+

Selectores básicos y avanzados

Hay varios selectores y formas de combinarlos, que podemos consultar en los siguientes enlaces:

- Selectores CSS (español) - developer.mozilla
- CSS Selectors (english) - developer.mozilla
- CSS Selectors Reference (english) - W3Schools

También hay formas de crear selectores más avanzados o concretos, con pseudo-clases y con pseudo-elementos, para repasar podemos revisar los siguientes enlaces:

- Pseudo-clases CSS (español) - developer.mozilla
- CSS Pseudo-classes (english) - developer.mozilla
- CSS Pseudo-classes (english) - W3Schools
- Pseudo-elementos CSS (español) - developer.mozilla
- CSS Pseudo-elements (english) - developer.mozilla
- CSS Pseudo-elements (english) - W3Schools

Para dar un repaso general a cómo funcionan los selectores en lugar de revisar documentación y guías más extensas, recomiendo este artículo:

- Beginner concepts: How CSS selector works (english) - css-tricks

Especificidad y herencia

¿Qué ocurre cuando más de una regla CSS aplica al mismo elemento HTML? ¿Qué estilos serían los que aplicaría el navegador? Aquí es donde entra en juego la especificidad. El navegador aplicará los estilos de la regla con el selector más específico.

A continuación dejo varios enlaces donde repasan este tema y muestran cómo calcular la especificidad de cada selector:

- Especificidad CSS (español) - developer.mozilla

- CSS Specificity (english) - developer.mozilla
- Specifics on CSS Specificity (english) - css-tricks
- CSS Specificity (english) - W3Schools
- CSS Specificity (english) - Dev.to

Por lo general se pueden extraer las siguientes buenas prácticas:

- **Evitar el uso de `!important`**: Al añadir esta declaración a una propiedad se sobreescribe a cualquier otra. Considerar hacer selectores más específicos si se quiere sobrescribir estilos de otras reglas, que al final da lugar a un código más limpio. Las únicas excepciones son cuando se quiere sobrescribir los estilos de otros frameworks CSS que ya tienen un `!important`, como puedan ser **Bootstrap** o **Tailwind**; y cuando se quiere sobrescribir los estilos en línea (*inline*) de algún framework de componentes, como pueda ser **Vuesax**.
- **Evitar el uso de estilos en línea (*inline*)**: al igual que en el punto anterior, los estilos en línea sobreescriben cualquier estilo aplicado desde CSS (salvo que la propiedad tenga un `!important`). La única excepción es cuando se quieren hacer animaciones complejas para las que se necesite JavaScript o establecer valores difíciles de calcular desde CSS. Aunque en este último caso, si hay compatibilidad con el navegador, es preferible utilizar variables de CSS y cambiarlas programaticamente con JavaScript en lugar de establecer el estilo en línea.
- **Tener en cuenta la herencia**: muchas propiedades se heredan de padres a hijos si no hay una regla que las sobreescriba, especialmente las relacionadas con la fuente, como puedan ser la familia, el tamaño, el color, etcétera. Así que hay que tener en cuenta la herencia: lo mismo podemos ahorrarnos algunos selectores si establecemos esas propiedades en un elemento padre común.

Para ampliar sobre uso de variables de CSS:

- Uso de propiedades personalizadas CSS (español) - developer.mozilla
- Using CSS custom properties (english) - developer.mozilla
- Las variables CSS son una realidad (español) - edteam
- Soporte variables CSS en distintos navegadores - caniuse

Guías de estilo y pre-procesadores

Recomiendo usar la metodología de nombrado **BEM** para escribir un código CSS limpio, claro y con una orientación a componentes que nos hace estructurarlo de forma que las reglas no se pisen unas a otras. Si queremos aprender, lo mejor es consultar directamente en su web: BEM -- Block Element Modifier (english).

Si queremos solamente repasar algunos conceptos clave, dejo algunos artículos más livianos:

- ¿Qué es BEM? (español) - animaticss
- Una introducción a la metodología BEM (español) - tuts+
- BEM 101 (english) - css-tricks
- 10 Common BEM Problems and How to Avoid Them (english) - smashingmagazine
- BEM By Example: Best Practices (english) - seesparkbox

Por otra parte, nunca está de más conocer algunas guías de estilo, como la anterior o como la de Mozilla, aunque no las sigamos al 100%. Otro ejemplo, aunque algo desactualizado, es la interesante recopilación de las guías de estilo que siguen diversas empresas bastante importantes que hicieron en CSS-Tricks.

Además, existen una gran variedad de pre-procesadores CSS que merece la pena conocer: SCSS/Sass, Less, Stylus, etcétera. Así como algún post-procesador como PostCSS, una de las últimas y más novedosas herramientas en llegar, y que ya cuenta con gran variedad de *plugins*, lo que lo convierte en una herramienta muy potente.

Aunque no es sencilla de configurar también es compatible con los pre-procesadores anteriores, si bien incorpora su propio pre-procesador, llamado PreCSS y muy parecido a **SCSS/Sass**, por lo que quizá sea innecesario el esfuerzo.

Pero puesto que nuestro pre-procesador estándar es **SCSS/Sass**, voy a centrarme en éste para dejar algunas guías por si se quiere refrescar algún concepto (aunque lo mejor siempre es acudir a la documentación oficial):

- The Complete Guide to SCSS/Sass (english) - medium
- The definitive guide to SCSS (english) - logrocket

Por último, recordar que existen asimismo guías de estilo para SCSS/Sass que nunca está de más conocer:

- Sass Guidelines (english)
- Sass Style Guide (english) - css-tricks

Otras guías y recursos

Para terminar, dejo una lista de guías, artículos, utilidades y recursos en general por si se quiere profundizar en algunos apartados de CSS o para que sirvan de referencia:

- DOM (Document Object Model) (english) - developer.mozilla
- CSS Object Model (CSSSOM) (english) - developer.mozilla
- A Complete Guide to Flexbox (english) - css-tricks
- A Complete Guide to Grid (english) - css-tricks
- The CSS Box Model (english) - css-tricks
- A Complete Guide to CSS Functions (english) - css-tricks
- Using the Web Animations API (english) - developer.mozilla
- A Complete Guide to CSS Media Queries (english) - css-tricks
- Centering in CSS (english) - css-tricks
- A Complete Guide to Data Attributes (english) - css-tricks
- HTML Cheat Sheet (english) - websideup
- CSS Cheat Sheet (english) - websideup
- Transition (english) - css-tricks
- Animation (english) - css-tricks
- Clipping and Masking in CSS (english) - css-tricks
- Basics of CSS Blend Modes (english) - css-tricks
- PostCSS - Sass Killer or Preprocessing Pretender? (english) - ashleynolan
- PostCSS: ¿Qué es? ¿Es mejor que Sass, Less o Stylus? (español) - bufa
- Moving from SASS to PostCSS: what, why and how (english) - medium
- CodePen: Online code editor for front-end developers - (ide)
- CodeSandbox: Online IDE for fast web development - (ide)
- JSFiddle: JS, CSS & HTML code playground - (ide)
- JSBin: Collaborative JavaScript debugging - (ide)
- Dabblet: interactive playground for quickly testing CSS & HTML snippets - (ide)
- LiveWeave: JS, CSS & HTML real-time preview - (ide)
- CSSMatic: Ultimate CSS tools (gradient, borders, noise texture, shadows) - (tool)
- Paletton: The Color Scheme Designer - (tool)
- Coolors: The super fast color scheme generator - (tool)
- CSS Arrow Generator - (tool)
- PXtoEM: PX to EM conversion made simple - (tool)
- The Ultimate Web Code Generator - (tool)
- The Ultimate CSS Generator - (tool)
- The Ultimate HTML Generator - (tool)
- CSS Code: Interactive CSS generator by example - (tool)
- HTML Code: Interactive HTML generator by example - (tool)

Rendimiento de CSS

Si bien el rendimiento de CSS es una de las últimas cosas que nos debería preocupar cuando estamos empezando una página web pequeña, puesto que los cuellos de botella más comunes suelen ser JavaScript y la carga de recursos, conforme la web va creciendo y se van acumulando reglas sí se convierte en un factor a tener en cuenta. Especialmente en *web apps* como la nuestra, cuyos ficheros CSS acumulan miles de líneas y en la cual la experiencia de usuario, que también incluye que la página vaya lo más fluida posible, se convierte en un requisito indispensable.

En este apartado vamos a estudiar el rendimiento de CSS, pasando por selectores, propiedades, animaciones, cómo interfiere JavaScript en él y otros conceptos; para tratar de dar una serie de consejos y pautas generales.

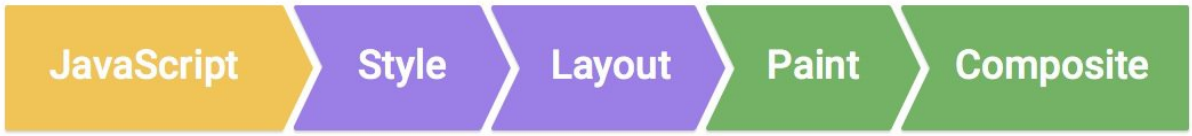
¿Qué afecta al rendimiento de CSS?

Lo **primero** de todo y más evidente es el **tamaño** del código. Cuanto más pese el CSS, más tiempo tardará en ser descargado y parseado. Por tanto, la primera optimización pasa por:

- **Minificar el código de CSS** antes de servirlo en producción con alguna herramienta de post-procesado. Esto va desde lo más básico, como eliminar comentarios y espacios innecesarios para ahorrar caracteres; hasta soluciones más radicales y complejas de implementar en las que no profundizaremos, como el renombrado de clases (ej: sustituir `.container--small` por `.ab` en toda la app).
- **Ahorrar código CSS** eliminando fuentes y reglas no usadas, agrupando selectores cuando sea posible, aprovechando la herencia y teniendo una buena arquitectura de clases: con clases que apliquen estilos generales con poca especificidad y clases más concretas para componentes (siguiendo la metodología BEM) con una especificidad más alta.
- Tener en cuenta **otros aspectos generales de rendimiento web**, como la forma en que se cargan los recursos, cómo los recursos bloqueantes que impiden que la página se renderice hasta que se han cargado, el orden correcto de carga de los recursos, la pre-carga de algunos recursos o la carga asíncrona de otros, aprovechar el cacheado si el recurso no ha cambiado, etcétera.

Renderizado de CSS

Lo **segundo** es en sí el **renderizado** de CSS, donde nos centraremos, junto con algunas interacciones de JavaScript con el **DOM** o los estilos. Cuando se cargan una serie de reglas CSS para aplicar estilos a los elementos HTML del DOM, conviene diferenciar entre los distintos pasos del renderizado para saber detectar cuellos de botella:



1. **Estilo (*style*)**: Determina qué reglas y propiedades CSS se aplicarán a qué elementos HTML, siguiendo el esquema de especificidad de los selectores. En el siguiente capítulo se profundizará en el rendimiento de los selectores.

2. **Diseño (*layout*)**: Determina cómo se posicionará el elemento en la pantalla y cuánto espacio ocupará, según cómo están posicionados el elemento padre, los elementos anteriores y algunas propiedades CSS: `height`, `width`, `display` (ej: si es `flex` o `grid`), `position` (`top`, `right`, `left` y `bottom` también), `margin`, `padding`, `border-width`, etcétera.

3. **Pintado (*paint*)**: Determina el aspecto visual del componente, según algunas propiedades CSS como la fuente, el color, el fondo, los bordes, etcétera.

4. **Composición (*composite*)**: Crea las capas (*layers*) y les aplica ciertos estilos para mezclarlas entre sí según algunas propiedades: `opacity`, `transform`, `z-index`, `filter`, etcétera.

Una vez que se ha hecho el primer renderizado, es útil saber cuándo se va a disparar (o *trigger*, desencadenar) el re-calculado de cada uno de los pasos. Por ejemplo, el primer paso de **estilo** solo se dispara en el primer renderizado, en la navegación entre páginas (en una *SPA*) y en general cuando cambie el **DOM**. Por tanto, solo es importante en estos casos.

El resto de pasos se dispararán cada vez que cambien las propiedades CSS asociadas a ese paso, generalmente porque mute el elemento y se le apliquen otras reglas CSS, por ejemplo, durante un `:hover` o al cambiar sus clases. También múltiples veces durante una **animación** (`animation`) o transición (`transition`). El número de veces que se disparará durante una animación (o transición) depende del rendimiento, idealmente serán 60 veces por segundo (60 FPS o *frames* por segundo) hasta que termine la animación.

Esas son muchas veces, así que hay que tener en cuenta las siguientes cuestiones. Puesto que el **diseño** de un elemento viene determinado también por los elementos anteriores y el elemento padre (y los ancestros), cada vez que se anime cualquier propiedad asociada a este paso, también se forzará el re-calculado del diseño de todos los elementos hijos (y descendientes general) y de los posteriores. Estos re-calculados se ejecutan en cascada: primero en los elementos afectados directamente, después en sus hijos, después en los hijos de sus hijos, etcétera. Este fenómeno se conoce como ***reflow*** o ***layout shift***.

Por tanto, el **diseño es muy costoso de animar** y hay que tratar de evitarlo siempre que sea posible, ya que con seguridad se reducirá la fluidez de la animación. Más adelante veremos cómo. Lo menos costoso de animar es la composición.

También hay ciertas acciones que provocarán un *reflow*, como hacer *scroll* o cambiar el tamaño de la ventana, pero estos escenarios no se pueden prever ni controlar.

Hay otro fenómeno conocido como ***layout trashing***, menos común, que también causa el re-calculado del diseño y del pintado de algunos o todos los elementos: el acceso vía JavaScript a algunas propiedades o métodos del elemento o de la ventana. Generalmente aquellas relacionadas con el tamaño, el *scroll* y la posición del elemento como `el.scrollHeight`, `el.getBoundingClientRect()`, o `el.offsetHeight` entre otras.

Por tanto, hay que tener en cuenta el impacto que tendrá cada vez que se acceda a una de estas propiedades. En teoría, si los accesos son secuenciales y no se hace ninguna modificación del DOM no se provocará un *layout trashing* por cada uno de ellos.

Algo **parecido también sucede con las mutaciones del DOM**: o peor, suele ocurrir que cada mutación individual del DOM provoque un re-calculado de alguno de los pasos en ciertos elementos, aunque la próxima mutación esté literalmente en la siguiente línea de código JavaScript. Entrando un poco en profundidad: el motor del navegador almacena un *snapshot* del *frame* anterior para intentar aprovecharlo en el siguiente *frame*. Tanto las operaciones de lectura (las que causan *layout trashing*) como las de escritura (las modificaciones del **DOM**) causan una invalidación del *snapshot*, lo que provoca que sea necesario un recalculado. Más adelante profundizaremos en esto.

Para mitigar problemas de rendimiento, se recomienda agrupar todas las llamadas en lotes (*batch*) para que el navegador haga todo el re-renderizado de golpe en el mismo *frame*. Más adelante veremos cómo.

De aquí se pueden extraer algunos consejos generales o valoraciones a tener en cuenta:

- Hacer todas las mutaciones del DOM y todas las operaciones que provocan *layout trashing* por lotes. Si son mutaciones del DOM y estamos usando algún framework con *virtual DOM*, ya se encargará él de aplicar las mutaciones por lotes. Si no

usamos ningún framework o tratamos de evitar un *layout trashing*, podemos usar la función **requestAnimationFrame()**, que hará que el navegador ejecute el *callback* recibido como parámetro antes de pintar el siguiente *frame*.

- Intemendientemente de que usemos la función **requestAnimationFrame()**, hay que tratar de agrupar las operaciones de escritura (las que causan *layout trashing*) y las de escritura (las modificaciones del **DOM**), y hacer antes las primeras. Eso en caso de que se usen conjuntamente y no usemos ningún framework, claro. Esto se debe a que las operaciones secuenciales de cada tipo sólo invalidan el *snapshot* del *frame* una vez: la primera. En las operaciones del mismo tipo siguientes el *snapshot* sigue siendo válido porque ya se ha recalculado anteriormente todo lo necesario. Si mezclamos operaciones de lectura con escritura el *snapshot* se invalidará constantemente. Pero al agruparlas, el *snapshot* se invalidará solo dos veces: una vez con la primera operación de lectura y otra con la de escritura.
- Las búsquedas de elementos es mejor hacerlas fuera de la llamada a la función **requestAnimationFrame()**, y generalmente es recomendable mantener los elementos cacheados o pre-calculados en alguna variable (un ejemplo de búsqueda, también llamadas *queries*: `document.querySelector('.header')`). Esto es debido a que las búsquedas de elementos en el DOM son una acción costosa. Por lo general, deben evitarse acciones y cálculos costosos que hagan que el *callback* tarde más de 16 milisegundos en ejecutarse (el tiempo entre un *frame* y otro cuando tenemos 60 FPS). Como con las *queries*, siempre que sea posible es mejor cachear estos cálculos o al menos pre-calcularlos antes de la función.
- Evitar animar cualquier propiedad que fuerce un *reflow/layout shift* en cada *frame*, mejor tratar de conseguir el mismo efecto animando otras propiedades. **Regla de oro**: prácticamente cualquier propiedad que afecte al diseño se puede simular usando `transform`, por ejemplo: `translate()` (también `translateX()` o `translateY()`) en lugar de `margin`, `top`, `right`, `bottom` o `left`; o `scale()` (también `scaleX()` o `scaleY()`) en lugar de `width` o `height`.
- Si conocemos a priori que una propiedad se va a animar bastantes veces, podemos optimizar la animación con la propiedad `will-change: nombre-propiedad`. Esto hace que el navegador aplique ciertas optimizaciones a costa de un mayor consumo de memoria. Por tanto, hay que usarla con cautela: en transiciones que sabemos que van a ser muy frecuentes o costosas. Por otra parte, si se controla cuándo se va a producir la transición, otra opción es activar la propiedad poco antes de la transición y desactivarla poco después.
- Si vamos a hacer cambios en el DOM de forma muy frecuente y asociados a eventos (como al *scroll*), valorar agrupar los cambios con funciones como `debounce()`. Otra opción a valorar, según lo qué se quiera conseguir, es hacer uso de la API **Intersection Observer**.
- El posicionamiento absoluto (`position: absolute`) y fijo (`position: fixed`) hace que el elemento deje de afectar al diseño de elementos posteriores.

- Aunque no sea demasiado costoso el *reflow* porque se use posicionamiento absoluto o fijo, sigue siendo más eficiente utilizar `transform: translate()` en una animación en lugar de `top`, `right`, `bottom` o `left`. En primer lugar, porque el código JavaScript y los tres primeros pasos del renderizado se ejecutan en el **hilo principal** (*main thread*) del navegador, que suele estar más saturado, mientras que del último paso se encarga un segundo hilo del navegador, el **hilo compositor** (*compositor thread*). En segundo lugar, porque se evita el *reflow* de los elementos descendientes. Y, por último, porque este segundo hilo no solo es ejecutado por la CPU sino también parcialmente por la GPU, mucho más eficiente tareas gráficas.
- En resumen, lo ideal es animar únicamente aquellas propiedades relacionadas con el paso de composición, tanto para evitar recalculados en muchos elementos como porque se ejecutan en un hilo aparte apoyado por la GPU. En caso de que no te quede más remedio hacerlo de otra forma, mejor animar los elementos del árbol del DOM más profundos o con menos hijos para que la penalización al rendimiento sea la menor posible (consejo: con este fin, a veces es mejor animar pseudo-elementos, cosa que veremos en otro capítulo más adelante).

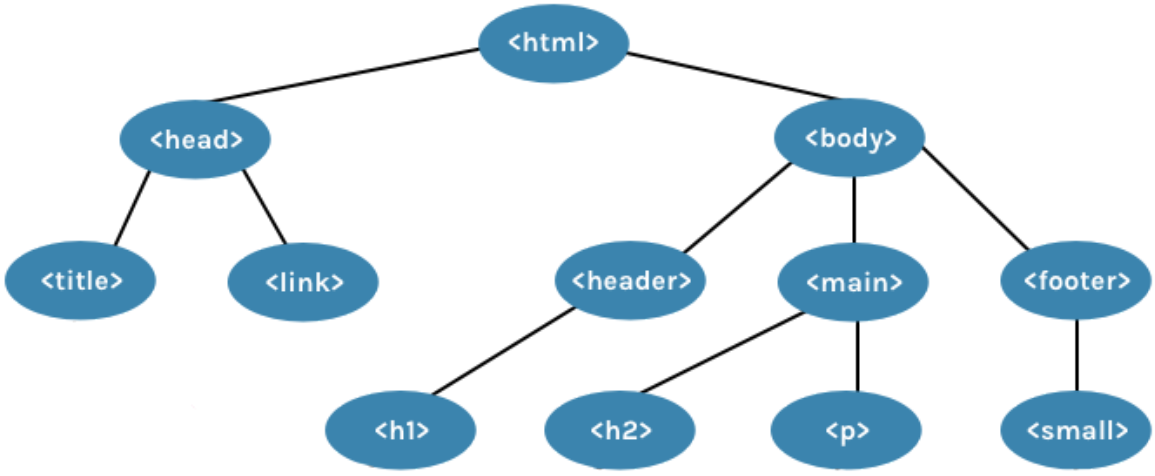
Todo lo anterior está bastante simplificado, así que recomiendo varios artículos por si se quiere profundizar o ampliar información:

- Improve CSS performance and file size (english) - dev.to
- Things nobody ever taught me about CSS (english) - dev.to
- Rendimiento de CSS (español) - developers.google
- Improving performance by eliminating render-blocking CSS and JS (english) - dev.to
- Avoid Large, Complex Layouts and Layout Thrashing (english) - developers.google
- Web Performance: Minimising DOM Reflow (english) - medium
- Understanding Repaint and Reflow in JS (english) - usejournal
- Performance Fundamentals (english) - developer.mozilla
- Performance best practices for front-ends (english) - developer.mozilla
- CSS GPU Animation: Doing It Right (english) - smashingmagazine
- An Introduction to Hardware Acceleration with CSS (english) - sitepoint
- Using will-change (english) - css-tricks
- Using requestAnimationFrame (english) - css-tricks
- Debouncing and Throttling Explained Through Examples (english) - css-tricks
- Animation Performance 101: Browser Under the Hood (english) - viget
- Animation Performance 101: Optimizing JavaScript (english) - viget
- Intersection Observer API (english) - developer.mozilla
- Lighthouse performance scoring Google's tool (english) - web.dev
- Best way to lazy load images for maximum performance (english) - dev.to
- CSS Performance Hacks (english) - readium
- Browser painting and considerations for web performance (english) - css-tricks

Selectores y propiedades de CSS

Habiendo repasado el capítulo anterior, ahora toca profundizar en otras cuestiones relativas a la eficiencia del CSS. Los selectores son una de ellas. Las propiedades, aunque menos importantes, también. ¿Qué selectores tardan más en procesarse? ¿Qué propiedades son más difíciles de calcular?

Empezando por los selectores, lo importante es **conocer cómo interpreta el navegador un selector y cómo determina a qué elementos del DOM aplicaría**. Aunque parezca contra-intuitivo, lo hace al revés de cómo sería natural para una persona: de derecha a izquierda en el caso de los selectores y recorriendo el árbol del DOM desde las hojas.



Para entenderlo mejor, veamos un ejemplo práctico: supongamos que tenemos una regla para aplicar estilos a los enlaces de navegación de una cabecera, tal que así:

```
body header nav ul li a[href] { }
```

Para encontrar los elementos a los que aplicar esa regla, el navegador leería su selector de derecha a izquierda, es decir, empezando por la última condición. Primero buscaría todos los elementos `a[href]` que existen en el DOM (empezando por los elementos sin hijos), luego filtraría los resultados por aquellos que tienen un ancestro `li`, después por aquellos que tienen un ancestro `ul`, etcétera.

A la primera parte del selector, es decir a `a[href]`, se le suele llamar parte principal del selector (*key selector*), pues es el elemento al que se aplicarán los estilos y también la parte más crítica para medir la eficiencia del selector. Por tanto, aunque no parezca intuitivo al principio porque los selectores de ID son los más eficientes, éste sería un ejemplo de mal selector:

```
#nav-bar a { }
```

Sabiendo esto se pueden extraer los siguientes consejos generales y consideraciones:

- **Los selectores cuanto más cortos mejor**. Un selector con menos condiciones tardará menos en procesarse que uno con muchas, ya que tiene que hacer menos filtrados y comprobaciones. Este es uno de los motivos por los que se recomienda tener una serie de reglas generales con baja especificidad que apliquen por estilos generales, en lugar de crear reglas específicas con selectores más complejos para elementos de componentes donde queremos aplicar el mismo estilo.
- **Si un selector va a fallar, que lo haga cuanto antes**. Cuando un selector a medio analizar falla en una rama del árbol del DOM, se detiene y pasa a la siguiente. Así que hay que tratar de ser lo más restrictivo posible para que el selector falle pronto y ahorrar en tiempo de computación.
- **Evitar utilizar etiquetas HTML en los selectores**. Está muy relacionado con lo anterior. El selector, empezando por su parte principal, debe ser lo más restrictivo posible. De esta forma fallará pronto y se ahorrará en tiempo de procesamiento. Un ejemplo: `.car-detail p` es un mal selector porque elementos `p` en el DOM puede haber muchos y el navegador tendrá que comprobar para cada uno de ellos si tienen un ancestro con `.car-detail`. Por otra parte, puede haber algunos elementos `p` dentro de `.car-detail` a los que no queramos aplicar el estilo. Sin embargo `.car-detail .car-detail__text` es un selector mucho más eficiente porque elementos con `.car-detail__text` habrá bastantes menos.
- **El DOM cuanto menos profundo y con menos ramas mejor**. Esta es quizá la parte más complicada de aplicar. Cada elemento nuevo en el DOM, añade una complejidad extra a la hora de renderizar la página (no solo por la parte de CSS, que también. Por ejemplo también afecta al *virtual DOM* de los frameworks). Mejor mantenerlo lo más simple posible, valorando si cada nuevo elemento es necesario o se podría conseguir lo mismo con menos elementos (sin perder nunca de vista el valor semántico de los elementos HTML, claro). **Regla de oro**: casi siempre podemos unir todos los *wrappers* (elementos envoltorio) en uno solo, y muchos efectos visuales podemos obtenerlos sin elementos extra de igual forma usando los pseudo-elementos `::before` y `::after`.
- **Evita condiciones lentas**. Las pseudo-clases `:nth-child()`, `:nth-of-type()`, `:nth-last-of-type()` y `:nth-last-child()` son particularmente lentas. Los selectores de tipo también, como hemos visto antes, por la gran cantidad de elementos a los que podrían aplicar. Evita usarlos si es posible, o al menos como parte principal del selector. En tercer lugar quedan los selectores de atributo, si bien éstos pueden ser relativamente útiles usados en conjunto con otras condiciones más rápidas. Regla de oro: los selectores de ID son los más rápidos junto con los selectores de clase, sin embargo, los ID son únicos y tienen un significado especial en HTML, así que al final la mayoría de tus selectores deben usar clases.

- **No sobre-cualifiques los selectores**. Evita añadir condiciones innecesarias o redundantes. Por ejemplo, el selector `a.link` es un poco redundante y algo lento. Se supone que no vas a añadir la clase "link" a nada que no sea un enlace, así que usa mejor únicamente `.link`. Otro ejemplo, utilizando BEM, sería el selector `.container.container--small`: es más eficiente usar `.container-small` a secas.
- **Ten en cuenta la herencia**. Algunas propiedades se heredan. No es necesario que apliques `.navbar .link { font-family: Roboto; }` porque con `.navbar { font-family: Roboto; }` te vale ya que es una propiedad que se hereda.
- **No abuses de los selectores descendientes**. Como hemos leído hasta ahora, cuanto más sencillo sea el selector mejor. Pero a la hora de escribir reglas CSS con Sass y siguiendo la metodología BEM es un consejo difícil, pues es bastante común anidar muchos clases hasta llegar a selectores demasiado largos como `.car .car__detail .car__detail__description .car__description__text`. Los selectores descendientes son bastante lentos y mejor no anidar muchas reglas. El selector anterior podría simplificarse perfectamente como `.car .car__description__text`. La excepción a esta regla es cuando se necesita escribir selectores con mayor especificidad que otros para sobrescribir los estilos.
- **Limpia tu CSS**. Una regla CSS sin utilizar siempre va a ser más costosa que cualquier otra regla. No tengas CSS innecesario y no usado. Utiliza herramientas de limpieza o detección. Procura no usar más frameworks de los necesarios, y trata de extraer de estos solo lo que necesites, no importes el framework entero.
- **Es mejor no usar selectores de ID salvo en casos concretos**: Lo hemos comentado anteriormente, pero creo que merece la pena hacer hincapié en esto. Los ID son únicos, las clases no, y se supone que a la hora de escribir una regla CSS no la haces pensando en que va a afectar a un único elemento. Generalmente, escribes o una regla genérica de baja especificidad que va a ser utilizada en múltiples elementos, o reglas específicas de componentes que pueden renderizarse más de una vez en la página. Además, los ID tienen un significado especial en HTML, principalmente, la de crear partes del documento enlazables desde el mismo documento.

Sobre lo segundo, las propiedades, lo importante es conocer **qué propiedades son más costosas**. En resumen, hay unas pocas propiedades que son conocidas por ser bastante costosas:

- `border-radius`
- `box-shadow`
- `filter`
- `position: fixed`

Esto no quiere decir que no debas usarlas, porque el impacto tampoco es crítico. Pero sí se pueden tener en cuenta los siguientes consejos generales:

- Es mejor no animarlas. Salvo en el caso de `box-shadow`, que veremos más adelante, tampoco es común querer animar alguna de esas propiedades, así que no debería ser un problema.
- Si se quiere tener una caja con bordes redondeados, es más eficiente aplicar `border-radius` una sola vez junto con `overflow: hidden` al elemento padre o a un *wrapper* antes que `border-radius` a varios de sus hijos o descendientes.
- Si se quiere animar `box-shadow`, por ejemplo al hacer `:hover`, es más eficiente que el `box-shadow` se encuentre en un pseudo-elemento como `::after` o `::before` junto con `opacity: 0`, y que lo que se anime no sea `box-shadow` sino ésta propiedad, pasando a `opacity: 1` cuando la sombra deba ser visible. De esta forma la sombra, que es lo costoso, se calcula una única vez y luego sencillamente se utiliza la opacidad de la pseudo-clase para mostrarla u ocultarla.

Como en capítulos anteriores, dejo una lista de recursos para quien quiera profundizar:

- Efficiently Rendering CSS (english) - css-tricks
- Writing efficient CSS selectors (english) - csswizardry
- Optimizing CSS: ID selectors and other myths (english) - sitepoint
- CSS selector performance (english) - ecss
- CSS performance revisited: selectors, bloat and expensive styles (english) - benfrain
- Improving CSS performance (english) - johno
- 20 Tips for Optimizing CSS Performance (english) - sitepoint

Por último, dejo una serie herramientas que nos pueden ayudará a tener un código CSS más limpio y depurado:

- mini-css-extract-plugin
- clean-css
- CSSO (CSS Optimizer)
- minimalcss
- PurgeCSS

También mencionar de nuevo el post-procesador PostCSS, que tiene todo tipo de *plugins*, algunos como cssnano para minificar, y otros como PostCSS Cleaner u la adaptación de Purge CSS para eliminar el CSS no utilizado. Actualmente es una locura migrar de **SCSS/Sass** a **PostCSS** (o incluso hacer que convivan), pero pienso que es importante ir conociendo las nuevas herramientas.