APLICACIONES WEB

HOJAS DE ESTILOS EN CASCADA (CSS)

Presentado por:
Edwin Enoc Vargas Cruz

Matrícula:
240513

Carrera:
DESM

Grado y Grupo:
4-B-6

Nombre Maestro:
Carlos Fernando Ovalle García

Hojas de Estilos en Cascada (CSS): Conceptos, Técnicas Avanzadas y Reglas de Usabilidad para Interfaces Web

1. Introducción al documento

Este documento está dirigido a desarrolladores web y diseñadores de interfaces que deseen comprender y dominar CSS (Cascading Style Sheets). Se explica en detalle qué es CSS, su relación con HTML y JavaScript, las ventajas de usarlo y su evolución histórica. A lo largo de secciones organizadas, se abordan desde conceptos básicos (selectores, modelo de caja, diseño responsivo) hasta técnicas avanzadas (Flexbox, Grid, animaciones, APIs modernas, accesibilidad). Se incluyen numerosos ejemplos de código, explicaciones paso a paso y buenas prácticas. El objetivo es ofrecer una guía completa que sirva como referencia profesional.

2. Tabla de Contenido

Hojas de Estilos en Cascada (CSS): Conceptos, Técnicas Avanzadas y Reglas de Usabilidad para Interfaces Web 1

1. Introducción al documento 1
3. Introducción a CSS 3
4. Sintaxis y conceptos básicos 4
    4.1 Selectores CSS básicos 4
    4.2 Pseudo-clases y pseudo-elementos 5
    4.3 Especificidad, cascada y `!important` 6
    4.4 Herencia y orden de origen 7
    4.5 Comentarios y organización 8
5. Selectores avanzados y rendimiento 8
6. Unidades y valores 9
7. Modelo de caja (Box Model) y propiedades relacionadas 10
8. Diseño y layout 12
    8.1 Display y flujo de elementos 12
    8.2 Flexbox 13
    8.3 CSS Grid 14
    8.4 Container Queries (Consultas de contenedor) 15
9. Posicionamiento y stacking 15
    9.1 Posicionamiento CSS 15

9.2 Orden Z (`z-index`) y contextos de apilamiento 16
9.3 Transformaciones 2D/3D (`transform`) 17
10. Tipografía y fuentes 18
10.1 Propiedades tipográficas básicas 18
10.2 Web Fonts y `@font-face` 18
10.3 Rendimiento de fuentes 19
11. Colores y modelos de color 19
11.1 Modelos de color 19
11.2 Contraste y accesibilidad (WCAG) 20
11.3 Gradientes, blending y filtros 21
12. Variables CSS (Custom properties) 21
13. Interactividad: pseudo-clases, estados y formularios 23
13.1 Estados de formularios y controles 23
13.2 Controles personalizados y accesibilidad 23
14. Transiciones, animaciones y rendimiento 24
14.1 Transitions 24
14.2 Animations (Keyframes) 25
14.3 Rendimiento en animaciones 26
15. Filtros, máscaras y recortes 26
15.1 Filters (`filter`) 26
15.2 Máscaras y `clip-path` 27
16. Diseño responsivo y media queries 27
16.1 Media Queries 27
16.2 Imágenes responsivas 28
16.3 Layouts fluidos 29
17. Accesibilidad (A11y) aplicada al CSS 29
18. Compatibilidad entre navegadores y autoprefixing 30
19. Rendimiento y optimización 31
20. Arquitectura CSS y patrones 31
21. Herramientas y workflow 32
22. Seguridad y políticas (CSP) ..... **¡Error! Marcador no definido.**
23. Internacionalización y escritura bidireccional 33
24. Print styles y medios no-pantalla 34

25. Depuración y herramientas de desarrollo.....34
26. Nuevas APIs y tendencias.....35
27. Buenas prácticas de usabilidad (checklist) .....**¡Error! Marcador no definido.**
28. Ejemplos completos y casos de estudio.....**¡Error! Marcador no definido.**
29.2 Referencias APA 35

3. Introducción a CSS

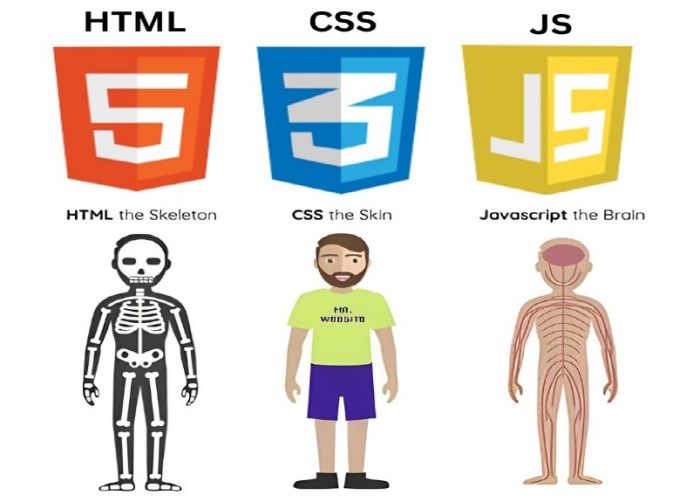
¿Qué es CSS? CSS (Cascading Style Sheets) es el lenguaje de estilos usado para describir cómo se presenta un documento escrito en HTML en distintos medios (pantalla, papel, etc.). Permite separar la estructura del contenido (HTML) de su presentación (colores, fuentes, diseño), lo que mejora la accesibilidad y facilita el mantenimiento. CSS **se introdujo a mediados de los 90**, impulsado por Håkon Wium Lie en el W3C, para suplir las limitaciones de estilo de los navegadores de la época. Desde entonces ha evolucionado en varias versiones (CSS1, CSS2, CSS3, CSS4), incorporando nuevas capacidades como Flexbox, Grid, variables, etc.

Relación HTML/CSS/JS: Mientras HTML define la estructura semántica de la página, CSS controla su estilo visual, y JavaScript añade interactividad dinámica. Juntos forman el stack base de la Web. La separación de preocupaciones (estructura vs. presentación) hace que el contenido sea accesible (por ejemplo, para lectores de pantalla) y permite cambiar el diseño de todo un sitio modificando solo los archivos CSS. Por ejemplo, el mismo HTML puede mostrarse diferente en móvil o escritorio usando reglas CSS apropiadas.

Ventajas de CSS: CSS permite **definir estilos reutilizables** que se aplican a múltiples elementos o páginas, ahorrando mucho trabajo. Al centralizar el diseño en archivos externos, basta con cambiar una regla CSS para actualizar muchas páginas. Además, CSS

ofrece potentes capacidades de diseño (posicionamiento, layouts flexibles, animaciones) sin necesidad de código JavaScript. Por ejemplo, con `display: grid` o `display: flex` es posible crear estructuras complejas de forma simple.

Responsabilidades de CSS: A efectos prácticos, CSS se encarga de todo el comportamiento visual de la interfaz: colores, tipografía, espaciados, disposición de elementos, respuestas a estados (hover, foco), etc. Una responsabilidad importante es **accesibilidad**: por ejemplo, elegir contrastes adecuados o proveer indicadores de foco (outline) para usuarios de teclado. También optimizar el rendimiento visual (minimizar repaints) es tarea del desarrollador CSS.

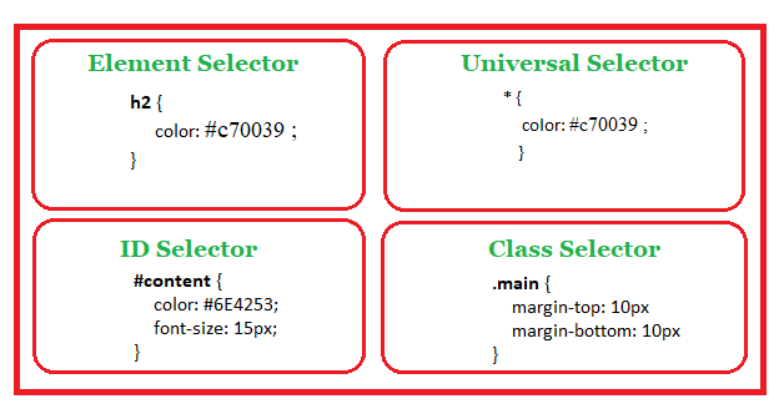


4. Sintaxis y conceptos básicos

4.1 Selectores CSS básicos

Los *selectores* definen qué elementos HTML serán afectados por las reglas CSS. Los más comunes son:

- **Selector de tipo (elemento):** Apunta a etiquetas HTML por nombre. Ej: `p { ... }` aplica a todos los `<p>`.
- **Selector de clase:** Antecede un nombre con punto (`.clase`) y aplica a elementos que tengan `class="clase"`. Ej: `.boton { ... }` aplica a `<div class="boton">` o `<button class="boton">`.
- **Selector de ID:** Antecede con `#` y selecciona el elemento con ese `id`. Ej: `#encabezado { ... }` refiere a `<div id="encabezado">`. (Debe ser único en la página.)
- **Selector universal `*`:** Selecciona todos los elementos. Ej: `* { margin:0; padding:0; }` remueve márgenes/paddings por defecto.
- **Selectores de atributos:** Seleccionan por presencia o valor de atributo. Ej: `a[target] { ... }` apunta a todos los enlaces con atributo `target`; o `input[type="text"] { ... }` a campos de texto.



Ejemplo de selectores simples:

```
/* Selector de tipo: aplica a todos los párrafos */
p {
    color: darkblue;
    font-size: 16px;
}
/* Selector de clase: aplica a elementos con class="resaltado" */
.resaltado {
    background-color: yellow;
    font-weight: bold;
}
/* Selector de ID: aplica al elemento con id="principal" */
#principal {
    border: 2px solid #333;
    padding: 10px;
}
```

En este ejemplo, el primer bloque pinta el texto de todos los `<p>` de azul, el segundo da fondo amarillo y negrita a cualquier elemento de clase “resaltado”, y el tercero rodea de borde el elemento con id “principal”. Los selectores pueden combinarse (p.ej. `p.resaltado` para `<p class="resaltado">`), pero deben usarse con cuidado para no complicar demasiado la especificidad (ver más abajo).

4.2 Pseudo-clases y pseudo-elementos

Las **pseudo-clases** son indicadores de estados especiales de los elementos. Algunos ejemplos comunes:

- **`:hover`:** estado cuando el cursor pasa sobre el elemento.
- **`:active`:** mientras el elemento está siendo pulsado (mouse o tecla).
- **`:focus`:** cuando el elemento (enlace, campo, botón) tiene foco (normalmente teclado).
- **`:visited`:** color de enlaces ya visitados.
- **`:nth-child(n)`:** selecciona el enésimo hijo dentro de un contenedor. Ejemplo: `li:nth-child(odd)` son los items impares en una lista.
- **`:first-child`, `:last-child`:** primer o último hijo.
- **`:disabled`, `:checked`, `:placeholder` (HTML5):** estilos de estados de formularios.

Las **pseudo-elementos** representan partes de un elemento:
- **`::before`, `::after`:** permiten insertar contenido antes o después del contenido real de un elemento. Muy útiles para decoraciones y efectos.
- **`::selection`:** aplica al área seleccionada por el usuario (texto marcado con mouse).
- **`::marker`:** apunta al marcador de listas (bullets).
- **`::backdrop`:** el fondo de un elemento en modo de pantalla completa.

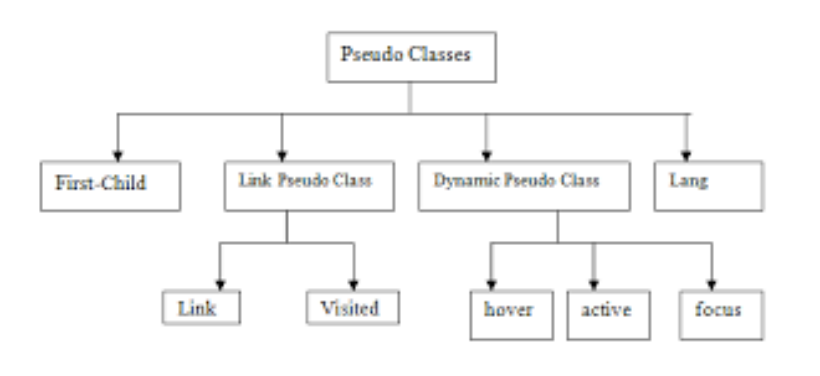
Por ejemplo:

```css
/* Pseudo-clase :hover */
a:hover {
    text-decoration: underline;
}

/* Pseudo-elemento ::before */
h2::before {
    content: "★ ";
    color: orange;
}
```

El primer bloque subraya los enlaces al pasar el cursor, el segundo inserta un símbolo "★" antes de cada `<h2>`.

Compatibilidad: Muchas pseudo-clases y -elementos están bien soportados en navegadores modernos. Algunas más nuevas como `:has()`, `:focus-visible` o `:has()` están en fase de adopción (por ejemplo `:has()` no funcionaba en la mayoría hasta 2023). Siempre es buena práctica verificar su compatibilidad o usar `@supports`/polyfills.



4.3 Especificidad, cascada y `!important`

Especificidad: Cuando múltiples reglas apuntan al mismo elemento, gana la de mayor especificidad. El cálculo es:

- Reglas inline (atributo `style`) tienen la mayor prioridad (como un “ID” especial).
- Luego se consideran los selectores del autor: cada ID cuenta como 100 puntos, cada clase/atributo/pseudo-clase 10, cada selector de elemento/pseudo-elemento 1. Ej: `#menu ul.item:hover` tendría 100 (ID) + 10 (clase) + 1 (elemento `ul`) + 10 (`:hover`) = 121.

- Si dos reglas tienen la misma especificidad, la que aparece *última en el CSS* (o cargada después) prevalece.
- La cascada también distingue el origen: estilos de usuario vs. autor vs. agente. En la práctica: los estilos en la hoja del autor prevalecen sobre los del navegador (user-agent), y los estilos inline del autor prevalecen sobre los externos.

El modificador `!important` anula la cascada normal: si se aplica, la propiedad con `!important` vence ante cualquier otra regla de menor origen o igual especificidad.

Ejemplo:

```
p { color: red !important; }
#principal p { color: blue; }
```

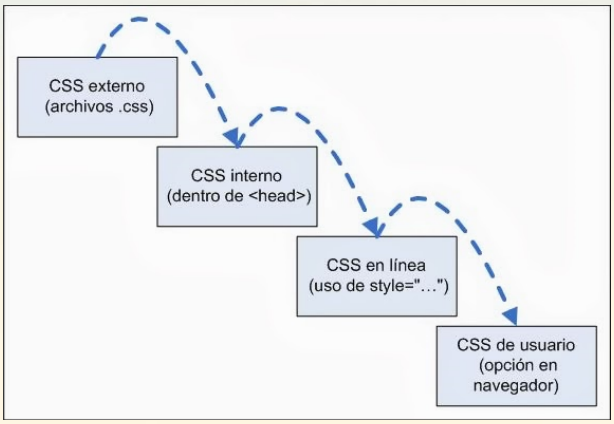
En este caso, incluso si `#principal p` es más específico, el texto en párrafos será **rojo** porque la regla `color:red` tiene `!important`. Sin embargo, abusar de `!important` dificulta el mantenimiento y raramente se recomienda usarlo salvo casos extremos (temas dinámicos, hojas de estilo de terceros, etc.).

4.4 Herencia y orden de origen

La **herencia** en CSS implica que ciertos atributos de un elemento padre (como `color`, `font-family`) se propagan a sus hijos si estos no los redefinen. Por ejemplo, si `<body>` tiene `color: darkblue`, sus párrafos e hijos usarán ese color a menos que se especifique otro. Muchos atributos relacionados con la fuente y el texto son heredables, mientras que atributos de caja (margen, borde, ancho) no lo son por defecto. Si un valor no se hereda, el elemento usa su valor inicial (por ejemplo, `border: none` por defecto).

El orden de origen en la cascada usualmente es:
1. *User agent styles* (los estilos por defecto del navegador, p.ej. `h1 {display:block}`),
2. *Estilos del usuario* (hojas de estilo personalizadas que un usuario puede cargar),
3. *Estilos del autor* (hojas CSS del desarrollador),
4. *Estilos inline del autor* (atributo `style` en el HTML).

Dentro de cada origen se aplica la especificidad y el orden de aparición. En resumen, un estilo inline definido por el autor con `!important` sería lo más fuerte, mientras que un estilo simple en la hoja del usuario-agent sin conflicto sería lo más débil. En desarrollo normal de páginas, basta con saber que los estilos definidos por el autor prevalecen sobre los predeterminados del navegador, salvo interferencia con `!important` o estilos de usuario.



4.5 Comentarios y organización

En CSS, los comentarios se escriben entre `/* ... */` y son completamente ignorados por el navegador. Sirven para aclarar código o desactivar reglas temporalmente. Por ejemplo:

```
/* Este comentario explica la siguiente regla */
.container {
    padding: 10px; /* Espacio interno de 10px */
    /* background-color: red; */ /* A usar sólo en modo debug */
}
```

Los comentarios **no pueden anidarse** (es decir, no se puede abrir otro `/* ... */` dentro de un comentario ya abierto). Es buena práctica organizarlos y usarlos para documentar secciones y decisiones de diseño. También se recomienda dividir la hoja CSS en bloques lógicos (reset, tipografía, layout, módulos) y agregar comentarios tipo cabecera para cada bloque. Un estilo organizado facilita encontrar errores y mantener la coherencia.

5. Selectores avanzados y rendimiento

Además de los selectores básicos, CSS permite selectores complejos y combinadores para afinar la selección:
- **Selectores de descendiente ()**: Selecciona todo elemento que sea descendiente de otro. Ej: `nav a` selecciona todos los `<a>` dentro de `<nav>`.
- **Selectores de hijo (>)**: Solo hijos directos. Ej: `ul > li` selecciona solo `<li>` hijos directos de `<ul>`.
- **Selectores de hermanos adyacentes (+)**: Selecciona un elemento que sigue inmediatamente a otro. Ej: `h2 + p` selecciona el primer `<p>` justo después de cada `<h2>`.
- **Selectores de hermanos generales (~)**: Selecciona todos los elementos hermanos que siguen a otro. Ej: `h2 ~ p` selecciona todos los `<p>` después de cada `<h2>` en el mismo contenedor.
- **Selectores de atributos avanzados:** Se pueden usar expresiones como `[href^="https"]` (empieza con), `[lang|="en"]` (prefijo), `[class~="btn"]` (contiene palabra), etc.

También existen selectores funcionales como `:nth-child(2n+1)` para filas impares, `:not()` para negar, `:is()` y `:where()` para agrupar selectores sin aumentar especificidad, `:has()` (consúltalo en el siguiente capítulo). Por ejemplo, `ul li:nth-child(3n+1) { color: blue; }` pinta de azul cada tercer elemento de lista (índices 1,4,7...).

Rendimiento: No es lo mismo usar un selector simple que uno complejo. Los navegadores suelen evaluar selectores de derecha a izquierda, de modo que selectores muy genéricos o con múltiples combinadores pueden ralentizar el renderizado en páginas grandes. Por ejemplo, un selector `body *` (cualquier elemento dentro de body) es muy pesado. Como consejo de rendimiento, se recomienda usar selectores más específicos y directos (p.ej. clases o IDs) siempre que sea posible. Evita el selector universal `*` para aplicar estilos generales, pues afecta todos los elementos (y fuerza al navegador a evaluar cada uno). En general, favorece `.clase`, `#id` o selectores de elemento sencillos, y minimiza el uso de selectores que requieran recorrer mucho el DOM (como combinar varios descendientes).

6. Unidades y valores

CSS admite **unidades absolutas** (px, pt, in, cm, etc.) y **relativas** (%, em, rem, vw, vh, vmin, vmax, ch, ex, fr). Cada una tiene un uso apropiado:

- **Absolutas:** `px` (píxeles), `pt` (puntos tipográficos), `cm`, etc. Se usan para medidas fijas. Por ejemplo, `1px` es un píxel de pantalla (1/96 de pulgada). Estos garantizan tamaño constante pero no adaptativo.
- **Relativas al contenedor:** `%` es relativo al contenedor padre (p.ej. ancho). `fr` es la unidad en CSS Grid que representa "fracción" del espacio disponible en la rejilla.
- **Relativas a la fuente:** `em` se basa en el tamaño de fuente del elemento padre; `rem` en la fuente del elemento raíz (`<html>`). Si la fuente base es 16px, `1rem` = 16px, `2rem` = 32px. Son útiles para tipografía fluida y elementos escalables.
- **Viewport:** `vw`/`vh` son porcentajes del ancho/alto del viewport (ventana). 1vw = 1% del ancho de la ventana. `vmin`/`vmax` toman el mínimo/máximo de `vw` y `vh`.

- **Otras unidades relativas:** `ch` equivale al ancho del carácter "0" en la fuente actual, `ex` a la altura de la letra "x". Su uso es raro, pero pueden ayudar en algunos diseños tipográficos.

Ejemplos: Para texto es común usar `rem` para que escale con la configuración de usuario. Por ejemplo:

```
html { font-size: 16px; }
h1 { font-size: 2rem; } /* 32px */
p { font-size: 1rem; }  /* 16px */
```

Para layouts fluidos, `%` y `fr` en CSS Grid o Flexbox con `flex:1` permiten que los contenedores se adapten al espacio disponible. En diseños responsivos, `clamp()`, `min()` y `max()` son funciones modernas: p.ej. `font-size: clamp(1rem, 2vw, 2rem);` asegura que la fuente nunca sea menor a 16px ni mayor a 32px, escalando en el medio. Estas funciones (CSS4) admiten expresiones avanzadas (aunque no citamos aquí especificación).

Recomendaciones de uso: Use `px` cuando necesite precisión absoluta (bordes, sombras), `rem/em` para escalabilidad, `%` para anchos de contenedores fluidos. Evite unidades absolutas en layouts responsivos. Para tipografía moderna, técnicas de tamaño fluido con `clamp()` pueden mejorar la legibilidad en diferentes pantallas.

7. Modelo de caja (Box Model) y propiedades relacionadas

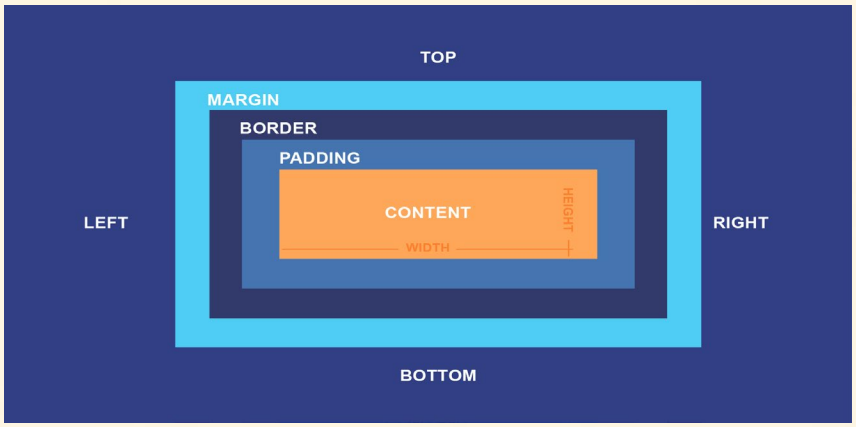
En CSS cada elemento es una *caja* compuesta por **contenido**, **padding**, **borde** y **margen**. La anchura total de la caja se calcula (por defecto) como: ancho de contenido + padding (izq. y der.) + borde (izq. y der.). El margen no se incluye en el ancho; es espacio extra fuera de la caja. Ejemplo visual:

`[margin][border][padding][content][padding][border][margin]`

La propiedad `box-sizing` controla este modelo: por defecto es `content-box` (ancho solo contenido). Si se usa `border-box`, entonces el ancho o alto establecidos incluirán padding y borde, facilitando el cálculo (p.ej. `width:100%` con `border-box` asegura que padding no exceda el contenedor). Muchos frameworks recomiendan aplicar `box-sizing: border-box;` globalmente.

Propiedades clave del modelo de caja:
- **`width`, `height`:** dimensiones del contenido (por defecto).
- **`padding`:** espacio interior entre contenido y borde.
- **`border`:** línea alrededor del padding. Se define con `border-width`, `border-style`, `border-color`.
- **`margin`:** espacio exterior entre la caja y otros elementos. Márgenes pueden colapsar verticalmente (dos márgenes de elementos adyacentes se superponen).

Propiedades extra relacionadas:
- **`overflow`:** controla qué ocurre si el contenido desborda la caja. Valores comunes: `visible` (por defecto, se muestra), `hidden` (corta el contenido sobrante), `scroll` (añade barras de desplazamiento) y `auto` (sólo barras si es necesario).
- **`visibility`:** `visible` (por defecto) o `hidden`. A diferencia de `display:none`, **`visibility:hidden`** oculta el contenido pero el elemento **sigue ocupando espacio** en el diseño.
- **`clip/clip-path`:** recorta la caja a una forma (rectángulo o SVG/clip-path). Por ejemplo, **`clip-path:`** `circle(50%)` crea un recorte circular.
- **`box-decoration-break`**: controla cómo se dibuja el fondo y borde en elementos fragmentados (p.ej. cajas multilineales). Su valor por defecto `slice` muestra un borde continuo; `clone` dibuja bordes individuales en cada fragmento.



Ejemplo práctico: Un párrafo con un borde gris, padding y margen:

```
p {
    box-sizing: border-box;
    padding: 10px;
    border: 2px solid #ccc;
    margin: 20px 0;
    width: 400px;
    overflow: auto; /* muestra scroll si el contenido es muy largo */
}
```

Este ejemplo ilustra el modelo: el contenido de 400px + 20px de padding total (10px izq + 10px der) + 2x2px de borde. Si el texto excede esos 400px, aparecerá scroll horizontal por `overflow:auto`.

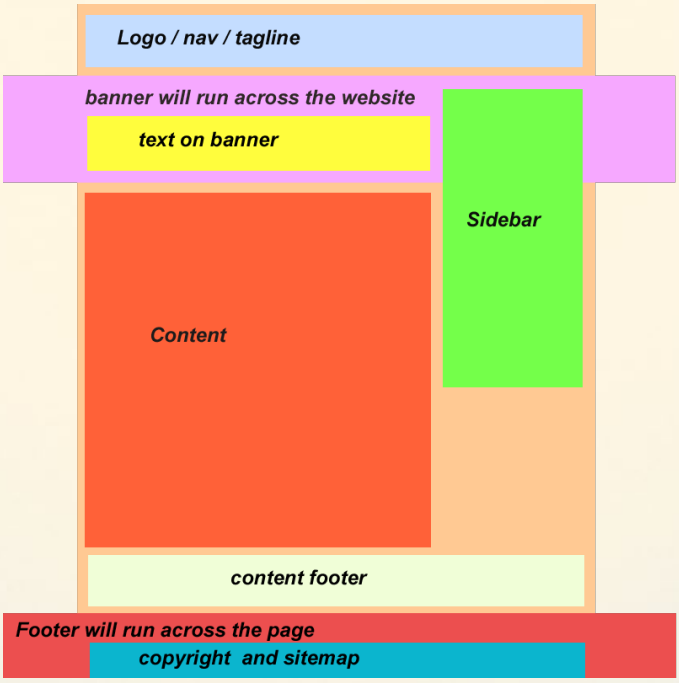
Para depurar problemas de modelo de caja, las herramientas de desarrollo (DevTools) suelen ofrecer un panel de Layout donde visualizar márgenes, bordes, padding y dimensiones de cada elemento en diferentes colores.

8. Diseño y layout

8.1 Display y flujo de elementos

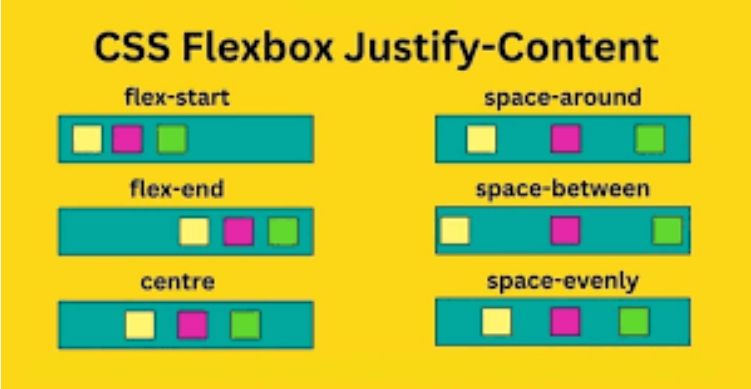
La propiedad `display` define cómo se comporta un elemento en el flujo:
- **`display: block`** (por defecto en `<div>`, `<p>`, `<h1>`): ocupa todo el ancho disponible, comenzando en nueva línea.
- **`display: inline`** (por defecto en `<span>`, `<a>`): sólo ocupa el ancho de su contenido, no inicia nueva línea.
- **`display: inline-block`**: como `inline` pero permite ancho/alto definidos.
- **`display: flex`, `display: inline-flex`:** define un contenedor flex y distribuye automáticamente sus hijos (ver Flexbox).
- **`display: grid`:** define un contenedor de cuadrícula CSS Grid, con filas/columnas explícitas.
- **`display:`** none: oculta el elemento completamente (no ocupa espacio).

Float y Clear: Antiguamente se usaban `float: left/right` para layouts, con elementos flotados permitiendo que otros fluyan a su alrededor. Hoy día está deprecado para layout principal (se reserva para envolver texto alrededor de imágenes, etc.). `clear` se usa para "limpiar" un float (p.ej. `clear: both` obliga a que un elemento aparezca debajo de elementos flotantes anteriores). En layouts modernos casi siempre usamos Flexbox o Grid en lugar de floats.



8.2 Flexbox

Flexbox es un sistema de layout unidimensional (fila o columna). Para usarlo, un contenedor define `display: flex;`. Sus principales propiedades son:
- En el **contenedor flex**:
- `flex-direction` (row/column) orienta los ítems.
- `flex-wrap` (nowrap/wrap) controla si los ítems deben saltar a nueva línea al desbordar.
- `justify-content` alinea ítems en el eje principal (ej. `flex-start`, `center`, `space-between`).
- `align-items` alinea ítems en el eje transversal (ej. `stretch`, `center`, `flex-end`).
- `gap` añade espacio uniforme entre ítems.
- En cada **ítem flex**:
- `order` define su orden en el layout (valor numérico).
- `flex` es shorthand para `flex-grow`, `flex-shrink`, `flex-basis`. Ej. `flex: 1 1 auto` permite crecer/encoger con base automática.
- `align-self` anula `align-items` para un ítem en concreto.



Ejemplo Avanzado Flex: Para centrar un elemento vertical y horizontalmente:

```
.container {
    display: flex;
    justify-content: center; /* horizontal */
    align-items: center;     /* vertical */
    height: 100vh;
}
.item {
    width: 200px;
    height: 100px;
    flex: 0 0 auto; /* no crece ni se encoge */
}
```

Este contenedor `.container` llena la altura de la ventana (`100vh`) y centra su único `.item` en el medio tanto vertical como horizontalmente.

8.3 CSS Grid

CSS Grid es un sistema bidimensional de rejilla. Un contenedor con `display:grid;` puede definir filas y columnas explícitamente. Propiedades clave:
- **`grid-template-columns`, `grid-template-rows`**: definen tamaños de columnas/filas (ej. `1fr 2fr` crea una columna doble de la otra).
- **`grid-template-areas`:** permite nombrar áreas de layout para referencia en los elementos hijos.
- **`grid-gap` o `gap`:** espacio entre filas y columnas.
- **`grid-auto-flow`:** controla cómo se colocan automáticamente los ítems que no tienen área asignada.
- **Auto-fill/auto-fit en conjunción con `repeat()`:** por ejemplo `repeat(auto-fill, minmax(200px, 1fr))` crea tantas columnas de mínimo 200px como quepan.

Subgrid: Una característica avanzada es `subgrid` (actualmente soportada en Firefox), que hereda la estructura de la rejilla del elemento padre para un elemento hijo, permitiendo más consistencia en layouts anidados.

Ejemplo Simple Grid: Tres columnas iguales, con filas automáticas:

```
.grid-container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    grid-auto-rows: auto;
    gap: 10px;
}
.grid-container > div {
    background: lightgray;
    padding: 10px;
}
```

Así todos los hijos directos de `.grid-container` se distribuyen en 3 columnas iguales, con espacio de 10px entre.

8.3.1 Comparativa Flex vs. Grid

- **Flexbox** es más sencillo para alineación en una sola dimensión (fila o columna), perfecto para barras de navegación, galerías horizontales, layouts lineales.
- **Grid** es ideal para layouts complejos bidimensionales (p.ej. la disposición general de un sitio con encabezado, menús, contenido y pie de página). Con Grid se puede posicionar elementos en filas y columnas específicas sin necesidad de estructuras HTML adicionales.

En la práctica, ambos suelen usarse juntos. Un consejo práctico: use Flexbox para componentes lineales (menú horizontal, lista de ítems que deben fluir) y Grid para la maquetación global que requiere filas/columnas.

8.4 Container Queries (Consultas de contenedor)

(Funcionalidad emergente en CSS) Las *container queries* permiten aplicar estilos basados no solo en el viewport, sino en el tamaño de un contenedor padre. En vez de `@media`, se usa `@container`. Por ejemplo:

```
.card {
    container-type: inline-size;
}
@container (min-width: 30em) {
    .card {
        /* estilos cuando el contenedor tenga al menos 30em de ancho */
        flex-direction: row;
    }
}
```

Para que esto funcione, el contenedor debe establecer `container-type` (actualmente por compatibilidad se usa `width` o `inline-size`). Luego, dentro de `@container`, se ponen reglas que dependen del ancho (o altura) del contenedor, en lugar de la ventana. Esto permite componentes verdaderamente modulares y adaptativos. (Nota: aún en proceso de adopción en navegadores; se requiere prefijos o polifills en algunos casos.)

9. Posicionamiento y stacking

9.1 Posicionamiento CSS

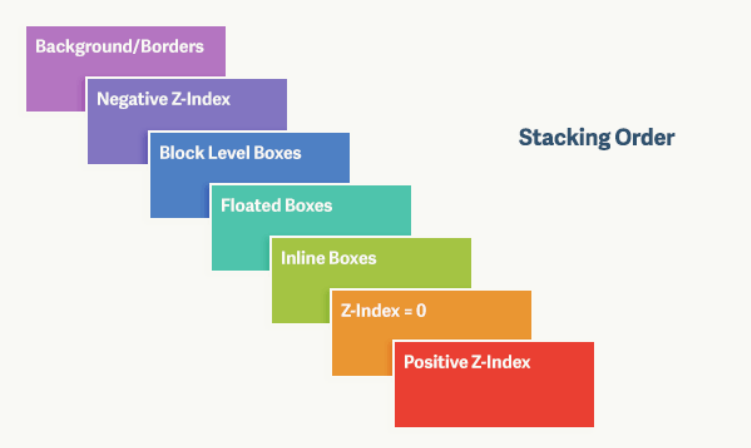
La propiedad `position` especifica cómo se posiciona un elemento:
- **`static` (por defecto):** el elemento sigue el flujo normal del documento. No acepta `top/right/bottom/left`.
- **`relative`:** permanece en el flujo, pero permite ajustar su posición relativa a donde estaría originalmente, usando `top`, `left`, etc. Otros elementos **no** ocupan su espacio cuando se mueve.
- **`absolute`:** el elemento se saca del flujo normal y se posiciona relativo a su contenedor más cercano que sea *posicionado* (que tenga `position` distinto de `static`). Si no hay, se posiciona respecto al viewport. Usa `top/left` para ubicarlo.
- **`fixed`:** similar a `absolute`, pero siempre relativo al viewport de la ventana. El elemento permanece en la misma posición incluso al hacer scroll.
- **`sticky`:** (mezcla de relative/fixed) el elemento actúa como `relative` hasta que la página se desplaza hasta cierto punto (definido con `top/left/etc`), luego se "fija" como `fixed`. Muy útil para menús que se quedan pegados al hacer scroll.

Ejemplo de `position:sticky`:

```
header {
    position: sticky;
    top: 0;
    background: white;
```

```
    z-index: 10;
}
```

Este header quedará en la parte superior al hacer scroll hacia abajo, gracias a `top:0`.



9.2 Orden Z (`z-index`) y contextos de apilamiento

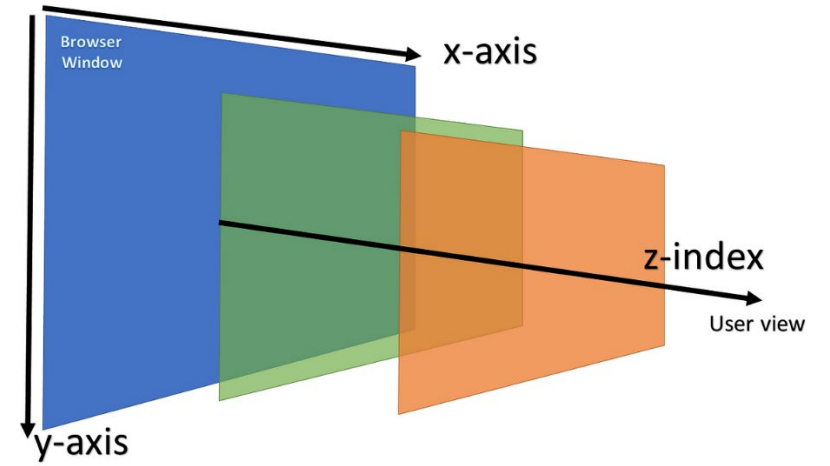
El eje Z determina qué elementos se muestran por encima de otros. La propiedad `z-index` define el orden en una capa de stacking. **Importante:** Sólo los elementos posicionados (position ≠ static) o flex/grids con `z-index` participan. Un elemento con mayor `z-index` se renderiza encima de otro con menor valor. Ejemplo:

```
.modal {
    position: fixed;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    z-index: 1000;
    background: white;
    padding: 20px;
}
.overlay {
    position: fixed;
    top:0; left:0;
    width:100%; height:100%;
    background: rgba(0,0,0,0.5);
    z-index: 500;
}
```

La `.overlay` semitransparente quedará debajo de la `.modal`, pues el modal tiene mayor `z-index`.

Los **contextos de apilamiento** son entornos aislados creados por ciertos elementos (p.ej. cualquiera con `position` y `z-index` distinto de auto crea uno, así como elementos con transformaciones, opacidad < 1, `isolation`, etc.). Dentro de un contexto, los hijos se apilan entre sí, pero nunca “rompen” hacia afuera. Por ejemplo, un elemento hijo con `z-`

`index: 100` puede cubrir a su hermano interno, pero no a un elemento en otro contexto con `z-index: 50` del mismo nivel. Los stacking contexts son conceptualmente “cajas” 3D independientes.



Como práctica, simplifica la gestión de z-index manteniendo pocos contextos y evitando valores altos arbitrariamente. Recuerda también que propiedades como `transform: translateZ(0)` crean un nuevo contexto (frecuentemente usado para forzar renderizado por GPU).

9.3 Transformaciones 2D/3D (`transform`)

La propiedad `transform` aplica transformaciones geométricas a un elemento:
- 2D: `translate`, `scale`, `rotate`, `skew`, etc. Ej: `transform: translateX(50px) rotate(45deg)`.
- 3D: incluidas `perspective`, `rotateX`, `rotateY`, `translateZ` (requiere vista con perspectiva).

Estas transformaciones se aplican sobre la caja del elemento y pueden forzar la creación de un contexto de apilamiento propio (por ejemplo, `transform: translateZ(0)` muchas veces se usa para crear un nuevo stacking context y activar GPU).

Ejemplo simple 2D:

```
.box {
    transition: transform 0.3s ease;
}
.box:hover {
    transform: scale(1.1) rotate(5deg);
}
```

Al pasar el ratón, `.box` agranda y rota suavemente. Como se verá en la sección de animaciones, transformar con `translate`, `rotate` o cambiar `opacity` suele ser más eficiente (se anima en la GPU) que cambiar propiedades que provocan repintado o reflujo (como `width` o `top`).

10. Tipografía y fuentes

10.1 Propiedades tipográficas básicas

CSS ofrece muchas propiedades para controlar texto: `font-family` (lista de fuentes y genérico), `font-size`, `font-weight` (normal/bold/400/700), `font-style` (normal/italic), `font-variant` (ligaduras, small-caps), `line-height` (altura de línea), `letter-spacing`, `text-align`, `text-decoration` (subrayado, etc.), `text-transform` (mayúsculas, etc.).

Ejemplo:

```
body {
    font-family: Arial, sans-serif;
    font-size: 16px;
    line-height: 1.5;
    color: #333;
}
h1 {
    font-size: 2rem;
    font-weight: bold;
    margin-bottom: 0.5em;
}
```

Estos estilos definen una fuente clara para todo el documento. El uso de `rem` en títulos permite escalar la tipografía relativa al tamaño base (16px). Se recomienda siempre terminar la lista de `font-family` con una familia genérica (`serif`, `sans-serif`) como respaldo.

10.2 Web Fonts y `@font-face`

Para usar fuentes no estándar, se pueden usar servicios de fuentes web (como Google Fonts) o la regla `@font-face`. Con Google Fonts, basta con incluir en el `<head>`:

```
<link href="https://fonts.googleapis.com/css?family=Roboto:400,700"
rel="stylesheet">
```

y luego en CSS: `font-family: 'Roboto', sans-serif;`. Siempre se añade un fallback genérico (p.ej. `sans-serif`).

Con `@font-face` se descargan archivos de fuente propios. Ejemplo:

```
@font-face {
    font-family: 'MiFuente';
    src: url('MiFuente.woff2') format('woff2'),
         url('MiFuente.woff') format('woff');
    font-weight: normal;
    font-style: normal;
    font-display: swap;
}
body {
```

```
    font-family: 'MiFuente', sans-serif;
}
```

Esto define la fuente “MiFuente” usando archivos `.woff2/.woff`. La propiedad `font-display: swap` (soportada en Chrome, Firefox, etc.) indica al navegador mostrar texto con la fuente de sistema hasta que la fuente web cargue, evitando invisibilidad (FOIT).

Las *variable fonts* son formatos modernos donde una sola fuente puede actuar como muchas (interpolando peso, inclinación, ancho). Se definen igual, pero usando ejes de variación (p.ej. `font-variation-settings`). No lo cubrimos aquí en detalle, pero es tendencia en tipografía.

Accesibilidad tipográfica: Se debe garantizar que los textos sean legibles: usar tamaños no menores a ~12–14px en párrafos, suficiente contraste de color (ver próximo tema), interlineado confortable (line-height alrededor de 1.4–1.6), y espaciados adecuados. Evitar fuentes decorativas difíciles de leer en texto largo. Utilizar `font-variant: normal` si ligaduras o capitals confunden al usuario, y no eliminar el enfoque de teclado de inputs sin ofrecer un indicador alternativo.

10.3 Rendimiento de fuentes

Cargar múltiples fuentes o pesos puede ralentizar el sitio. Google Fonts genera múltiples peticiones si se listan varios tipos. Un truco es combinar familias usando | en la URL, o bien alojar archivos localmente. También se puede *preload* la fuente esencial (p.ej. `<link rel="preload" href="MiFuente.woff2" as="font" type="font/woff2" crossorigin>`) para forzar su descarga temprana. La regla `font-display` mencionada (`swap`, `fallback`) es clave para no bloquear el renderizado con la carga de la fuente.

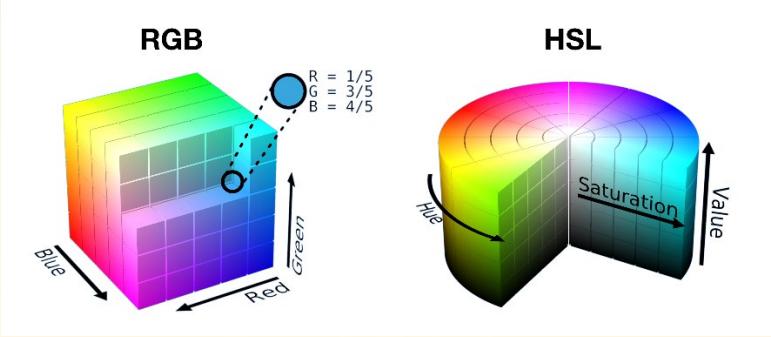
11. Colores y modelos de color

11.1 Modelos de color

CSS acepta múltiples formatos de color:

- **HEX:** formato hexadecimal `#RRGGBB` (o corto `#RGB`). Ej: `#00ff00` es verde. Se puede añadir dos dígitos extra para transparencia (`#RRGGBBAA`).
- **RGB:** función `rgb(r,g,b)`, con valores 0–255 o porcentajes. Ej: `rgb(255,0,0)` rojo. **RGBA** añade canal alpha: `rgba(255,0,0,0.5)` es rojo 50% transparente.
- **HSL:** función `hsl(hue, sat, light)`. Hue en grados (0=rojo, 120=verde, 240=azul); saturación y luminosidad en %. Ej: `hsl(240, 100%, 50%)` azul puro. **HSLA** añade alfa (opacidad).
- **Otros:** `lab()`, `lch()`, `color()`: modelos de color más avanzados (CSS Color 4), que permiten espacio de color amplio y cálculos precisos (p.ej. cambios perceptuales). Y palabras clave (`red`, `blue`, etc. existen 140 nombres estándares).

Para gradientes y fondos complejos, CSS ofrece `linear-gradient()`, `radial-gradient()`, e incluso `conic-gradient()`. También propiedades como `background-blend-mode` o el uso de capas pueden crear efectos avanzados.



11.2 Contraste y accesibilidad (WCAG)

Un aspecto crítico es garantizar **contraste** suficiente entre el texto y el fondo. Las pautas WCAG 2.1 establecen mínimos: ratio de contraste de al menos 4.5:1 para texto normal y 3:1 para texto grande (18pt normal o 14pt bold). Por ejemplo, texto negro en fondo blanco es 21:1 (muy seguro), pero gris claro en blanco sería inaceptable. Herramientas en línea (Contrast Checker) ayudan a validar estos ratios. CSS puede calcularlo con `color-contrast()` (CSS Color 4), pero su soporte es experimental. Lo habitual es elegir colores contrastantes manualmente o con ayuda.

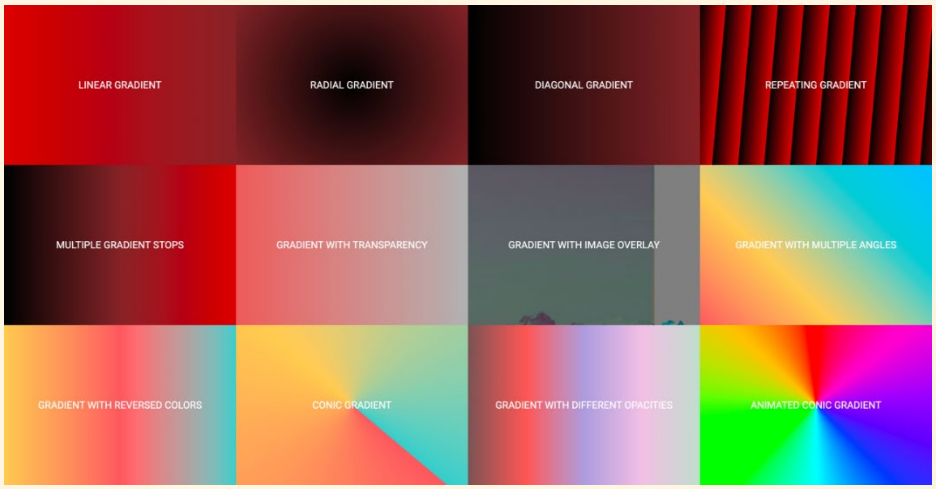
Se recomienda usar **variables CSS** para definir paletas de colores semánticas (p.ej. `--color-primary`, `--color-background`, `--color-text`) y reutilizarlos. Así se facilita mantener relaciones de color consistentes y cambiar temas (p.ej. modo oscuro) al alterar unas pocas definiciones. Por ejemplo:

```
:root {
  --color-primary: #1e90ff;
  --color-text: #333;
  --color-bg: #fff;
}
body {
  background: var(--color-bg);
  color: var(--color-text);
}
.button {
  background: var(--color-primary);
  color: var(--color-bg);
}
```

Este uso de variables aporta claridad semántica (primario, fondo, texto) y flexibilidad de theming.

11.3 Gradientes, blending y filtros

- **Gradientes:** lineales (`linear-gradient`) o radiales (`radial-gradient`) para fondos suaves. Ejemplo: `background: linear-gradient(45deg, red, blue);` crea un degradado diagonal.
- **Blending modes:** `background-blend-mode` y `mix-blend-mode` permiten mezclar colores/elementos (como en Photoshop: multiply, screen, etc.). Útil para efectos visuales.
- **Filtros CSS:** la propiedad `filter` aplica efectos visuales (blur, sepia, grayscale, saturate, drop-shadow, etc.) a cualquier elemento. P.ej. `filter: blur(5px) brightness(1.2);`. Estos efectos pueden afectar el rendimiento (porque rehacen el renderizado), pero animar filtros como `opacity` o `blur` puede lograrse con GPU. Importante: filtros *reinterpretan visualmente* el elemento sin cambiar su caja de layout (a diferencia del transform).



Ejemplo de blur:

```
.thumbnail img {
    filter: grayscale(100%); /* imagen en blanco y negro */
    transition: filter 0.3s;
}
.thumbnail img:hover {
    filter: grayscale(0); /* vuelve al color al pasar el cursor */
}
```

Este fragmento vuelve la imagen coloreada al hacer hover (porque se elimina gradualmente el filtro gris).

12. Variables CSS (Custom properties)

Las *propiedades personalizadas* (CSS variables) permiten almacenar valores reutilizables en CSS. Se declaran con un nombre que comienza con `--` (case-sensitive) y se usan con la función `var()`. Por ejemplo:

```css
:root {
    --espacio-base: 16px;
    --color-primario: #1e90ff;
}
article {
    padding: var(--espacio-base);
    color: var(--color-primario);
}
```

En este ejemplo, `--espacio-base` y `--color-primario` son variables globales (declaradas en `:root`). Luego se aplican usando `var(--nombre)`. Se puede proveer un valor de respaldo: `var(--no-existe, green)`. Si la variable no está definida, se usa el segundo parámetro.

Características importantes:
- **Alcance:** Si se definen en `:root`, la variable es global. Si se definen dentro de un selector (p.ej. `.contenedor { --color: red; }`), solo aplica a ese elemento y sus descendientes.
- **Herencia:** Las variables se heredan, por lo que pueden definirse en un contenedor y usadas en sus hijos.
- **Dinámica:** Pueden cambiarse por JavaScript (`document.documentElement.style.setProperty('--color', 'blue')`) o en media queries, lo que facilita temas y adaptaciones dinámicas[45].
- **Ventajas:** Mejoran el mantenimiento (cambiar un color en una variable actualiza todas las referencias) y claridad semántica. También permiten sistemas de diseño (design tokens) sencillos.

Temas (Theming): Un patrón común es alternar entre modos claro/oscuro redefiniendo variables:

```css
:root {
  --bg: white;
  --text: #333;
}
@media (prefers-color-scheme: dark) {
  :root {
    --bg: #222;
    --text: #ddd;
  }
}
body {
  background: var(--bg);
  color: var(--text);
}
```

Aquí el `prefers-color-scheme` del usuario (sistema operativo) define variables distintas para dark mode.

13. Interactividad: pseudo-clases, estados y formularios

13.1 Estados de formularios y controles

CSS define muchas pseudo-clases para formularios:

- **`:focus`** (y moderno `:focus-visible`) para el elemento enfocado. Debe usarse para mostrar outline claro (p.ej. `outline: 2px solid #00f;`). Nunca se debe quitar el contorno de foco sin reemplazarlo por otro indicativo, pues daña accesibilidad. `:focus-visible` aplica solo si el usuario navega por teclado (excluye click), y está en respaldo en Firefox/Chrome.

- **`:disabled`** para inputs desactivados (editar con colores grises, etc.).

- **`:checked`** selecciona casillas o radios marcados. Se utiliza en ejemplos de controles personalizados, donde escondemos el input original (`opacity:0`) y estilizamos un `<label>` asociado usando `label:before` o `:after` condicionados a `input:checked + label`.
- **`:placeholder-shown` o `::placeholder`** permiten cambiar estilo del texto placeholder de un campo.

- **`:valid`, `:invalid`** (HTML5) según validación de `pattern`, `type=email`, etc. Se pueden usar para marcar campos en rojo/verdes automáticamente.
- **`:required`, `:optional`** diferencian campos con atributo `required`.

Ejemplo básico de estado de foco accesible:

```
input:focus {
    outline: 3px solid #66afe9;
    box-shadow: 0 0 0 2px rgba(102,175,233,0.5);
}
```

Esto garantiza que al tabular por un formulario se vea claramente qué campo está activo.

13.2 Controles personalizados y accesibilidad

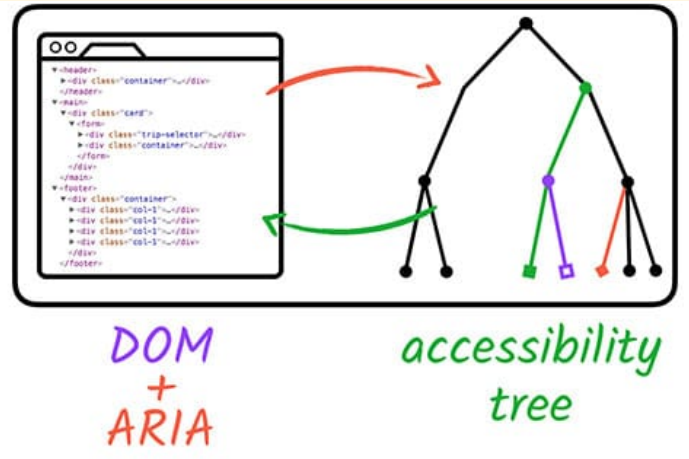
Se pueden diseñar controles propios (checkboxes, radios, select) usando HTML semántico + CSS. Por ejemplo, se ocultaría el checkbox original (`opacity:0; position:absolute;`) y se estilizaría un `<span>` o `<label>` con `:before`/`:after` que responda a `:checked`. Pero hay que tener cuidado: siempre mantener la funcionalidad del control (focus, toggle con teclado). Asegúrese de que la etiqueta `<label>` esté correctamente vinculada (`for="id"`) y de no romper la **accesibilidad ARIA**.

Un caso común: crear un *toggle* (switch) a partir de checkbox:

```
<input type="checkbox" id="toggle1" class="toggle-input">
<label for="toggle1" class="toggle-label"></label>

.toggle-input { opacity: 0; width: 0; height: 0; }
.toggle-label {
```

```
    display: inline-block;
    width: 40px; height: 20px;
    background: #ccc;
    border-radius: 10px;
    position: relative;
    transition: background 0.3s;
}
.toggle-label::after {
    content: "";
    position: absolute;
    width: 18px; height: 18px;
    top: 1px; left: 1px;
    background: white;
    border-radius: 50%;
    transition: transform 0.3s;
}
.toggle-input:checked + .toggle-label {
    background: #4CAF50;
}
.toggle-input:checked + .toggle-label::after {
    transform: translateX(20px);
}
```



Este CSS mueve el círculo al estado "encendido". El uso de `:checked + .toggle-label` colorea el fondo y desplaza la “perilla” en el label cuando el checkbox está marcado. Este control sigue siendo accesible (teclable) porque el input existe e influye en el label.

14. Transiciones, animaciones y rendimiento

14.1 Transitions

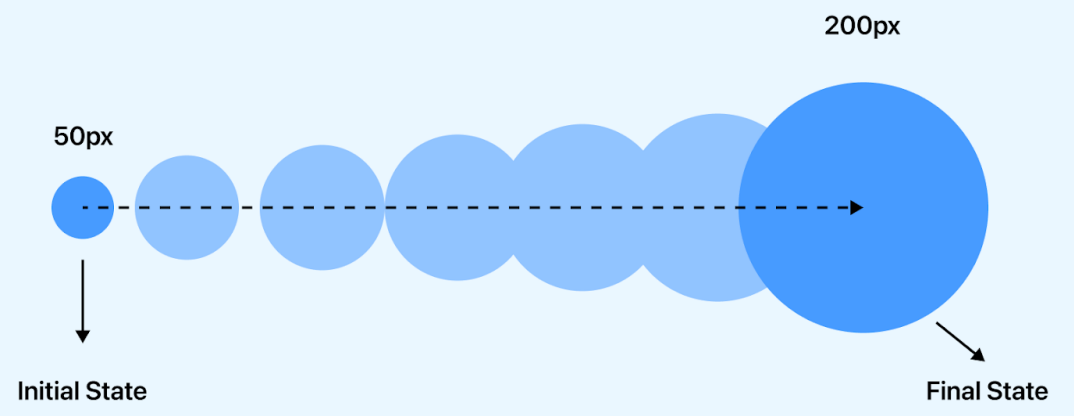
Las transiciones (`transition`) permiten animar cambios de propiedades de forma suave. Se especifican 4 componentes:

- **`transition-property`:** propiedad(s) a animar (p.ej. `width`, `background-color`, `transform`).

- **transition-duration:** tiempo que dura (p.ej. `2s`).
- **transition-timing-function**: curva de velocidad (`linear`, `ease-in-out`, `cubic-bezier(...)`).
- **transition-delay:** retraso antes de iniciar.

Ejemplo:

```
.button {
    background-color: #f0f0f0;
    transition: background-color 0.3s ease-in-out;
}
.button:hover {
    background-color: #ddd;
}
```



Al hacer hover, el fondo cambia gradualmente en 0.3s (curva ease-in-out). Se pueden usar propiedades individuales (`transition-duration: 0.3s;`) o shorthand (`transition: all 0.3s`). Se recomienda especificar solo las propiedades necesarias para minimizar trabajo del navegador.

14.2 Animations (Keyframes)

Las animaciones (`@keyframes`) definen etapas. Por ejemplo:

```
@keyframes girar {
    from { transform: rotate(0deg); }
    to   { transform: rotate(360deg); }
}
.rotar {
    animation: girar 2s linear infinite;
}
```

Este `.rotar` girará indefinidamente. `animation-name`, `animation-duration`, `animation-timing-function`, `animation-delay`, `animation-iteration-count`, `animation-fill-mode` son las partes de la animación.

14.3 Rendimiento en animaciones

Para buena performance, evite animar propiedades que obliguen al navegador a recalcular el diseño (reflow) o repintar pesadamente. Propiedades seguras de animar (con aceleración por GPU) son `transform` y `opacity`. Otras como `left`, `top`, `width`, `height` causan repintados masivos. Además, use `will-change: property` con moderación para indicar al navegador qué animará (precarga optimizaciones), pero no abusar (pisa memoria). Use también la media feature `@media (prefers-reduced-motion)` para detectar usuarios que han pedido menor movimiento en su sistema, y desactivar animaciones o hacerlas más suaves.

Ejemplo de animación con transform para mejor rendimiento:

```
.tooltip {
    opacity: 0;
    transform: translateY(-10px);
    transition: opacity 0.2s ease, transform 0.2s ease;
}
.element:hover .tooltip {
    opacity: 1;
    transform: translateY(0);
}
```

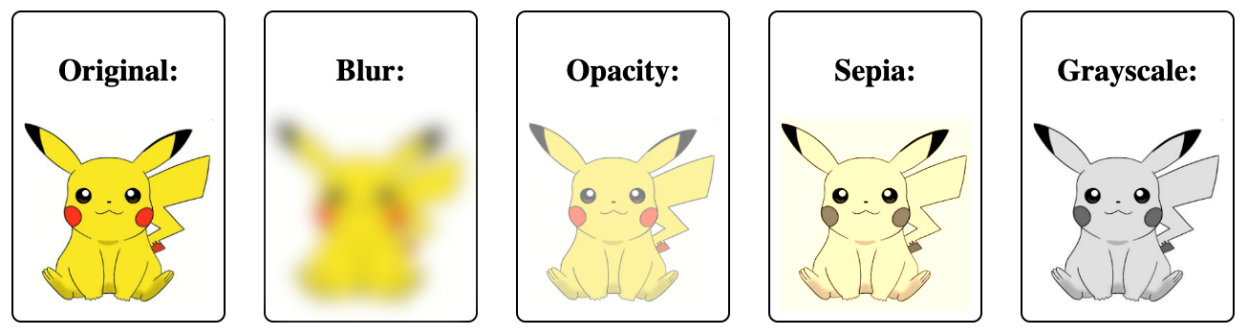
Aquí el tooltip aparece deslizándose desde arriba, animando `opacity` y `translateY` (transform) que es eficiente.

15. Filtros, máscaras y recortes

15.1 Filters (`filter`)

Ya se mencionó `filter` en el apartado de color: permite aplicar efectos gráficos como difumino (`blur()`), tonos de color (`grayscale()`, `sepia()`, `brightness()`, etc.), opacidad parcial (`opacity()`), o sombras (`drop-shadow()`). Por ejemplo, una imagen sombreada:

```
.image {
    filter: drop-shadow(5px 5px 10px rgba(0,0,0,0.5));
}
```



Estas funciones se pueden encadenar (separadas por espacios). Son útiles para efectos dinámicos sin necesidad de preprocesar imágenes. Sin embargo, cada filtro puede ser costoso en rendimiento si se anima, porque obliga a repintar el elemento.

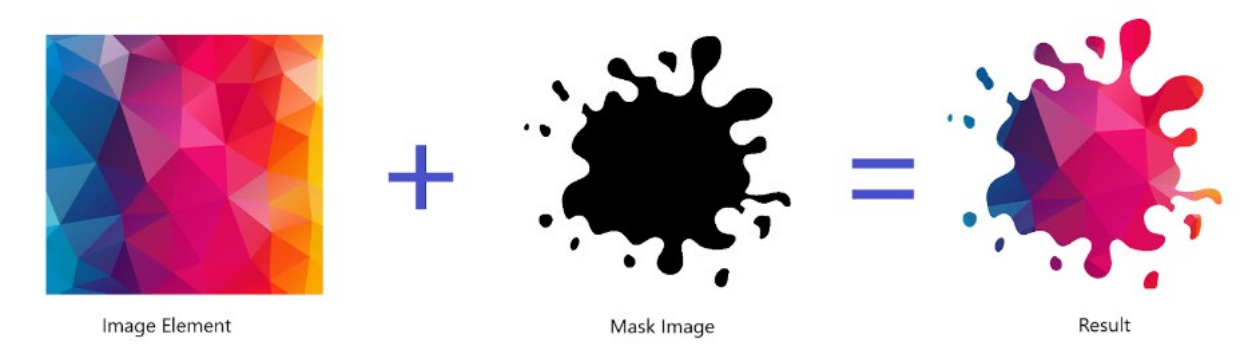
15.2 Máscaras y `clip-path`

CSS Masking permite ocultar parcialmente un elemento con otra imagen o forma. Por ejemplo, `mask-image: url(mascara.svg)` aplicaría un SVG como máscara de transparencia. Hay propiedades `mask-mode`, `mask-size`, etc. Esta técnica es relativamente compleja y con soporte limitado, así que su uso es más especializado (p.ej. para efectos visuales, iconos adaptativos).

En contraste, `clip-path` (CSS Shapes) recorta el elemento a una forma geométrica CSS o SVG. Ejemplo:

```
.photo {
    clip-path: circle(50% at 50% 50%);
}
```

Esto recorta `.photo` a un círculo (50% radio en el centro). Es útil para obtener bordes no rectangulares. Combinado con transiciones puede animarse el recorte.



Ambas técnicas (máscaras y clip-path) son potentes para efectos gráficos modernos, pero es importante proporcionar alternativas (por ejemplo, no depender de estas para mostrar información esencial).

16. Diseño responsivo y media queries

16.1 Media Queries

Las *media queries* permiten aplicar reglas CSS solo bajo ciertas condiciones (típicamente tamaño de pantalla o características del dispositivo)[50]. Se usan con `@media`. Ejemplo mobile-first:

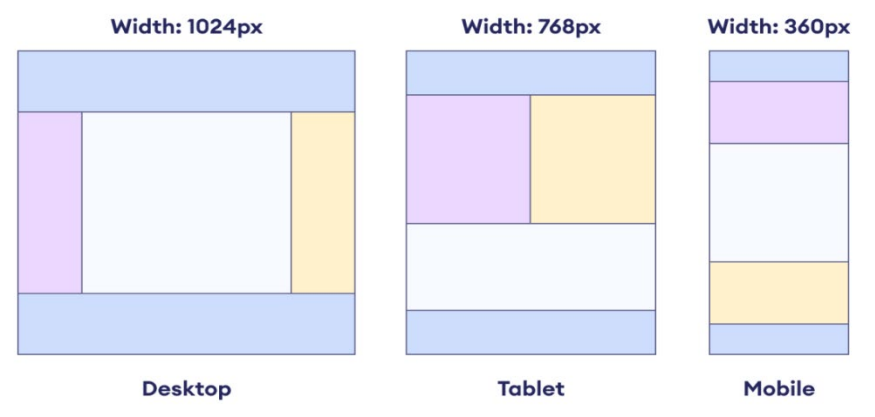
```
/* Estilos base para móvil (small) */
body {
    font-size: 16px;
    display: block;
```

```
}
/* Desde 768px en adelante (tablets y más) */
@media (min-width: 768px) {
    body {
        display: flex;
    }
}
```

En este ejemplo, se aplica un layout en bloque para móviles, y en pantallas ≥768px se cambia a flex (cambio de diseño). Se suelen usar breakpoints típicos (480px, 768px, 1024px, 1200px) según necesidades.

Además de anchura/altura (`width`, `height`), hay media features: `orientation` (portrait/landscape), `resolution` (dpi), `prefers-color-scheme` (claro/oscuro), `prefers-reduced-motion` (para detectar animación reducida), entre otras. Ejemplo para modo oscuro:

```
@media (prefers-color-scheme: dark) {
    body {
        background: #121212;
        color: #eee;
    }
}
```



16.2 Imágenes responsivas

Para reducir ancho de banda y servir la imagen adecuada, se usa el elemento `<picture>` o atributos `srcset` y `sizes`. Por ejemplo:

```
<picture>
  <source srcset="imagen-pequena.jpg" media="(max-width: 600px)">
  <source srcset="imagen-grande.jpg">
  <img src="imagen-grande.jpg" alt="Descripción">
</picture>
```

Así, si la pantalla es ≤600px, se carga `imagen-pequena.jpg`; de lo contrario, `imagen-grande.jpg`. También se puede usar `<img srcset="..." sizes="...">` para diferentes

densidades de píxel y anchos. Esto garantiza que dispositivos móviles no descarguen imágenes excesivamente grandes.

Para fondos con `background-image`, se pueden usar media queries o la propiedad `background-size` (`contain`, `cover`) para ajustar la visualización sin distorsionar la proporción.

16.3 Layouts fluidos

Además de breakpoints, se recomienda usar diseños fluidos que se adapten continuamente. Por ejemplo, usando porcentajes (`width: 50%`) o unidades relativas (`vw`). Para texto fluido, se puede usar `font-size: clamp(1rem, 2.5vw, 2rem);` para que la fuente crezca con la pantalla hasta un límite.

17. Accesibilidad (A11y) aplicada al CSS

Algunas prácticas de CSS que mejoran la usabilidad y accesibilidad:

- **Contraste adecuado:** Como se dijo, cumplir los ratios WCAG 2.1 (4.5:1 para texto estándar). Evitar texto claro sobre fondo claro, o viceversa, sin mínimo contraste (esto incluye placeholder del input).
- **Indicadores de foco visibles:** No eliminar el `outline` de los elementos enfocados. Si se personaliza el foco, asegúrese de que sea claramente visible (p.ej. borde brillante)[10]. Usuarios de teclado dependen de esto para navegar.
- **Tamaño de fuente legible:** Evite fijar texto demasiado pequeño. Generalmente ≥16px base para párrafos.
- **Touch targets grandes:** En dispositivos táctiles, elementos clicables (botones, enlaces) deben ser al menos 44–48px de área táctil. Ajustar con `padding` suficiente.
- **Estado claro (hover/focus/active):** Dar retroalimentación visual inmediata. Por ejemplo, botones cambian de color o sombra al pasar ratón (`:hover`) o al enfocarse (`:focus`). Mostrar estado activo. Esto da **affordance** clara de interactividad.
- **Saltos de navegación:** Incluir un “Skip link” al inicio (CSS oculta el enlace a menos que el enfoque esté en él) permite saltar al contenido principal. Ejemplo:

```
<a href="#main-content" class="skip-link">Saltar al contenido</a>
```

```
.skip-link {
    position: absolute;
    top: -40px; left: 0;
    background: #000; color: #fff;
    padding: 8px;
    z-index: 100;
}
.skip-link:focus {
    top: 0;
}
```

- **Largo de línea y espacios:** No abarrotar texto. Mantener longitudes de línea de 50–75 caracteres para facilitar lectura. Usar `line-height` adecuado (~1.4–1.6).
- **Evitar animaciones molestas:** Por defecto, respete la configuración `prefers-reduced-motion`. Si un usuario pide reducir movimiento, desactive animaciones innecesarias.

Anti-patrones: Evitar efectos de parpadeo, texto escondido (como `text-indent: -9999px`) sin alternativa, animaciones rápidas que puedan desencadenar epilepsia, o eliminar feedback (como esconder cursor, outlines). CSS puede mejorar accesibilidad (colores altos, visibilidad) o romperla (por ejemplo, ocultar textualmente información visual sin versión textual). Siempre verifique las WCAG relevantes al diseñar.

18. Compatibilidad entre navegadores y autoprefixing

Para garantizar que las propiedades modernas funcionen en la mayoría de navegadores, se usan:
- **Autoprefixer/PostCSS:** Herramientas de build que agregan automáticamente prefijos (`-webkit-`, `-moz-`) según la configuración de navegadores objetivo. P. ej., `display: flex` puede requerir `-webkit-box` en navegadores antiguos.
- **@supports (feature queries):** Permite probar soporte de una característica. Ej:

```
@supports (display: grid) {
    .layout { display: grid; }
}
```

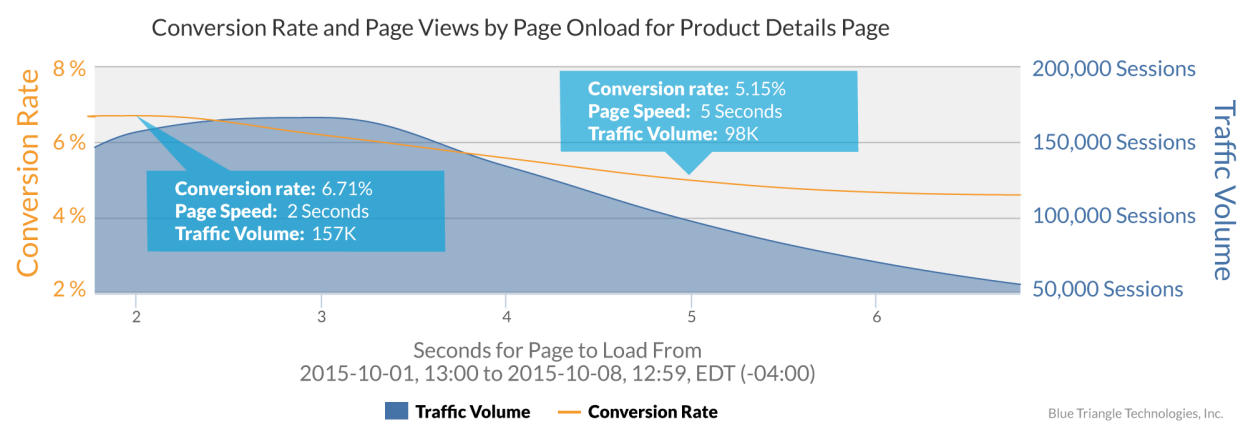
De ese modo, se puede usar Grid solo donde exista.

- **Polyfills:** Para características no soportadas, se pueden usar polifills JS (p.ej. para `:has()` se está desarrollando, o variables CSS no viejas sin soporte).
- **Degradación elegante (progressive enhancement):** Escribir estilos base simples y mejoras condicionales. Por ejemplo, estilos funcionales en caso de no soportar Grid, y un diseño más avanzado en navegadores nuevos.

Se aconseja consultar compatibilidad (MDN proporciona tablas de soporte). Evite hacks de navegador. Mantenga una lista de versiones críticas a soportar (p.ej. IE11, ciertos Android) y compruebe con herramientas como Autoprefixer qué necesita incluir.

19. Rendimiento y optimización

- **CSS crítico:** Colocar los estilos imprescindibles para el "above-the-fold" directamente o en un bloque `<style>` en el `<head>`, y cargar el resto de CSS de forma asíncrona. De este modo se renderiza rápido el contenido visible.
- **Minimizar y combinar:** Siempre combiné múltiples archivos CSS en uno solo (para reducir peticiones HTTP) y minifíquelos (eliminar espacios, comentarios) para reducir tamaño. Herramientas como cssnano o clean-css lo automatizan.
- **Evitar selectores costosos:** Como visto, seleccionadores muy generales (*), excesivamente anidados, o `:nth-child()` complejos pueden forzar más trabajo al navegador. Mantener selectores eficientes.
- **PurgeCSS / tree-shaking:** En frameworks modernos, eliminar del CSS compilado reglas no utilizadas (p.ej. clases de utilidad no referenciadas) con herramientas de purge para no cargar CSS “muerto”.
- **Cargar con `media`:** Se pueden especificar `<link rel="stylesheet" media="print">` para diferir estilos de impresión, o usar `media="(min-width: ...)"` para condicionar carga de ciertas hojas.
- **Medición:** Use Lighthouse, PageSpeed o panel de Performance de DevTools para auditar CSS. Detecte CSS no usado (Coverage tool) y muévalo a archivos secundarios. Los selectores sobrantes o estilos duplicados afectan.



Resumen de buenas prácticas: mantener selectores simples, evitar operaciones que disparen reflow (usar transform en su lugar), animar eficientemente (transform/opacity), usar hojas externas minificadas y cacheadas, evitar inline y `@import`.

20. Arquitectura CSS y patrones

Con proyectos grandes es fundamental organizar el CSS. Hay varias metodologías:

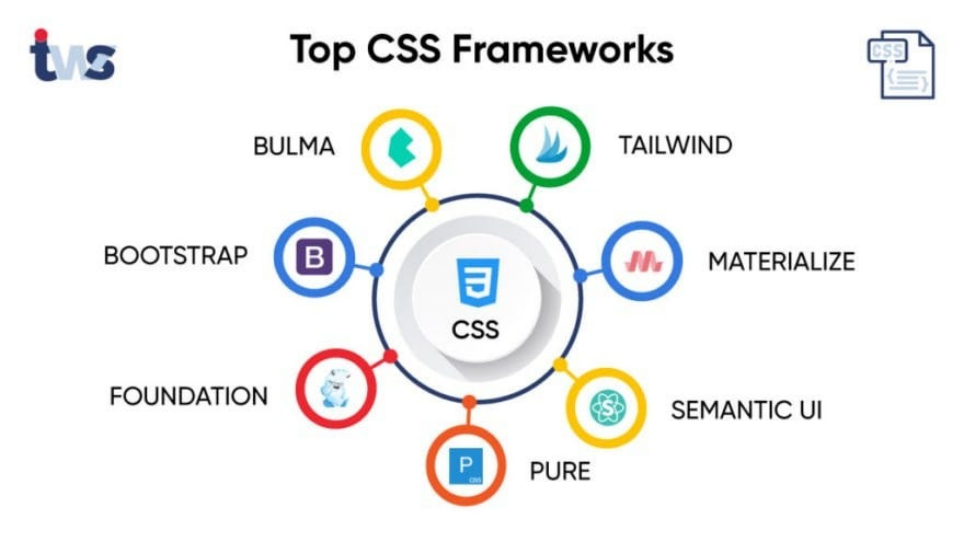
- **BEM (Block Element Modifier):** Nomenclatura para evitar conflictos: `.bloque__elemento--modificador`. Ej: `.menu__item--activo`. Facilita identificar relación en HTML y CSS.

- **OOCSS (Object Oriented CSS):** Separa estructura (contenedores reutilizables) de apariencia. Se crean clases "objetos" con responsabilidades separadas (contenedores vs skins).
- **SMACSS:** Categoriza estilos en Base, Layout, Módulo, Estado, Tema (Categories). Propone clasificar reglas CSS para mantener orden.
- **ITCSS (Inverted Triangle):** Organiza de más genérico a más específico (settings, tools, generic, elements, objects, components, trumps) en capas.
- **Atomic/CSS-in-JS:** Cada clase hace una sola cosa (p.ej. `.mt-1` un margen). Tailwind CSS es un framework utility-first que usa cientos de clases atómicas. Ventajas: muy eficiente y flexible, desventajas: HTML con muchas clases. CSS-in-JS (styled-components, Emotion) lleva esta idea a componentes JS, con scoping automático.

Cada patrón tiene pros/cons: BEM es claro para equipos grandes, pero crea clases largas; Atomic es ágil, pero puede ser verboso. Lo importante es elegir **un solo enfoque consistente** en el proyecto. Por ejemplo, con BEM la estructura de carpetas podría reflejar bloques y módulos, con CSS separando temas de componentes.

21. Herramientas y workflow

- **Preprocesadores:** SASS/SCSS y Less permiten variables, mixins y anidación en CSS, que al compilar generan CSS estándar. Ej: `@mixin` en SASS simplifica código repetido.
- **PostCSS:** Procesadores posteriores, donde plugins como autoprefixer o cssnano transforman el CSS moderno. También permite usar futuras sintaxis (CSSNext).
- **CSS Modules:** En entornos JavaScript (React, Vue), CSS Modules scopeean clases automáticamente a nivel de componente (el nombre de clase se transforma a algo único). Evita colisiones globales.
- **CSS-in-JS:** bibliotecas (styled-components, Emotion) escriben CSS dentro de JS, adjuntándolo dinámicamente. Ofrecen ventajas como temas JS, auto-remoción de estilos no usados. Pero aumenta dependencia de JS y puede penalizar SSR si no se configura bien.
- **Build Tools:** Webpack, Vite, Parcel y similares se usan para empaquetar y procesar CSS junto con JS/HTML. Pueden integrar SASS, PostCSS, etc.
- **Linters:** *stylelint* (similar a eslint para CSS) detecta errores, malas prácticas y inconsistencias según reglas definidas. Muy útil para mantener calidad de código.
- **Testing visual:** Herramientas como Percy o Chromatic (para Storybook) hacen capturas visuales automatizadas de la interfaz en varios estados y comparan versiones para detectar regresiones de estilo.
- **CI/CD:** Incluir pasos en la integración continua para validar CSS: ejecución de linter, tests unitarios/visuales, compilación, etc. Esto asegura que cambios en CSS no rompan el diseño.



22. Internacionalización y escritura bidireccional

CSS ofrece propiedades lógicas que se adaptan a la dirección del texto:

- Propiedades como `margin-inline-start`, `margin-block-end` en lugar de `margin-left/margin-top`, hacen que el layout respete `direction: rtl` sin código duplicado. Por ejemplo, en modo RTL, `margin-inline-start` corresponderá al lado derecho.

- `writing-mode` permite textos verticales o controles de flujo internacional (por ejemplo, `writing-mode: vertical-rl;`).

- La propiedad `direction` y el atributo `dir="rtl"` en HTML cambian la dirección base; combinado con propiedades lógicas (inline/block), el diseño se invierte correctamente.

- Para guiones: `hyphens: auto;` permite cortar palabras al final de línea según reglas de idioma. En CSS3, la propiedad `line-break` controla reglas de corte en texto no alfabético (p.ej. chino).

- También existen propiedades de soporte: `text-align: start/end` en lugar de `left/right`.

Un ejemplo:

```
.container {
    direction: rtl;
    padding-inline-start: 1rem; /* paddera en lado derecho en RTL */
}
```

Esto asegura que el sitio sea natural en idiomas de derecha a izquierda (árabe, hebreo). Aplicar `lang` y `dir` correctos en HTML, y usar propiedades lógicas, facilita la internacionalización.

23. Print styles y medios no-pantalla

Para estilos en impresión, se usan `@media print { ... }` o el at-rule `@page`. En estos bloques se puede ajustar diseño: ocultar elementos no relevantes (`.noprint { display:none; }`), reducir colores, mejorar legibilidad. Ejemplo:

```
@media print {
    nav, .ads, footer { display: none; }
    body { font-size: 12pt; line-height: 1.2; }
    a::after { content: " (" attr(href) ")"; } /* mostrar URLs */
}
```

`@page` permite controlar márgenes de página, orientación (`@page { size: A4 landscape; margin:2cm; }`). También se pueden forzar saltos de página con `page-break-before: always;` en títulos importantes.

Para otros medios como speech (`@media speech`) o braille, CSS ofrece `@media speech` y propiedades ARIA, pero en práctica CSS aural es menos común. Lo esencial es asegurar impresión legible con imágenes de fondo opcionalmente removibles, texto legible y enlaces marcados (como se mostró, añadiendo URL con `::after`).

24. Depuración y herramientas de desarrollo

Las DevTools de navegadores son fundamentales:

- **Inspección:** Permite ver en vivo el CSS aplicado a cualquier elemento, incluyendo cascada y especificidad. Útil para entender qué reglas están activas o si hay conflictos.

- **Computed/Styles panel:** Muestra todas las propiedades calculadas, incluso si no están explícitamente en CSS.

- **Box Model visual:** Muchas DevTools muestran un diagrama de la caja (content, padding, border, margin) y sus dimensiones en colores. Facilita ver espaciados y ajustar `box-sizing`.

- **Coverage tool:** Identifica CSS no usado en la página actual. Ayuda a eliminar código innecesario en producción.

- **Performance/Timeline:** Con solapa de **Performance** se puede grabar la carga y detectar si el parseo de CSS enlentece el inicio, o si hay repaints costosos.

- **Responsive Design Mode:** Emula varios tamaños de pantalla para probar responsividad desde la misma ventana.

Técnicas para encontrar fugas o conflictos:

- Comentar temporalmente secciones del CSS para aislar problemas.

- Revisar `z-index`: la solapa **Layers** (Chrome) muestra visualmente apilamiento de contexto.

- Verificar `@import`: DevTools Network muestra cómo se cargan hojas (p.ej. evitar `@import` tras cargar la página, pues genera retraso).

- Buscar en el inspector elementos con `outline:0` o `display:none` que no esperábamos (scripts a veces inyectan estilos inline).

La herramienta de búsqueda en CSS (pestaña Sources) permite buscar texto dentro de todos los archivos CSS cargados. Es útil para localizar definiciones de clases o detectar si usamos selectores larguísimos.

25. Nuevas APIs y tendencias

- **Houdini:** Conjunto de APIs emergentes para extender el navegador CSS:
- **Typed OM:** Manejo de valores CSS como objetos JS en scripts (permite leer propiedades como `getComputedStyle(element).fontSize`).
- **Properties & Values API:** Definir propiedades CSS personalizadas con tipos y animables, para variables más seguras y con tipado.
- **Animation Worklet:** Personalizar animaciones en JS a nivel de compositor. Estas APIs (Houdini) son experimentales y requieren configuración especial o polyfills, pero ofrecen gran poder (podrías crear tu propio *gradient* sintético en JS que se renderiza como CSS).

- **Container Queries:** Ya mencionadas (capítulo 8.4). Cada vez más navegadores las implementan nativamente (Chrome 105+, Safari 16+). Permiten estilos adaptativos a contenedores en tiempo real.

- **Selector `:has()`:** A veces llamado “parent selector”. Permite aplicar estilo al padre si coincide con un selector dentro. Ej: `div:has(> img)` selecciona un `div` que tiene una imagen hija directa. Puede mejorar CSS dinámico, pero debe usarse con cuidado por su costo de rendimiento (requiere verificación interna). Ya disponible en Chrome 105+, Safari 15.4+.

26. Referencias APA

- Refsnes, R. (2025). *CSS Tutorial*. W3Schools. Recuperado de https://www.w3schools.com/css/default.asp.
- Mozilla Contributors. (2025). *Cascading Style Sheets (CSS) – MDN*. MDN Web Docs. Recuperado de https://developer.mozilla.org/es/docs/Web/CSS.