Tema 2: CSS3

Parte 2: CSS

BLOQUE 1: Desarrollo web en cliente - HTML y CSS

Módulo: Desarrollo de Interfaces Web
Ciclo: Desarrollo de Aplicaciones Web
Curso: 2023 – 2024
Profesora: Rosa Medina

Índice

Una vez llegados a este punto donde hemos repasado los contenidos de HTML y CSS vistos en el curso pasado, ha llegado el momento de aprender las nuevas características y funcionalidades que tiene CSS y que antes si queríamos beneficiarnos de algunas de ellas teníamos que usar preprocesadores como LESS o Sass.

1. Funciones matemáticas CSS

CSS ha introducido una serie de funciones que nos permite realizar cálculos u operaciones sencillas de forma fácil, estas funciones son:

- **calc()**: permite realizar operaciones matemáticas con dichos valores o expresión (+, -, *, /).
- **min()**, **max()**: Devuelve el valor mínimo o máximo de dicha expresión
- **clamp()**: Equivale a max(MIN, min(VAL, MAX)), donde nos devuelve un valor entre un mínimo y un máximo establecidos.
- **round()**, **mod()**, **rem()**: Función para redondear, calcular el módulo o el resto
- **sin()**, **cos()**, **tan()**: Funciones para calcular el seno, coseno o tangente
- **asin()**, **acos()**, **atan()**: Funciones para calcular el arcoseno, arcocoseno o arcotangente de un valor.
- **atan2()**: Función que calcula el arcotangente de dos valores.
- **pow()**, **sqrt()**, **hypot()**, **log()**, **exp()**: Permite realizar la potencia, raíz cuadrada, hipotenusa, logaritno o potencia de e.
- **abs()**, **sign()**: Funciones que devuelven el valor absoluto de un número o el signo de un número.
- **e, pi, Infinity, NaN**: Representa los valores matemáticos.
- **attr()**: Permite conocer el valor de un atributo desde CSS

Por tanto, tenemos en CSS funciones matemáticas de comparación, de cálculo, trigonométricas, exponenciales, así como los valores constantes matemáticos. Veamos un ejemplo ahora de alguno de ellos.

```css
.elemento{
    width: calc(10px + 1em); /* Devuelve el resultado de la suma aun teniendo diferentes unidades */
    height: 300px;
    background: blue;
}
```

Tanto la función `max`, como `min` nos permite introducir tantos parámetros como queramos evaluar cuál de todos es el máximo/mínimo. Uno de esos parámetros puede ser una operación. Ejemplo:

```css
.elemento{
    width: min(10px + 1em, 25%, 2vw);
    height: 300px;
    background: blue;
}
```

Con la función `clamp`, podemos indicar un valor específico con una unidad relativa pero a la que quieres establecer un mínimo y un máximo.

```
.element {
    width: clamp(100px, 25%, 300px);
    height: 200px;
    background: red;
}
```

El ejemplo anterior, el `width` sería como si hubiéramos puesto: `width: max(100px, min(25%, 300px))`;. Primero calcula el mínimo, a continuación coge el máximo entre el 100px y el resultado anterior.

A continuación, vamos a ver la función `round`, a esta función le pasaremos el valor y el redondeo.

```
.element {
    width: round(124.75px, 1px); /* Devuelve 125px */
    width: round(124.75px, 10px); /* Devuelve 120px */
    width: round(124.75px, 100px); /* Devuelve 100px */
    width: round(124.75px, 0.1px); /* Devuelve 124.7px */
}
```

El método `attr()` nos devuelve el valor de un atributo. A diferencia de los anteriores, éste no es un método matemático ya que nos devuelve el valor de un atributo HTML. Ejemplo

```
blockquote {
    margin: 1em 0;
}

blockquote::after {
    display: block;
    content: ' (source: ' attr(cite) ') ';
    color: hotpink;
}
```

Nos va a mostrar por pantalla el valor del atributo `cite` que tenga ese `blockquote`

2. Variables en CSS

Hace unos años, la única forma que teníamos para guardar valores de CSS era mediante preprocesadores como LESS o Sass, a día de hoy tenemos las custom properties que nos permite asignar un valor a una propiedad y poderlo utilizar en múltiples partes del documento, de tal forma que si debemos cambiar su valor, se cambia 1 vez y no en tantas líneas como sitios hemos asignado ese valor a dicha propiedad.

¿Cómo declaramos una variable en css y cómo la utilizamos? Para declarar una variable en CSS utilizaremos los --previos al nombre que le vamos a dar. Además, debemos fijarnos en el elemento en el cual nos vamos a crear la variable. Veamos un ejemplo:

```css
:root {
    --main-bg-color: brown;
}
```

En el ejemplo anterior, se está usando la pseudoclase :root, es decir, a la raíz del documento, `<html>`. Con esto estamos haciendo que esta custom property esté definida para esa etiqueta html (o cualquier elemento hijo que tenga), por tanto, para todo el documento. ¿Cómo utilizo esta variable? Para ello usaremos la expresión `var()`

```css
element {
    background-color: var(--main-bg-color);
}
```

En el ejemplo anterior, al declarar la variable en root, no estamos viendo los ámbitos a la hora de definir una variable, veamos un ejemplo más interesante. Ejemplo HTML:

```html
<div class="parent">
    <div class="first child">1:</div>
    <div class="second child">2</div>
</div>
<div class="third child">1:</div>
```

Con el siguiente CSS:

```css
.parent {
    --background-color: black;
    color: white;
}

.first {
    --background-color: purple;
}

.child {
    background: var(--background-color, blue);
}
```

En el anterior ejemplo definimos la variable `--background-color`, de tal forma que los dos primeros `child` serán negro (clase `parent` e hijos), el primer elemento `child` se sobreescribe por `purple` (al aplicarse .first), y el tercer elemento `child` no va a tener ninguna variable definida (está fuera de `parent`) y por tanto pasará a tener el color `blue`.

3. Colores CSS

En el siguiente punto, vamos a ver algunas de las nuevas funciones que dispone CSS para asignar color a un elemento.

3.1. Función hwb()

La función hwb(hue whiteness blackness) es un método muy parecido al ya visto hsl pero un poco más fácil. Los parámetros que recibe son:

- **h**: ángulo o matiz de color, el valor será entre 0deg y 360deg.
- **w** que es el porcentaje de la claridad, de 0% a 100%
- **b** que es el porcentaje de la oscuridad, de 0% a 100%

Ejemplo:

```
.elemento {
    background: hwb(12 50% 0%);
    background: hwb(50deg 30% 40%);
    background: hwb(0.5turn 10% 0% / .5);
}
```

El cuarto parámetro que está recibiendo en el último `background`, corresponde con el canal alfa o transparencia y que se indica o bien con un número entre 0 y 1 o con un porcentaje entre 0% Y 100%, precedidos en ambos casos por una barra /.

3.2. Función lab() y oklab()

Cuando hablamos de LAB, hacemos referencia a un motor de color mucho más completo que los vistos hasta ahora como RGB o HSL. LAB significa: L – luminosidad de negro a blanco, A - luminosidad de rojo a verde y B luminosidad de azul a amarillo. Los parámetros por tanto que usamos con la función lab pueden ir en formato numérico o porcentaje y si le añadimos una / podemos añadir el canal alfa de transparencia.

Los parámetros de estas funciones pueden tomar los siguientes valores:

- **l** (luminosidad negro-blanco): un procentaje entre 0% y 100%, o un número entre 0 y 100
- **a** (luminosidad rojo-verde): un porcentaje entre -100% y 100%, o un número entre -125 y 125.
- **b** (luminosidad azul-amarilla): un porcentaje entre -100% y 100%, o un número entre -125 y 125.

```
.elemento {
    background-color: lab(100 125 125);
    background-color: lab(100 125 125 / 0.4);
}
```

La función oklab es una mejora de la función lab, es una función útil para:

- Transformar una imagen en escala de grises sin cambiar la luminosidad
- Modificar la saturación de los colores mientras el usuario sigue teniendo la misma percepción de color e iluminación

- Crear un gradiente de colores suave y uniforme por ejemplo cuando se interpolan en un elemento canvas.

Los valores que pueden tener los tres parámetros de la función son:

- **l** (luminosidad negro-blanco): un porcentaje entre 0% y 100% o un número entre 0 y 1.
- **a** (luminosidad verde-rojo): un porcentaje entre -100% y 100%, o un número entre -0.4 y 0.4.
- **b** (luminosidad azul-amarillo): un porcentaje entre -100%y 100% o un número entre -0.4 y 0.4

```
.elemento {
    background-color: oklab(80 0.50 0.11);
    background-color: oklab(80% 0.2 -0.2 / 0.75);
}
```

3.3. Función lch() y oklch()

La función LCH corresponde al modelo de color CIE LCH, donde se utilizan las coordenadas cilíndricas en los parámetros CH. La L corresponde a la luminosidad de blanco y negro, la C a la saturación en forma de distancia geométrica desde el eje L, y la H al tono de color en forma de ángulo. Por tanto, los parámetros que pueden tener los valores de la función lch (l c h [/alfa]) son:

- **l**: luminosidad negro-blanco dado en valores de 0% a 100% o un número entre 0 y 100.
- **c**: saturación con un valor entre 0% a 100% o un número entre 0 y 150.
- **h**: tono de color con un valor entre 0deg y 360deg o un número sin unidades

```
.elemento {
    background-color: lch(80 120 100 / 75%);
    background-color: lch(50% 132 180 / 0.75);
}
```

La función oklch es como la función *lch*() pero con una pequeña mejora, como pasaba con la función lab y oklab. Los parámetros que recibe la función lch (l c h [/alfa]) son:

- **l**: es la luminosidad negro-blanco, un valor entre 0% y 100% o un número entre 0 y 100.
- **c**: corresponde a la saturación, un valor entre 0% y 100% o un número entre 0 y 0.4
- **h**: corresponde al tono de color, un ángulo entre 0deg y 360deg o un número sin unidades.

```
.elemento {
    background-color: oklch(0.5 0.3 50);
    background-color: oklch(0.5 0.3 50 / 75%);
}
```

3.4. Función color() y color-mix()

El espacio de color tradicionalmente usado es el sRGB creado por Microsoft y HP. Un espacio de color no es más que un conjunto de colores a mostrar en un medio, por ejemplo SRGB es un conjunto de colores con diferentes valores RGB. Un modelo de color es una forma de describir un color, por ejemplo RGB es usando el Rojo Verde y Azul.

Los diferentes espacios de colores definidos y soportados en web son:

- **sRGB**: Espacio de color estándar creado por Microsoft y HP
- **srgb-linear**: Espacio idéntico a sRGB pero con diferente función de transferencia de luz
- **display-p3**: Variación de Apple para pantallas, TV o portátiles modernos
- **a98-rgb**: Espacio de color creado por Adobe
- **prophoto-rgb**: Espacio de color creado por Kodak

La función color() requiere que le indiquemos el espacio de colores y luego separado por espacio los valores de cada uno de los componentes. De forma opcional se puede indicar el canal alpha precedido de /. Veamos algún ejemplo:

```
.elemento {
    background-color: color(srgb 50% 25% 75%);
    background-color: color(display-p3 50% 25% 75%);
    background-color: color(srgb-linear 50% 25% 75%);
    background-color: color(a98-rgb 50% 25% 75% /0.5);
}
```

El método color-mix() nos permite mezclar dos colores en un espacio de colores determinado (srgb, srgb-linear, lab o oklab, xyz, xyz-d50, xyz-d65, hsl, hwb o lch o oklch). Además podemos indicar la cantidad de cada color, en el caso de no hacerlo estamos diciendo que sea 50% del primer color y 50% del segundo.

```
.elemento {
    background: color-mix(in xyz-d50, red 25%, blue 75%);
    background: color-mix(in lch, plum, pink);
    background: color-mix(in lch, plum 40%, pink);
    background: color-mix(in srgb, #34c9eb 20%, white);
    background: color-mix(in hsl longer hue, hsl(120 100% 50%) 20%, white);
    background: color-mix(in oklch longer hue, red 40%, blue 60%);
    background-color: color-mix(in srgb, #34c9eb, white);
    background-color: color-mix(in srgb, #34c9eb 75%, white);
}
```

Si usamos espacios de colores polares (hsl, hwb o lch o oklch), entonces podemos indicar el método de interpolación. Usando las palabras longer, increasing/longer o decreasing/shorter siempre seguidas de hue nos permitirá definir los valores de la rueda de color interpolados.

4. Unidades CSS

Al trabajar con CSS podemos observar la multitud de unidades que utilizamos, ya sea px, %, deg, etc. Vamos a ver los tipos de unidades que podemos utilizar en CSS.

4.1. Unidades absolutas

Las unidades absolutas son las más comunes para nosotros, ya que son iguales a una unidad de medida física. La más conocida para nosotros es el píxel (px, 1px = 0.26mm), pero además del píxel tenemos muchas otras como in (pulgadas, 1in = 25.4mm), cm (centímetros), pc (picas, 1pc = 4.23mm), mm (milímetros), pt (puntos, 1pt = 0.35mm), Q (cuarto de mm, 1Q = 0.248mm). La mayoría de estas unidades no suelen usarse a día de hoy a excepción de px. Es importante tener en cuenta que los px son útiles cuando empezamos a aprender CSS, pero poco a poco debemos empezar a ir utilizando unidades relativas o las unidades viewport que veremos a continuación.

4.2. Unidades relativas

Las unidades relativas son un tipo de unidades más potentes a las absolutas, y son ideales para trabajar en dispositivos con diferentes tamaños ya que nos vana ahorrar bastante trabajo. Las más conocidas son las siguientes: **%** (relativa a herencia, al elemento padre), **em/rem** (relativo al font-size en ese elemento), **ex/rex** (relativo a la altura del carácter x minúscula, **ch/rch** (relativo al ancho del carácter 0=, **cap/rcap** (relativo a la altura del primer carácter en mayúscula), **lh/rlh** (relativo al inline-height en ese elemento)

Cuando utilizamos la unidad **%** debemos tener en cuenta que depende del padre, y por tanto si el padre tiene un tamaño de 200px el hijo va a tener la mitad si le damos un width del 50%, pero si el padre no tiene una propiedad width definida, el elemento hijo tendrá un valor de 0.

La unidad **em** se utiliza para definir un tamaño en base a la tipografía que tenemos establecido en el navegador por defecto (tamaño). Por ejemplo, si tenemos puesta como tamaño de fuente 16px, 1em equivaldría a 16px, y 2em al doble 32px. Es importante que la propiedad font-size se hereda, y por tanto si a un elemento padre le damos un size, este tamaño se heredará en los hijos si queremos que sea proporcional al padre.

La unidad **rem** (root em), al estar fijado en root el tamaño del elemento padre, eso quiere decir que todo va a depender del tamaño del html, es decir, si nosotros le damos al html un tamaño de 22px (por ejemplo usando la pseudoclase *:root*), y queremos que el resto de elementos tengan un tamaño en relación al del html usaremos la unidad rem.

La unidad **ex** puede que sea una de las menos utilizadas, al hacer referencia al tamaño de alto de la primera letra minúscula de la tipografía. De la misma forma que vimos con em y rem tenemos **rex**, que hace referencia a la primera minúscula pero del elemento padre (html, raíz del documento o root). La unidad **cap**, al contrario que **ex** hace referencia a tamaño de la primera letra mayúscula de la tipografía, y también tenemos rcap que funciona con el mismo patrón que **rex**.

Por último, la unidad **lh** corresponde al tamaño en función del interlineado de la tipografía, es decir, viene definido en función de la propiedad *line-height*. rlh corresponde al valor que tiene el elemento padre (html, root).

4.3. Unidades que dependen del viewport

Cuando utilizamos estas unidades, estamos indicando que van a depender de la región que es visible de la web en el navegador (viewport) y por tanto que los elementos

tengan valores proporcionales al tamaño de la ventana. Estas unidades vienen precedidas por la letra **v**, haciendo referencia a un % del tamaño total de la ventana. Ejemplo:

vw (viewport width, donde 1vw = 1% del ancho del navegador), **vh** (viewport hegiht, donde 1vh = 1% del alto del navegador), **vmin** (viewport mínimo, donde 1vmin = 1% del mín(ancho, alto) del navegador), **vmax** (viewport máximo, donde 1vmax = 1% del max(ancho, alto) del navegador), **vi** (viewport inline de vw y/o vh), **vb**(viewport block de vw y/o vh). Veamos algún ejemplo:

```
.container {
    width: 50vw; /* Si el navegador ocupa 800px de ANCHO, estamos diciendo que el contenedor
                    ocupe 400px de ANCHO*/
    height: 50vh; /* Si el navegador ocupa 800px de ALTO, estamos diciendo que el contenedor ocupe
                     400px de ALTO*/
}

.container {
    width: 50vmin; /* Si el navegador ocupa 800x600(ancho x alto), estamos diciendo que el
                      contenedor ocupe de ancho la mitad de 600, es decir 300px*/
    height: 50vmax; /* Si el navegador ocupa 800x600(ancho x alto), estamos diciendo que el
                       contenedor ocupe de alto la mitad de 800, es decir 400px*/
}

.container {
    width: 50vi; /* Equivale a 50vw, ya que inline es la dirección horizontal */
    height: 50vb; /* Es la dirección vertical */
}
```

Las unidades anteriores son relativas al viewport, pero si queremos que sean relativas al contenedor usaremos la letra cq delante, es decir, **cqw**, **cqh**, **cqmin**, **cqmax**, **cqi** ó **cqb**.

5. Imágenes y objetos

En ocasiones, nos interesa poder adaptar ciertos objetos como imágenes, vídeos u otros a nuestra web. Para ello, vamos a ver ciertas propiedades CSS que nos permiten modificar estos elementos y adaptarlos al estilo que deseamos.

5.1. Propiedad object-fit

Nos permitirá rellenar una imagen u objeto a su contenedor padre. Los valores que puede tomar dicha propiedad son:

- **fill**: Rellena la imagen en el espacio del contenedor padre (la estira)
- **contain**: Mantiene el aspecto natural pero conteniendo el máximo posible de la imagen
- **cover**: mantiene el aspecto natural pero cubriendo el contenedor padre, por tanto no quedarán huecos sin cubrir. (se dimensiona manteniendo su aspecto, de tal forma que si no cabe se recortará para que se ajuste)
- **none**: El contenido no se redimensiona

- **scale-down**: Similar al none o contain pero escalando hacia abajo

5.2. Propiedad object-position

Nos permite cambiar la posición donde aparece la imagen, especialmente cuando está recortada y sólo aparece un fragmento o parte de la imagen, funciona muy parecido a la propiedad *background-position*. Los valores se dan en píxeles, o lo más habitual es con porcentajes, por defecto el valor es 50% 50%, pero también podemos usar las palabras claves como top, *left*, *right*, *bottom* o *center* para indicar en qué zona queremos centrar la imagen o incluso poniendo un porcentaje tras estas palabras para ajustarla más.

Ejemplo:

```
.element img {
    object-fit: cover;
    object-position: left 20% top 25%;
}
```

5.3. Propiedad object-view-box

Con esta propiedad podemos indicar la zona que queremos que sea visible de una imagen o vídeo, o incluso “hacer zoom” si aplicamos transiciones. Para ello, debemos hacer uso de la función inset para determinar la zona que recortaremos.

Veamos un ejemplo donde le indicamos que queremos recordar el 25% en cada lado, es decir, top, right, bottom y left.

```
.element img {
    object-view-box: isset(25%);
    object-view-box: isset(25% 25% 25% 25%);
}
```

5.4. Propiedad image-rendering

Esta propiedad permite establecer para una imagen el algoritmo de escalado que se utilice. Los valores que puede utilizar image-rendering son: auto (por defecto, el navegador decide qué algoritmo de escalado utilizar), smooth (la imagen es escalada con algoritmos que priorizan la apariencia, es idóneo para fotos), high-quality (como smooth, pero con mejor calidad), crisp-edges (enfocado en mantener contrastes y bordes de imágen, ideal para bocetos), pixelated (utiliza algoritmos de “el vecino más cercano” o similares). En el siguiente enlace se puede ver cómo funciona.

5.5. Propiedad image-orientation

Con esta propiedad podemos rotar una imagen. Los valores que puede tomar image-orientation son: none (la imagen se muestra tal cual), from-image (es el valor por defecto, debemos tener en cuenta que no se modifica la imagen original).

5.6. Propiedad aspect-ratio

Con aspect-ratio podemos cambiar la proporción de aspecto entre el ancho y alto de una imagen u objeto con CSS, pudiéndonos asegurar que no se deformarán o tendrán un aspecto raro. Los valores que pueden tener esta propiedad son;

- **auto**: el navegador calcula la proporción que considera adecuada
- **width /height**: Se utiliza la porporción ancho/alto indicado
- **auto width / height**: utiliza la proporción indicada, salvo que tenga un tamaño definido
- **número**: Número decimal en lugar de poner ancho y alto

6. Propiedad opacity

Mediante esta propiedad, podemos dar una transparencia total o parcial sobre el elemento indicado y todos los elementos hijos que tiene. La propiedad opacity se puede dar o bien en procentaje (0% completamente transparente, 100% completamente visible) o bien un número del 0 al 1, donde 0 es completamente transparente y 1 completamente visible.

7. Gradientes o degradados

7.1. Gradientes lineares, linear-gradient

Los gradientes más conocidos o más fáciles son los gradientes lineales, donde tenemos una transición progresiva entre dos colores. La sintaxis es la siguiente:

- **linear-gradient (color, color, …)**: gradiente de colores hacia abajo
- **linear-gradient (dirección, color, color, …)**: gradiente con dirección específica
- **linear-gradient (dirección, color tamaño, color tamaño, …)**: gradiente indicado dónde comienza a cambiar el color
- **linear-gradient (dirección, color tamaño tamaño, color tamaño tamaño , …)**: gradiente indicando el inicio y final de cada color.

Por ejemplo, si queremos crear un degradado con los colores azul – rojo desde arriba hacia abajo sería:

```
.element {
    background-image: linear-gradient(blue, red);
}
```

El ejemplo anterior sería un ejemplo básico, vamos a ver un poco más el resto de parámetros que le podemos decir a esta propiedad.

7.1.1. Dirección del gradiente

La dirección puede ser tanto un número de grados como “una palabra clave” que nos indique la dirección. Por ejemplo, las palabras claves asociadas a los grados serían:

- to top: correspondería a 0 ó 360deg
- **to top right**: correspondería a 45deg
- **to right**: correspondería a 90deg
- **to bottom right**: correspondería a 145deg
- **to bottom**: correspondería a 180deg
- **to bottom left**: correspondería a 215deg
- **to left**: correspondería a 270deg
- **to top left**: correspondería a 325deg

```
.element {
    background-image: linear-gradient(to right, blue, red);
}
```

7.1.2.Posición del color

Por defecto, las distancias entre los colores se ajustan automáticamente, pero si queremos cambiarlas debemos poner un tamaño (en porcentaje o píxel) después del color. Este porcentaje o pixel representa el punto central del color que lo precede respecto al tamaño del gradiente completo. Veamos un ejemplo

```
.element {
    background-image: linear-gradient(to right, blue, red);
}
```

7.1.3.Inicio y final del color

Podemos establecer un punto de inicio y fin del color. Podemos declarar varias paradas de color. Por defecto, si un color tiene 0% como inicio el primer color declarado empezará ahí, del mismo modo, si el color no tiene establecido el final se considerará que parará en el 100%. Veamos tres *linear-gradient* que hacen lo mismo:

```
.element {
    background-color: linear-gradient(red 0%, orange 10%, orange 30%, yellow 50%, yellow 70%, green 90%, green 100%);
    background-color: linear-gradient(red, orange 10% 30%, yellow 50% 70%, green 90%);
    background-color: linear-gradient(red 0%, orange 10% 30%, yellow 50% 70%, green 90% 100%);
}
```

7.1.4.Propiedad repeating-linear-gradient

Si en lugar de utilizar la función *linear-gradient* usamos repeating-linear-gradient podemos hacer que el patrón de colores se repita de forma continua. El navegador se va a encargar de repetirlos hasta que no quede más espacio en el html seleccionado.

7.2. Gradientes radiales, radial-gradient

Este tipo de gradiente es un gradiente circular. Hay que indicarle la forma que queremos (circle o ellipse, por defecto será ellipse) y los colores a utilizar. Los parámetros

que puede tomar esta función son: radial-gradient([forma] [tamaño] [at ubicación], color [inicio] [fin], color [inicio] [fin], …).

El gradiente básico sería al que le indicamos sólo los colores, luego el gradiente circular o elíptico, dejando paso al gradiente con posición. Veamos unos ejemplos:

```
.element {
    /* Modalidad básica */
    background: radial-gradient(gold, red, black);
    /* Gradiente elíptico */
    background: radial-gradient(ellipse, gold 50%, red 55%, black 75%);
    /* Gradiente circular */
    background: radial-gradient(circle 200px, gold 50%, red 55%, black 75%);
    /* Gradiente circular con posición */
    background: radial-gradient(circle 400px at left, gold 50%, red 55%, black 75%);
}
```

Como podemos ver, el primer parámetro corresponde a la forma del gradiente radial que queremos, donde si es circular se le puede dar un tamaño y si es elíptica le podemos indicar su tamaño también (ancho y alto). El tamaño puede ser un valor fijo, o la palabras reservadas **farthest-corner** (hasta la esquina más lejana, valor por defecto), **closest-corner** (esquina más cercana), **farthest-side** (hasta el lado más lejano), **closest-side** (hasta el lado más cercano).

A cualquiera de las formas del gradiente le podemos indicar su posición específica, para ello haremos uso de la palabra reservada at seguido de **center**, **top**, **left**, right, **bottom**, **top left**, **top right**, **bottom left**, **bottom right**.

Indicar que como el *linear-gradient*, tenemos la posibilidad de repetir el gradiente circular, para ello haremos uso de la función repeating-radial-gradient.

7.3. Gradientes cónicos, radial-gradient

El gradiente cónico es similar al radial pero haciendo un cono visto desde arriba. Los parámetros que puede recibir la función conic-gradient([from angle at ubicación] ,color [tamaño] [tamaño], color [tamaño] [tamaño], ...).

Para indicar desde dónde queremos que comience el gradiente usaremos la palabra reservada **from** seguida de los grados (por defecto será 0deg). A continuación, se indicaría la posición donde queremos poner el centro del gradiente, para ello usaremos la palabra at seguida del lugar donde queremos ubicarlo **posX posY**, **center**, **top**, **left**, **rigfht**, **bottom**, **top left**, **top right**, **bottom left**, **bottom right**.

```
.element {
    /* Arriba, centrado en horizontal */
    background: radial-gradient(at 50% 0%, blue, black);
}
```

Del mismo modo que el linear-gradient como el radial-gradient tenemos el repeating-conic-gradient.

8. Selectores CSS

Con el objetivo de reutilizar código, es muy común el agrupar selectores separándolos por comas. Por ejemplo, imaginad que tenemos dos o más bloques con las mismas propiedades, lo corrector para no duplicar el código sería agruparlo:

```
h1{
    color: blue;
}
.especial{
    color: blue;
}
```

Dejándolo:

```
h1, .especial{
    color: blue;
}
```

A demás de esta combinación, en CSS tenemos la posibilidad de utilizar combinadores lógicos, es decir, aquellos que nos permiten seleccionar elementos que cumplen unas determinadas restricciones y funcionan como si se tratase de una pseudoclase. Estos combinadores lógicos tienen la peculiaridad que sí se les puede pasar parámetros ya que son de tipo pseudoclase funcional. Los combinadores lógicos que podemos utilizar son:

8.1. :is()

Mediante la pseudoclase :is() podemos escribir selectores complejos de forma más compacta. Esta función recibe como argumentos una lista de selectores, y selecciona cualquier elemento que pueda ser seleccionado por un selector de esa lista. Esta pseudoclase antes se llamó ~~*:matches()*~~ y ~~*:any()*~~. Veamos un ejemplo donde podemos abreviar una combinación de selectores

```
.container .list, .container .menu,.container ul {
...
}
```

Se podría abreviar por:

```
.container :is(.list, .menu, ul) { /* Donde abreviamos mucho en estos casos */
...
}
```

8.2. :where()

La pseudoclase :where() recibe como argumento una lista de selectores y selecciona cualquier elemento que pueda ser seleccionado por un selector de esa lista. Pero,¿En qué se diferencia entonces con el selector *:is()* que hemos visto justo arriba? La diferencia se encuentra con la especificidad, donde :is() tiene por especificidad su valor más alto y :where() en cambio es 0.

```
.container :is(.list, .menu, ul) { /* Donde abreviamos mucho en estos casos */
...
}
```

```
.container :is(.list, .menu, ul) { /* Especificidad (0,1,0) */
...
}
```

8.3. :has()

La pseudoclase :has() selecciona un elemento precedido, si sus hijos cumplen una serie de criterios. Veamos un ejemplo:

```
a:has(>img) { /* Se aplica sobre el enlace a sólo si en el interior existe una etiqueta img */
...
}
```

O por ejemplo

```
h1:has(+p) { /* Selecciona un encabezados h1 con un párrafo  que inmediatamente vaya seguido por un encabezado h1 de forma inmediata y aplica estilo a h1*/
...
}
```

8.4. :not()

La pseudoclase *:not()* nos permite seleccionar todos los elementos que no cumplan los criterios indicados en sus parámetros.

```
p:not(.general) {
...
}
```

En este ejemplo vemos que todos los párrafos que no pertenezcan a la clase general se les aplicará dicho estilo.

9. Reglas CSS - at-rules

Las ***at-rules*** son un tipo de declaración que nos permite indicar comportamientos especiales. Su sintaxis suele ir acompañada del carácter @, como por ejemplo *@media*. Estas reglas por lo general las podemos poner en cualquier punto de nuestros documentos css, pero hay algunas como *@import* que debemos ponerlo al principio del fichero.

Vamos a ver las distintas reglas que podemos encontrar actualmente en CSS

9.1. @import

La regla @import nos permite importar reglas desde otras hojas de estilo. Estas reglas se suelen incluir al principio del fichero. La sintaxis de esta regla puede ser de las siguientes cinco formas:

```
@import url [list-of-media-queries];
@import url media query;
@import url supports(condición);
@import url layer(nombre);
```

```
@import url layer();
```

donde la ***url*** es el recurso a importar, pudiendo ser tanto una ruta absoluta como relativa. ***media query*** es una hoja css que se importará si coincide con el media query. ***layer*** con esta sintaxis importamos la hoja css y se pone o bien en la capa nombre (si lo especificamos), o en una nueva capa anónima. ***supports*** con esta sintaxis se importa el css siempre y cuando el navegador soporte la condición indicada.

El fichero css a importar puede ir entre comillas o bien utilizando la función url() de css. Veamos un ejemplo:

```
@import url("fineprint.css") print;
@import 'custom.css';
@import "common.css" screen, projection;
@import url('landscape.css') screen and (orientation:landscape);
@import "https://developer.mozilla.org/static/css/main.11730cda.css";
```

Como podemos ver, tenemos la posibilidad de realizar unas importaciones sólo si el navegador cumple unas determinadas condiciones. En el ejemplo anterior, el archivo fineprint.css sólo se aplicará si estamos imprimiendo con el navegador en la página principal, de lo contrario no se aplica y no se descarga por tanto. La regla por ejemplo donde está el import landscape.css se realizará sólo en el caso de tener el dispositivo en la orientación horizontal o apaisada.

Podemos utilizar esta regla para en función de si el navegador soporta o no una determinada característica. Ejemplo:

```
@import url("gridy.css") supports(display: grid) screen and (max-width: 400px);
@import url("flexy.css") supports(not (display: grid) and (display: flex)) screen and (max-width: 400px);
```

La regla ***@import*** como vemos es muy útil para evitar descargar y procesar tantos archivos css que muchas veces no son necesarios.

9.2. @layer

La regla @layer nos permite declarar una capa de cascada, es decir, nos permite agrupar código CSS en el interior de una capa para a posteriori unirlo manteniendo el orden o precedencia especificado en el caso de múltiples capas.

```
@layer special {
    .item {
        color: rebeccapurple;
    }
}
```

En el ejemplo anterior, tenemos una capa llamada special, este nombre lo da el desarrollador (nosotros) y significa que a partir de ahora va a existir una capa llamada special con los estilos que hemos puesto dentro. Si el desarrollador no indica un nombre a esa capa (@layer nombreCapa), estamos creando una capa anónima y por tanto, no las podremos reordenar ni añadir más código a ellas. Si creamos varias ***@layer*** anónimas, cada una será independiente y por tanto, diferente a la otra.

La forma de crear estas capas puede ser utilizando ***@layer*** o incluso ***@import***. Vamos a ver las distintas formas:

```
@layer
@layer nombre;
@ layer nombre.subcapa;
@ layer nombre1, nombre2, nombre3...;
@ import (url) layer(nombre);
```

Cuando le damos un nombre a una capa, estamos indicando el orden de las capas (orden por el que se han definido), y por tanto será el orden por el que se va a evaluar salvo que se modifique.

Podemos cambiar como hemos dicho el orden de las capas, para ello, usaremos la regla `@layer` seguida de las capas que queremos reordenar separadas por comas y antes de ser declaradas. Ejemplo:

```
@layer base, special;

@layer special {
    .item {
        color: rebeccapurple;
    }
}

@layer base {
    .item {
        color: green;
        border: 5px solid green;
        font-size: 1.3em;
        padding: 0.5em;
    }
}
```

En el ejemplo anterior, estamos indicando el orden de los layers en la primera línea (base y special), pero en el caso de no haberlo puesto el orden sería special y base y por tanto el estilo del documento cambia. Si por ejemplo definimos dos veces un mismo layer, lo que hace es añadir (o sobreescribir) estilos a una capa ya definida.

Para anidar capas CSS será tan fácil como usar la regla @layer dentro de otra regla @layer o usar una sintaxis rápida como la de separar los nombres con punto. Por ejemplo:

```
@layer base {
    @layer special {
        .item {
            color: rebeccapurple;
        }
    }
}
```

Se puede definir de la siguiente forma:

```
@layer base.special {
    .item {
    color: rebeccapurple;
    }
}
```

Ya conocemos la regla @import y por tanto, podemos utilizarla para importar ficheros externos incorporándolos a nuestra página y añadirlos a una capa específica, con esto conseguiremos que el código externo que tenemos en varios archivos se pueda usar directamente en una capa.

```
@import url("alertas.css") layer(alerts);
```

9.3. @supports

La regla @supports de CSS nos permite usar fragmentos de código sólo cuando se cumplen ciertas condiciones. Con esta funcionalidad podemos hacer que en el caso de no soportar el navegador una característica poder aplicar un css diferente.

La sintaxis de esta regla es: @supports [not] (condiciones) donde las condiciones irán anidadas por los operadores and y or. Veamos un ejemplo:

```
@supports not (display: grid) and (display: flex) {
    .content {
        display: flex;
        justify-content: center;
    }
}
```

Los estilos de la clase `.content` que hemos usado en el ejemplo anterior serán aplicados para aquellos navegadores que no soporten la propiedad display `grid` pero sí `flex`.

Lo correcto en estos casos sería tener un css genéricos, y usar esta regla para aquellos navegadores que sí soporten ciertas características.

```
.content {
    display: inline-block;
}
@supports (display: grid) {
    .content {
        display: grid;
        grid-template-columns: 1fr;
        justify-content: center;
    }
}

@supports not (display: grid) and (display: flex) {
    .content {
        display: flex;
        justify-content: center;
    }
}
```

9.4. Minificación

Con el término minificación se entiende a la acción de eliminar comentarios, y otros caracteres (saltos de línea, espacios) de un archivo (por ejemplo css o js). El objetivo de reducir código es para conseguir descargarlo más rápido aun perdiendo legibilidad.

Cuando creamos un archivo minificado es habitual conservar el original para a posteriori utilizarlo para modificaciones, etc. teniendo por tanto siempre dos archivos. Uno llamado estilos.css por ejemplo, y otro estilos.min.css que será el que pongamos en nuestra etiqueta link.

Podemos encontrar múltiples herramientas para minificar CSS, donde además nos permite suprimir propiedades repetidas, eliminar valores inútiles, etc. Los más habituales son: CSS Nano, Clean CSS, CSSO, HTML Minifier Tertser, entre otros.

10. StyleLint

StyleLint es un tipo de herramienta linter para detectar code smell, es decir, es una herramienta encargada de comprobar posibles problemas y/o malas prácticas de nuestro código.

10.1. Características de StyleLint

Las características que nos ofrece Stylelint son las siguientes:

- Admite la sintaxis más reciente de CSS
- Corrige de forma automática algunos estilos de errores (Autofix)
- Te permite activar/desactivar aquellas reglas que desees utilizar y configurarlas según tus preferencias.
- Te permite crear tus propias reglas.

10.2. Cómo instalar StyleLint

Como requisito antes de instalar StyleLint es que tengas npm, en el caso de no tenerlo instalado del módulo Despliegue de Aplicaciones Web y/o Desarrollo de Aplicaciones en Entorno Cliente visita la guía de la web para hacerlo en función de tu SO. Para Debian /Ubuntu sería:

1. Descargamos e importamos Nodesource GPG key

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl gnupg
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | sudo gpg --dearmor -o
/etc/apt/keyrings/nodesource.gpg
```

2. Creamos el repositorio deb

```
NODE_MAJOR=20
echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg] https://deb.nodesource.com/node_$NODE_MAJOR.x
nodistro main" | sudo tee /etc/apt/sources.list.d/nodesource.list
```

3. Actualizamos e instalamos node

```
sudo apt-get update
sudo apt-get install nodejs -y
```

Una vez tengamos **npm** instalado pasaremos a instalar **StyleLint** con el comando:

```
npm install --save-dev stylelint stylelint-config-standard
```

Con el comando anterior tenemos instalado styleLint (basado en proyectos al usar la opción `--save-dev` si queremos instalarlo de manera global quitaríamos esa opción) y un paquete de configuración estándar stylelint-config-standard Para comprobar que ha ido todo correctamente ejecutaremos:

```
npx stylelint -v
```

10.3. Configuración

Nos creamos un proyecto npm en nuestro directorio de trabajo:

```
npm init -y
```

A continuación para la configuración de stylelint, lo primero que haremos será configurar nuestro linter CSS y para deberemos crearnos el archivo `.stylelintrc.json` en la raíz del proyecto (es decir, al mismo nivel que se encuentra el paquete `package.json`, fuera del src)

```
touch .stylelintrc.json
```

Dentro del archivo que hemos creado incluimos las siguientes líneas.

```
{
    "extends": "stylelint-config-standard"
}
```

Con esto le estamos diciendo que lea la configuración base del paquete que instalamos antes (stylelint-config-standard), por lo que ya debería funcionar nuestro linter con las reglas básicas que están definidas en este paquete. Podemos añadir otra configuración además del stylelint-config-standard, se pueden añadir más paquetes de configuración.

Dentro del proyecto npm que nos hemos creado, vamos a crear el directorio src con los archivos index.html e index.css por ejemplo, y en el archivo css ponemos (con los saltos de línea incluidos):

```
.container{

    font-family: Corbel;

    display: flex;
}
```

Si ejecutamos el siguiente comando:

```
npx stylelint rutaCSS
```

Vemos los “errores” que tenemos en nuestro código.

```
rosa@rosa-desktop:~/miProyectoLint$ npx stylelint src/index.css

src/index.css
 3:5   ✖  Unexpected empty line before declaration  declaration-empty-line-before
 3:18  ✖  Unexpected missing generic font family    font-family-no-missing-generic-family-keyword
 5:5   ✖  Unexpected empty line before declaration  declaration-empty-line-before

3 problems (3 errors, 0 warnings)

rosa@rosa-desktop:~/miProyectoLint$
```

Con el anterior comando lo que estamos haciendo es decirle que me muestre los posibles errores de código que podamos tener en nuestro archivo css, ya sea de indentación, etc.

Si queremos que los corrija solos, ejecutaríamos el comando anterior pero con la opción --fix, es decir:

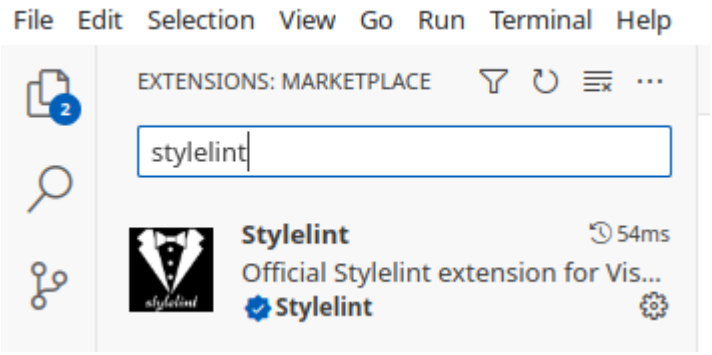
```
npx stylelint rutaCSS --fix
```

```
rosa@rosa-desktop:~/miProyectoLint$ npx stylelint src/index.css --fix

src/index.css
 2:18  ✖  Unexpected missing generic font family  font-family-no-missing-generic-family-keyword

1 problem (1 error, 0 warnings)

rosa@rosa-desktop:~/miProyectoLint$
```

Como vemos, nos corrige los errores a excepción de la fuente genérica que nos dice que pongamos la que queramos por defecto.

10.4. Integración StyleLint en VSCODE

Para integrar esta herramienta con vscode debemos instalarnos la extensión oficial de stylelint.

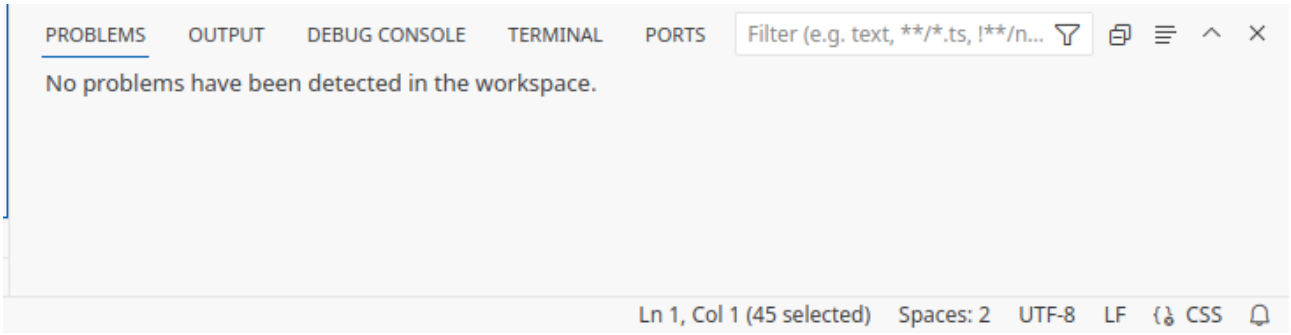


Esta extensión se comunica como si se ejecutara de forma continua el comando `npx stylelint rutaCSS`, incluso te dice la regla que está afectando si nos posicionamos sobre el error. Para configurar la opción fix en nuestro vscode pulsaremos F1 y escribimos: settings json y seleccionamos el que dice "*Abrir configuración de usuario json*".

Dentro de este archivo, al final añadiríamos:

```
"css.validate": false,
"scss.validate": false,
"less.validate": false,
"stylelint.enable": true,
"editor.formatOnSave": true,
"editor.codeActionsOnSave": {
    "source.fixAll.stylelint": true
},
```

Mediante `css.validate`, `scss.validate` y `less.validate` al estar a `false`, estamos diciendo a vscode que no use su linter en estos archivos, ya que usaremos *styleLint*. La propiedad `stylelint.enable` en `true`, estamos haciendo que StyleLint esté activado en VSCode mostrando los errores en la pestaña de "Problemas".



La propiedad `editor.formatOnSave` hace que cada vez que guardemos un fichero formatee el documento. La propiedad `editor.codeActionsOnSave` determina qué se haŕa al guardar el fichero, en nuestro caso al ponerle `source.fixAll.stylelint` a `true` intentará corregirnos los problemas automáticamente, es decir, es como si hubiéramos ejecutado el comando `npx stylelint rutaCSS` con la oppción --fix.

Hemos puesto la propiedad editor.formatOnSave en un ámbito global, eso quiere decir que se ejecutará al guardar cualquier tipo de archivo. Si sólo queremos que se formatee nuestro código al guardar un archivo css, debería ser así:

```
{
    ...
    "css.validate": false,
    "scss.validate": false,
    "less.validate": false,
    "stylelint.enable": true,
    "[css]": {
        "editor.formatOnSave": true,
        "editor.defaultFormatter": "stylelint.vscode-stylelint"
    },
    "editor.codeActionsOnSave": {
        "source.fixAll.stylelint": true
    },
}
```

11. CSS Nesting nativo

El término de Nesting podemos decir que es código CSS anidado, teniendo bloques de código unos dentro de otro y facilitando al programador el mantenimiento del nesting al ser mucho más intuitivo. Veamos un ejemplo, donde los estilos de `.item` será aplicado a todos los hijos de `.container`:

```
.container {
    width: 800px;
    height: 300px;
    background: grey;

    .item {
        height: 150px;
        background: orangered;
    }
}
```

Mediante este ejemplo, podemos ver que los estilos de `container`(clase padre) están dentro de `.item`(clase hijo) haciendo que el código quede más claro y más organizado. Vamos a ir poco a poco aprendiendo su sintaxis ya que será la misma que usaremos en Sass.

La sintaxis que hemos visto en el ejemplo anterior es la más sencilla y sirve para casos simples, pero en muchas ocasiones debemos usar el selector &.

Cuando hacemos uso del operador & , estamos haciendo referencia al padre (inmediato) del anidamiento, lo cual puede ser en ciertos casos realmente útil.

Por ejemplo, imaginad que queremos seleccionar cualquier elemento div que está dentro de container.

```
.container {
    background: grey;
    & div {
        background: orangered;
    }
}
```

En el ejemplo anterior, sin & es imposible de realizar con Nesting, ya que la sintaxis simple (sin &) no permite colocar elementos directos que **no sean clases, id, combinadores, etc.**

Veamos otro ejemplo donde es necesario usar el operador &

```
.container {
    background: grey;
    &:hover
        background: orangered;
    }
}
```

Si no ponemos el operador &, el código sería válido pero no equivalente ya que sería como poner `.container : hover` en lugar de `.container:hover`. Es decir, estaríamos aplicando los estilos a todos los elementos `:hover` que hay dentro de `.container` en lugar de al propio `.container`.

En los ejemplos anteriores, tenemos el operador & al principio, pero podemos ponerlo al final haciendo el anidamiento sobre el padre. Veamos un ejemplo:

```
.container {
    width: 800px;
    height: 300px;
    background: grey;

    .item & {
        background: green;
        height: 100px;
    }
}
```

Con este ejemplo, estamos haciendo que el selector sea `.item .container` , es decir, los elementos `.item` que contengan al padre. Además, también nos permite hacer anidaciones menos directas pero que pueden ser interesante por su relación con el elemento contenedor.

Veamos un ejemplo:

```
.container {
    width: 800px;
    height: 300px;
    background: grey;

    .item & .item & { } /* Equivalente a -> .item .container .item .container */
    :is(.page, .menu) & {} /* Equivalente a -> .page .container, .menu .container */
    :not(&) { } /* Equivalente a -> :not(.container) */
}
```

Todo esto nos va a permitir organizar y mantener mejor nuestro código.

Podemos utilizar nesting también para anidar reglas CSS como `@media`, `@supports`, `@layer`, `@scope` o `@container` consiguiendo una organización de nuestro código de forma compacta.

12. Sombras y efectos con CSS

Además de las sombras de texto (text-shadow) y las de caja(box-shadow), tenemos otras sombras bastante útiles que podemos aplicar en nuestras imágenes en este caso.

12.1. Propiedad filter

Los filtros de CSS nos permiten dar ciertos efectos a la imágenes como variación de brillo, contraste, cambiar a sepia de manera rápida en el propio navegador sin tener que retocar la imagen.

Para ello, haremos uso de la propiedad filter acompañada de la función que queremos darle.

```
filter: sepia(0.4);
```

En el ejemplo anterior estaríamos dando a la imagen a la que le aplicáramos el filtro un tono de sepia de 0.4.

12.2. Sombras idénticas CSS

La propiedad drop-shadow, más conocida como sombras idénticas es similar a la sombra que conseguimos con box-shadow. La sintaxis que se emplea es: `drop-shadow(posX posY [size] [color]);` .

12.3. Modos de fusión CSS

Cuando queremos combinar dos elementos mediante css tenemos las propiedades mix-blend-mode y background-blend-mode que nos permite mezclar colores y superponer elementos. La sintaxis de estas dos propiedades es: mix-blend-mode(normal | <blend-mode>) que aplica un modo de fusión a un elemento y mix-blend-mode(normal | <blend-mode>) que aplica un modo de fusión a un fondo. Además, tenemos la propiedad isolation(auto | isolation) que establece si un elemento debe aislarse del resto.

12.4. Formas básicas

Podemos utilizar ciertas funciones que crean formas básicas como círculos, rectángulos, polígonos, etc. las cuales nos pueden resultar útiles como recortes (clip-path) o colocaciones con la propiedad outside-shape.

Las formas básicas que tenemos son:

12.4.1. inset

La función inset crea un cuadrado o rectángulo donde cada parámetro de entrada que tiene esta función será la distancia desde su punto concreto. Si al final se le añade la palabra clave `round` estamos diciendo que se establezca un `border-radius` en cada esquina de los pixeles que le indiquemos. Veamos un ejemplo:

```
.div {
    clip-path: inset(50px round 20px);
}
```

Es equivalente a:

```
.div {
    clip-path: inset(50px 50px 50px 50px);
    border-radius: 20px 20px 20px 20px;
}
```

12.4.2. circle

Para el caso de la función circle, nos permite crear círculos, o semicirculos. Su sintaxis es `circle`(size at X Y), donde crearía un circulo con un tamaño (size) con el centro en x, y.

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

12.4.3. ellipse

En este caso, la función ellipse funciona exactamente igual que el círculo pero teniendo en cuenta el ancho (sX) y el alto (sY). Su sintaxis es `ellipse`(sX sY at x y)

```
.div {
    clip-path: ellipse(50% 25% at 50% 50%);
}
```

12.4.4. polygon

La función polygon es una de las formas básicas más potentes ya que nos permite crearla a partir de las coordenadas que le demo. Su sintaxis es `polygon`(x y, x y, ...), crean un polígono siguiendo las coordenadas indicadas.

```
.div {
    clip-path: polygon(0 5, 100% 0, 50% 100%);
}
```

12.4.5. path

Si queremos algo más flexible y potente tenemos la función path donde podemos crear un trayecto SVG haciendo formas más complejas. La sintaxis es `path`(coordenadas), Crea una forma basada en un trayecto SGV

```
.div {
    clip-path: path("m4,87l93,0l29,-84l29,84l93,0l-76,52l29,84l-76,-52l-76,52l29,-84l-76,-52z");
}
```

12.5. Propiedad shape-outside

Si queremos poner un icono o una imagen al lado de un texto como si fuera un carácter puede resultar bastante complejo sobre todo el adaptarlo. Para ello, tenemos una serie de propiedades que nos facilitan que una imagen se adapte a un texto como si estuviéramos trabajando con un editor de textos.

La propiedad shape-outside nos sirve para indicarle al navegador que ignore la parte transparente de una imagen y que así se adapte el texto a la imagen. Los valores que puede tomar esta propiedad es `shape-outside`(`none`| [<shape-box>] | [<basic-shape>] | [<image>]). Aquí tenemos algunos ejemplos de cómo podemos usarlo con las formas básicas.

Junto a esta propiedad, tenemos shape-margin que nos permite darle un margen a la imagen que por ejemplo hemos usado con la propiedad `shape-outside`. Este margen va a tener en cuenta la forma de la imagen, de tal forma que si tenía alguna forma "especial" el margen se le de teniendo en cuenta su forma. La sintaxis de esta propiedad es `shape-margin`(<length> | <percentage>)

Por último, dentro de este bloque nos encontramos con la propiedad shape-image-threshold, que nos permite indicar el umbral de transparencia (canal alfa) que debe tener cada píxel para considerarse transparente y crear la forma. Esto es muy útil para cuando no nos detecta la transparencia de una imagen. La sintaxis que tiene esta propiedad es: `shape-image-threshold`(<alpha-value>).

12.6. propiedad clip-path

Antiguamente teníamos la propiedad (hoy en día **obsoleta**) clip para hacer recortes, hoy en día ha sido reemplazada por clip-path que nos permite recortar y ocultar la parte que no queremos. La sintaxis que se emplea es: `clip-path`(`none` | <basic-shape> | <clip-source>). Cuando hablamos de `clip-source`, estamos haciendo referencia a una url que referencia a un elemento SVG. Encontramos un ejemplo usando `clip-path` aquí.

13. Responsive web design

Cuando estamos trabajando para que nuestro diseño web se adapte a todos los dispositivos, vemos que no todos tienen los mismos tamaños o características, en estos casos nos va a ser muy útil el uso de media queries.

13.1. CSS Media Queries

Gracias a las *media queries*, podemos hacer que ciertos elementos de una web no se visualicen en ciertas condiciones por ejemplo si estamos visualizando la web con el móvil. La sintaxis de una *media query* es con la regla `@media [not] (<condiciones>)`

```
@media (<condiciones>) {
    .elemento {
        ...
    }
}

@media not (<condiciones>) {
    .elemento {
        ...
    }
}
```

En el caso de cumplirse la condición que viene dada en la regla `@media` pasaría a establecerse ese estilo para `.elemento`, en el caso se establecería el estilo de la regla `@media not`. Se podría ve como el típico if-else que conocemos de programación

El parámetro `<condiciones>` que recibe la regla `@media`, nos permite una o varias condiciones utilizando los operadores lógicos `not`, `and`, `only` y `or`. El operador lógico `only` aplica para que aquellos navegadores antiguos que no "entienden" dicho operador lógico no la procesen. Por ejemplo:

```
@media only screen and (min-width: 320px) and (max-width: 480px) and (resolution: 150dpi) {
    body {
        line-height: 1.4;
    }
}
```

Los tipos de medios(tipos de dispositivos) que podemos utilizar en la regla `@media` son:

- `speech`: Lectores de texto para invidentes
- `all`: Todos los dispositivos o medios, es el que se utiliza por defecto.
- `screen`: monitores o pantallas de ordenador
- `print`: documentos de medios impresos o pantallas de previsualización antes de imprimir.

¿Qué condiciones podemos dar en los `@media`?

- width: Si el tamaño de la pantalla del dispositivo coincide con el width

- min-width:Si el tamaño de la pantalla del dispositivo tiene un tamaño de ancho mayor al indicado
- max-width: Si el tamaño de la pantalla del dispositivo tiene un tamano de ancho menor al indicado.
- height: Si el tamaño de alto del dispositivo es exactamente el height
- min-height: Si el tamaño de alto del dispositivo es mayor a lo indicado
- max-height: Si el tamaño de alto del dispositivo es menor a lo indicado
- aspect-ratio: Si el dispositivo encaja con la proporción de aspecto indicada
- `orientation`: Si el dispositivo está en modo `landscape` o `portrait`

Además de estos, podemos utilizar una modalidad llamada ***media query range syntax*** que no es más que el uso de los operadores matemáticos(=, <, <=, >, >=) que conocemos y que nos harán nuestro código más legible y fácil de interpretar.

```
@media only screen and (320px <= width <= 480px) and (resolution = 150dpi) {
    body {
        line-height: 1.4;
    }
}
```

13.1.1. Viewport o región visible

No debemos olvidarnos que la etiqueta **html** `meta name="viewport"` nos permite indicarle al navegador que vamos a usar diseño *responsive*, y que por tanto el nuevo ancho de la pantalla pasará a ser el ancho del dispositivo, y el aspecto del `viewport` se adaptará.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Por último, indicar que si queremos tener para cada dispositivo un archivo css diferente, podemos hacer uso a la etiqueta `link` en el **html** con la condición media.

```
<link rel="stylesheet" href="mobile.css" media="(max-width: 640px)">
<link rel="stylesheet" href="tablet.css" media="(min-width: 640px) and (max-width: 1280px)">
<link rel="stylesheet" href="desktop.css" media="(min-width: 1280px)">
```

13.2. CSS Container Queries

En el punto anterior hemos visto cómo modificar el estilo dependiendo el tamaño de la página o dispositivo. Con los *Container queries*, vamos a hacer lo mismo pero dependiendo del contenedor padre específico, pudiendo cambiar el tamaño de ciertos elementos y hacer que tengan una forma o estilos distintos dependiendo de su contexto.

Para utilizar esta funcionalidad,debemos determinar lo primero de todo qué elemento hará de contenedor y para ello haremos uso de la propiedad `container-name` (siempre que queramos hacer *container queries* debemos dar un nombre al contenedor para hacer referencia a él. La sintaxis que tiene esta propiedad es `container-name: nombre`. El siguiente punto será la propiedad container-type, donde permitirá establecer el tipo de tamaño de

los contenedores (normal | size | inline-size), donde usaremos size para indicar que serán elementos de bloque o inline-size para elementos en línea.

Otra opción de esta propiedad es usar `container`: nombreContenedor / tipoContenedor y nos estaríamos ahorrando las dos declaraciones anteriores.

Una vez que tenemos el contenedor definido haríamos uso de la regla `@container` de la siguiente forma: `@container nombreContenedor (<condiones>)`, como podemos observar es muy parecida a la sintaxis de los media queries.

```
.container{
    container: parent;
}
@container parent (max-width: 500px) {
    body {
        line-height: 1.4;
    }
}
```

Cuando estamos usando la regla container podemos utilizar las unidades de los container query como son: **cqw**(porcentaje relativo al ancho del contenedor), **cqh**(porcentaje relativo al alto del contenedor), **cqi**(porcentale relativo al tamaño en línea), **cqb**(porcentaje relativo al tamaño en bloque), **cqmin**(porcentaje más pequeño entre cqi y cqb) o **cqmax**(porcentaje mayor entre cqi y cqb). El objetivo es que si desconocemos el tamaño concreto del contenedor aplicamos con estas medidas un porcentaje de su tamaño. Es decir, que si ponemos 50cqw estamos diciendo que se le va a dar un tamaño del 50% de ancho del contenedor. Funcionan de forma parecida a las unidades vistas anteriormente **vw**, **vh**, **vmin** y **vmax**.

13.3. Preferencias de usuario

Dentro del punto de *media queries* tenemos la posibilidad no sólo de detectar el tamaño de la pantalla del dispositivo sino ciertas preferencias del usuario como son el tema (dark mode / light mode), evitar transiciones, reducción de datos de descarga(`prefers-reduced-data`, pero todavía no tiene soporte en los navegadores).

Dentro de la regla `@media`, tenemos al posibilidad de detectar si el usuario tiene preferencia por el tema ***dark*** o ***light*** y así poderle dar un estilo u otro a nuestra web con `prefers-color-scheme`.

```
@media (prefers-color-scheme: dark) {
    :root {
        --background-color: #333;
        --foreground-color: #eee;
    }
}
@media (prefers-color-scheme: light) {
    :root {
        --background-color: #eee;
        --foreground-color: #111;
    }
}
body{
    background-color: var(--background-color);
    color: var(--foreground-color);
```

```
}
```

Otra de las características que podemos usar dentro de las preferencias de usuario es conocer si el usuario prefiere eliminar o desactivar las animaciones o transiciones. Para ello, dentro de la regla `@media` miraremos si en `prefers-reduced-motion` tiene los valores `no-preference` o `reduce`.

```
@media (prefers-reduced-motion: reduce) {
    :root {
        --preferred-animation: soft;
    }
}

.animated {
    animation: var(--preferred-animation, hard) 2s linear infinite;
}
```

Al tener la característica de `prefers-reduced-motion` desactivada(`reduce`) estamos diciendo que la animación sea soft, o podríamos poner animation-name a none para que no haya.

14. Animaciones CSS

14.1. Transiciones

En el siguiente punto vamos a ver las transiciones, las cuales nos va a permitir dar ciertos efectos a nuestros diseños web para que sean más elegantes o agradables para el usuario final.

Las transiciones buscan tener un efecto suavizado entre dos instantes de tiempo al realizar una acción. Las propiedades que tenemos para trabajar las transiciones son:

14.1.1. transition-property

Esta propiedad CSS establece la propiedad CSS a la que se le aplicará una transición. Se puede especificar la propiedad concreta (`margin`, `color`…) o poner `all` como señal que será para todas las propiedades css. En el caso de poner `none` será que no se aplique ninguna transición. Podemos encontrar un ejemplo aquí, donde el tamaño de la fuente cambia al pasar el ratón por encima.

14.1.2. transition-duration

Duración de una transición, puede ser 0 ó un determinado tiempo indicado en segundos o milisegundos. Es importante tener en cuenta que una transición grande (larga) resulta molesta y pued que el navegador “vaya a golpes”.

14.1.3. Transition-timing-function

Corresponde al ritmo de la transición, es decir establece cómo de rápido pasarían a calcularse las propiedades a las cuales se le aplicará el efecto. Los valores que puede tomar esta propiedad son: `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, entre otros.

14.1.4. transition-delay

La propiedad transition-delay nos permite retrasar el inicio de la transición unos segundos. Si no especificamos nada en el `transition-delay` comenzaría de inmediato.

Como siempre podemos resumir o atajar estas propiedades usando la propiedad `transition` que recoge todas ellas. `transition: <property> <duration> <timing-function> <delay>`. Por ejemplo:

```
a:hover{
    transition: margin-right 2s ease-in-out .5s;
}
```

Sería igual que poner:

```
a:hover{
    transition-property: margin-right;
    transition-duration: 2s;
    transition-timing-function: ease-in-out;
    transition-delay: .5s;
}
```

En la propiedad `transition`, podemos separar por comas varias propiedades que queremos transicionar, es decir, podríamos decir que para el `hover` del enlace queremos que el `margin-right` cambie pero también por ejemplo el color de fondo. Podemos ver ejemplo de animaciones múltiples aquí.

14.2. Animaciones

Las transiciones como hemos visto es una forma de suavizar un cambio, las animaciones se busca lo mismo pero permitiendo añadir más estados. Para crear animaciones necesitamos hacer uso de la propiedad `animation` y definir con la regla `@keyframes` la animación y sus estados.

Como `transition` hay una propiedad que engloba todas las propiedades de `animation` y esta es: `animation: <name> <duration> <timing-function><delay><interation-count><direction><fill-mode><play-state>`. Veamos cada una de ellas poco a poco.

14.2.1. animation-name

Nombre de la animación a aplicar para uno o más `@keyframe`(o none en el caso de no querer aplicar ninguna animación). El nombre de la animación debemos ponerlo en kebab-case, es decir, primer carácter una minúscula y los espacios serán guiones.

14.2.2. animation-duration

Esta propiedad nos permite establecer la duración de una animación. Es importante indicarle las unidades (segundos o milisegundo).

14.2.3. animation-timing-function

Esta propiedad, nos permite establecer el ritmo que llevará la animación, de la misma forma que en las transiciones, los valores que puede tener esta propiedad los podéis encontrar aquí.

14.2.4. animation-delay

Esta propiedad nos permite establecer el retardo con el que empezará la animación

14.2.5. animation-interation-count

Con esta propiedad podemos indicar el número de veces que queremos que se repita una determinada animación, este valor puede ser un número o infinite si queremos que se repita continuamente.

14.2.6. animation-direction

Con esta propiedad podemos poner el orden en el que se van a producir los fotogramas. Los valores que puede tener esta propiedad son: **normal** (desde el primero al último), **reverse** (del último al primero), **alternate** (en interaciones par se reproducen normal, en impares como reverse), **alternate-reverse** (en interaciones par se reproducen como reverse, en impares como normal)

14.2.7. animation-fill-mode

La propiedad animation-fill-mode nos permite establecer una animación antes y después de acabar (salvo si está marcada como **infinite**). Con esto podemos establecer qué debe hacer la animación cuando no se está reproduciendo. Los valores que puede tomar son: **none**, **backwards** (la animación debe tener aplicados los estilos del fotograma inicial antes de empezar), **forwards** (la animación debe tener los estilos del fotograma aplicados al final), **both**(se aplica tanto backwards como forwards).

14.2.8. animation-play-state

Con esta propiedad podemos decirle a la animación que se pause (paused) o que se reproduzca (running). Esta propiedad tiene sentido si se combina con algo de JS.

De la misma forma que con las transiciones, podemos tener animaciones múltiples o incluso animaciones en cascada.

14.3. regla @keyframes

Dentro del apartado de animation-* debemos tener en cuenta la regla `@keyframe` que nos permitirá definir los fotogramas de la animación.

Como sabemos, una animación no es más que una secuencia de imágenes que mostradas una detrás de otra generan el efecto de movimiento. En CSS los fotogramas los generamos a partir de propiedades CSS, consiguiendo que no sea necesario definir tantos fotogramas sino que crearemos los justos para que el navegador haga el resto.

La sintaxis que se emplea en la regla `@keyframe` para definir los fotogramas es:

```
@keyframes nombre {
    from {
        propiedad: valor;
    }
    to {
        propiedad: valor;
    }
}
```

Donde el nombre será el nombre que utilicemos para identificar a los fotogramas de una animación, es recomendable utilizar la nomenclatura kebab-case. Dentro del `from`(0%) pondremos todos los estilos a aplicar antes de comenzar la animación, y dentro del `to`(100%) irá los estilos finales. La palabra reservada `from` y `to` puede ser reemplazada por el %, es decir, el valor que ocupa ese estilo dentro de la animación, al igual que podemos insertar nuevos porcentajes como señal de fotogramas intermedios.

Veamos un ejemplo sencillo donde el nombre de la animación será importante, y pasa de un fotograma donde el color (del texto en este caso) es de color rojo, y acaba siendo negro.

```
@keyframes importante {
    from { color: red; text-transform: uppercase;  } /* Primer fotograma */
    to { color: black; text-transform: none; } /* Segundo y último fotograma */
}
```

Recuerda que una vez que hemos definido el `@keyframes`, hay que asociarlo al elemento html al que se lo queremos aplicar.

```
.txt-imp{
    animation: importante 2s ease 0s infinite;
}
```

Como bien dijimos en el bloque de animación, es muy importante no pasarnos con la duración de una animación, es importante usar valores bajos como 0.25s, 0.5s o 1s.

```
@keyframes importante {
    0% { color: red; text-transform: uppercase;  } /* Primer fotograma */
    50% { text-transform: uppercase;  } /* Segundo fotograma */
    100% { color: black; text-transform: none; } /* Último fotograma */
}
```

Si tenemos fotogramas que van a utilizar los mismos estilos que uno anterior, siempre podemos separarlos por comas y así poder aprovechar ese código.

15. Transformaciones 2D/3D

Las transformaciones de CSS nos permite realizar efectos visuales 2Dy 3D. Con ellas podemos mover elementos, rotarlos, hacerlos más o menos grandes, etc.

La propiedad transform permite transformar un elemento de forma visual, pasándole como parámetro la función que queremos aplicar. La sintaxis de esta propiedad es:
`transform: none | <lista de transformaciones>;`

Por ejemplo, si queremos escalar una imagen (0.5) y transladarla, sería:

```
img {
    transform: scale(0.5) translate(-100%, -100%);
}
```

Es importante tener en cuenta, que si ponemos dos transform el último sobreescribirá al primero y por tanto el primero será ignorado. Las funciones de transformación deben ir separados por un espacio. Además, es que el orden de dichas transformaciones.

Una de las propiedades que tiene `transform` es transform-origin, que nos permite cambiar el punto de origen de una transformación. Para ello le indicaremos la posiciónX posiciónY.

15.1. Translaciones

La función de translación consiste en mover un elemento de un lugar a otro. La sintaxis de la propiedad translate es la distancia de x e y (opcionalmente z). Si queremos hacer una translación en 3D, debemos indicar también la distancia en el eje Z. Veamos un ejemplo:

```
img {
    transform: translate(10%, 10%);
    /* Es equivalente a lo de arriba */
    transform: translateX(10%) translateY(10%);
}
```

Como vemos estamos usando la función `translateX(size)`, pero podemos usar también `translateY(size)`, `translateZ(size)`, `translate(x, y)`, `translate(X)`, `translate3d(x, y, ,z)`

15.2. Rotaciones

Una rotación consiste en girar un elemento en un eje específico. Para ello usaremos la propiedad transform de css y la función `rotate`, donde del mismo modo que con `translate` podemos emplear las funciones: `rotateX(angleX)`, `rotateY(angleY)`, `rotateZ(angleZ)`, `rotate(angleZ)`, `rotate3d(x, y, z, grados)` .

```
img {
    transform: rotateX(30deg) rotateY(20deg) rotateZ(10deg);
}
```

En las nuevas versiones de los navegadores, tenemos la propiedad rotate, la cual nos permite ahorrarnos tener que usar `transform` para obtener el mismo resultado. Veamos qué parámetros recibe esta propiedad y cómo funciona. `rotate`: `none` | `numX numY numZ angle` | `eje angle` | `angle`. Como vemos, podemos pasarle un parámetro a esta propiedad para indicarle que no se rotará (none) o un ángulo que lo que hará sería rotar sobre sí mismo (mismo efecto que si hiciéramos un `rotateZ`), cuando le pasamos dos parámetros es porque le indicamos el eje sobre el que queremos que rote, y si le indicamos cuatro parámetros es porque vamos a decirle mediante un 0 ó 1 si queremos que sobre ese eje rote y qué grados. La función `rotate3d(x, y, z, grados)` equivale a este último caso, es decir, `rotate`: `numX numY numZ angle`.Veamos unos ejemplos:

```
img {
    rotate: 45deg; /* Equivale a transform: rotateZ(45deg); */
    rotate: x 45deg; /* Equivale a transform: rotateX(45deg); */
    rotate: y 120deg; /* Equivale a transform: rotateY(120deg); */
    rotate: 0 0 1 45deg; /* Equivale a transform: rotateZ(45deg); */
    rotate: 1 0 0 15deg; /* Equivale a transform: rotateX(15deg); */
    rotate: 0 1 1 5deg; /* Equivale a transform: rotateY(5deg) rotateZ(5deg); */
    rotate: 1 1 1 5deg; /* Equivale a transform: rotateX(5deg) rotateY(5deg) rotateZ(5deg); */
}
```

Del mismo modo que en las translaciones, tenemos la posibilidad de rotar un elemento sobre el eje Z.

`rotate3d(angle, angle, angle, angle)`.

15.3. Escalado

Cuando queremos aumentar o disminuir el tamaño de una imagen tenemos la función `scale` que nos permite modificar el tamaño de un elemento. Las distintas funciones de escalado que tenemos son: `scaleX(num)`, `scaleY(num)`, `scaleZ(num)`, `scale(numX, numY)`, `scale(numX)`, `scale3d(numX, numY, numZ)` .Veamos un ejemplo, donde duplicamos anchura y reducimos a la mitad la altura:

```
img {
    transform: scale(2, 0.5);
}
```

Como pasa con `rotate`, tenemos la propiedad `scale` en los nuevos navegadores de forma que no es necesario usar la propiedad `transform`. La propiedad `scale` acepta los siguientes valores:`scale`: `none` | `numXY` | `numX numY`| `numX numY numZ`. Donde los será `none` si no queremos que se escale, un número que será una escala para el ejeX y para el ejeY, cuando damos dos valores será la escala que coja al ejeX, y ejeY, y cuando damos tres valores será el valor de escalar cada eje.

```
#box1:hover {
    scale: 1.25;
    /* Equivale a poner */
    scale: 1.25 1.25;
}
```

15.4. Deformaciones

Si queremos retorcer o inclinar un elemento 2d tenemos la posibilidad de usar la función skew. `skewX(anguloX)`, `skewY(anguloY)`, debemos tener cuidado porque la función `skew` como tal está marcada como obsoleta, por tanto, mejor usarla por separado cada eje.

15.5. Transformaciones a 3D

Como hemos visto, hay muchas funciones que nos dan la posibilidad de realizar transformaciones 3d (otras no, como es el caso de las deformaciones).

Antes de utilizar las transformaciones 3d, es importante conocer alguna de las propiedades que han sido derivadas de las transformaciones como es transform-style: flat | preserve-3d y transform-origin: posX posY posZ.

Mediante la función `transform-style` podemos modificar el tratamiento de los elementos hijos, donde por defecto está establecido a `flat` (es decir, los trata como elementos 2d), si queremos que los trate como elementos 3d debemos darle el valor preserve-3d. La función `transform-origin` cambia el punto de origen del elemento en una transformación.

16. Dibujar con CSS

Además