Guía Responsive / SASS

Diseño responsivo

Ya hemos visto que con FlexBox y CSS Grid ya tenemos algo de diseño responsivo, layouts que se adaptan, items que se recolocan dependiendo del espacio disponible..., pero en algunos casos puede que necesitemos hacer un cambio más disruptivo, esto se suele dar en versiones móviles vs escritorio de una web.

Definición

¿Qué es el diseño responsivo? se le denomina así a los diseños web que tienen la capacidad de adaptarse al tamaño y formato de la pantalla en la que se visualiza el contenido.

Esto nos puede servir para evitar tener que implementar una web para cada tipo de dispositivo.

Para hacer esto:

- **Debemos conocer las resoluciones más utilizadas:** por ejemplo un restuarante normalmente va a tener muchos usuario que accedan desde el movil (por ejemplo consultar la carta).
- **Debemos conocer el público objetivo de nuestra aplicación o sitio web**: por ejemplo una aplicación para personas mayores para que introduzcan sus parámetros de salud, en este caso sabemos que el público son personas que pueden tener visibilidad reducidad etc... y podríamos poner botones más grandes, menos información en pantalla.

Esto de la web para cada tipo de dispositivo... hace unos años era normal tener dos sitios webs distintos uno para escritorio y otro para Móvil, con el sobrecoste que esto implicaba.

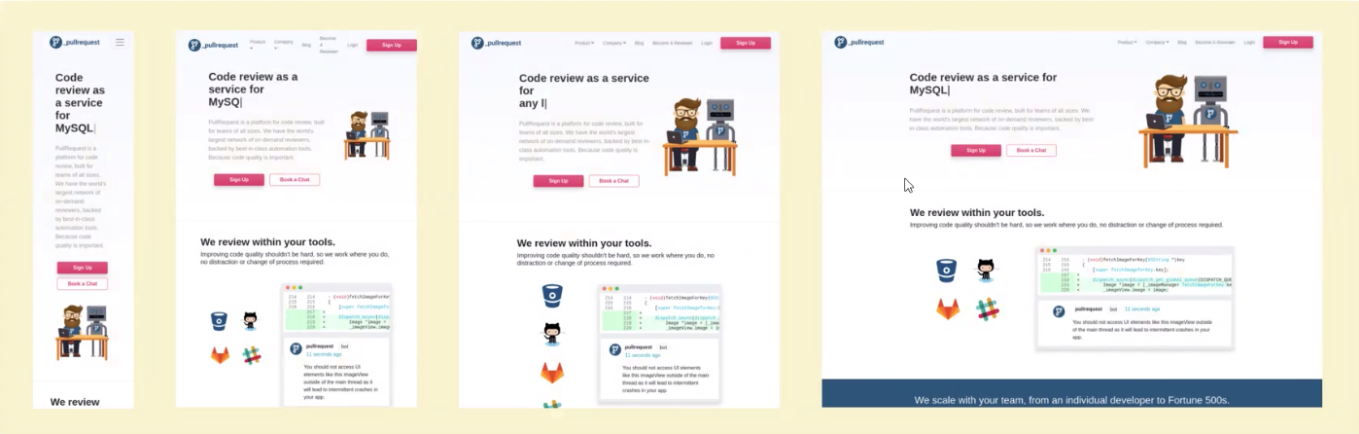
Técnicas básicas

Tres técnicas de diseño responsivo como principios fundamentales:

- **Fundamentos fluidos**: Empezar a pensar de manera proporcional, evitar las medidas absolutas (píxeles) y empezar a pensar de manera propocional (*rem* nos permite cambiar el tamaño del root y que se actualize todo sin tener que tocar en cada sitio, porcentajes)
- **Contenido flexible**: El contenido ha de ajustarse a la resoución (aplicar *fexbox*, *css-grid*).
- **Media Queries:** Cuando la maquetación y diseño dejan de funcionar en distintas resoluciones podemos corregirlo utilizando media queries, aquí podemos aplicar diseño específicos al contenido, en ciertas ocasiones un mismo diseño se puede adaptar bien de escritorio a tablet, pero cuando pasamos a móvil puede que tengamos que aplicar otro estilado de contenedores.

¿Cuando implementar?

Vemos un ejemplo de diseño responsivo:



No es perfecto, por ejemplo el menú superior podría haberse comprimido antes de que se empezara a montar, pero tiene una base interesante.

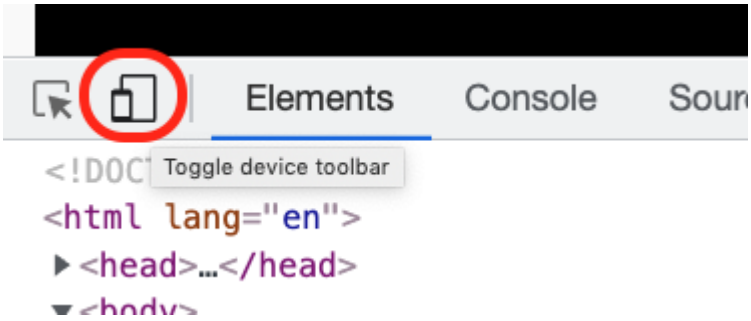
¿Como se adapta?:

- En escritorio mostramos todo el contenido, es más nos sobra espacio, el contenido se centrar y mostramos unos margenes en blanco a izquierda y derecha.
- En cuanto empezamos a hacer la ventana más pequeña, lo que vamos perdiendo son los márgenes, pero el contenido sigue siendo bastante parecido.
- Si seguimos disminuyendo espacio, el elemento secundarios como las imagenes se hacen más pequeñas y por ejemplo se agrupan en vertical u horizontal para ocupar menos espacio.
- Y si ya vamos a móvil, el menú de arriba desaparece y vemos el botón de hamburguesa para las opciones y todo el diseño está en una sóla columna en vertical (ya no está en el eje horizontal).

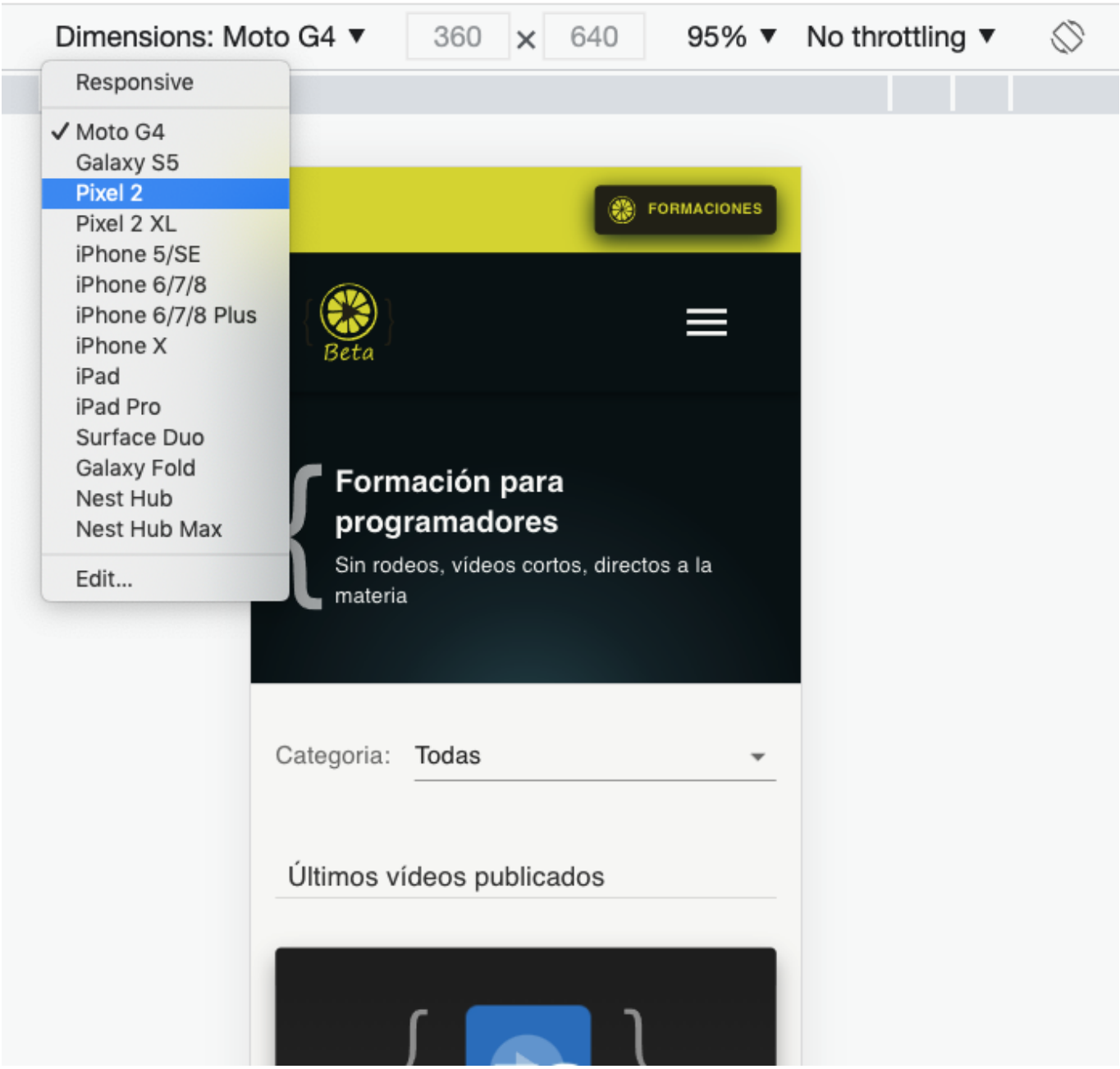
Es decir en este diseño partimos de dos secciones en el eje horizontal y estás a su ves se reparten en el eje vertical... en cuanto nos vamos a una resolución móvil pasamos a repartirlo todo en el eje horizontal (una sóla columna).

Herramientas y resoluciones

Un tema importante para hacer desarrollo responsivo es poder hacer un preview de nuestra ventana en diferentes resoluciones, para ello las *devtools* de *chrome* nos ofrece una utilidad para simular la resolución de diferentes dispositivos (icono *toggle-device-toolbar*).



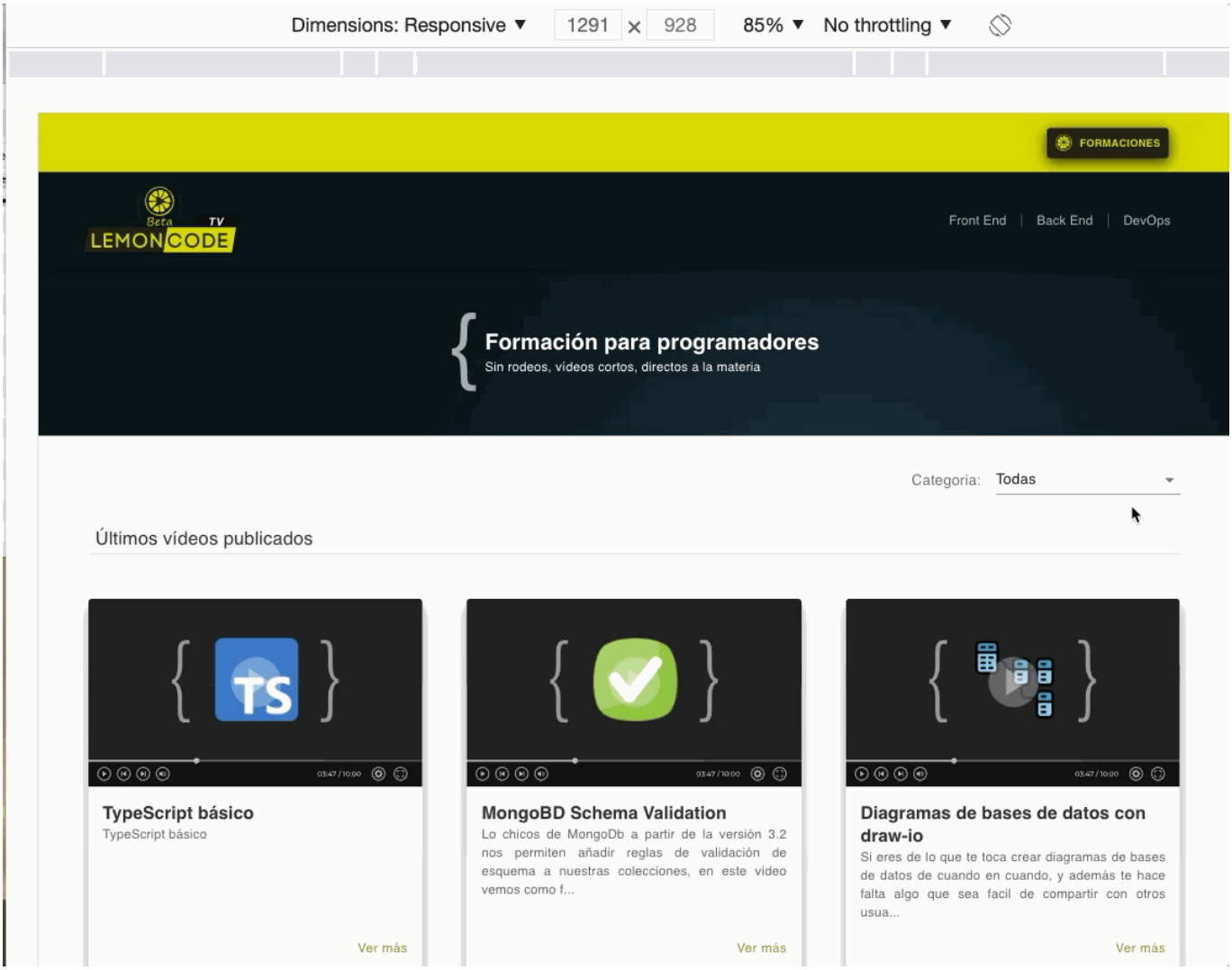
Si mostramos la barra, pinchando en el icono de *toggle-device-toolbar* se nos abre una barra de cabecera en la que podemos incluso elegir el tipo de dispositivo móvil para emular su resolución.



¡Ojo! esto no es un emulador del dispositivo, pero si podemos jugar con la resolución del mismo y ver como queda nuestra web.

En los settings podemos también añadir un dispositivo a la lista y especificarle la resolución, etc..

También podemos ir jugando a ir disminuyendo la resolución de forma progresiva y ver como se va adaptando la web:



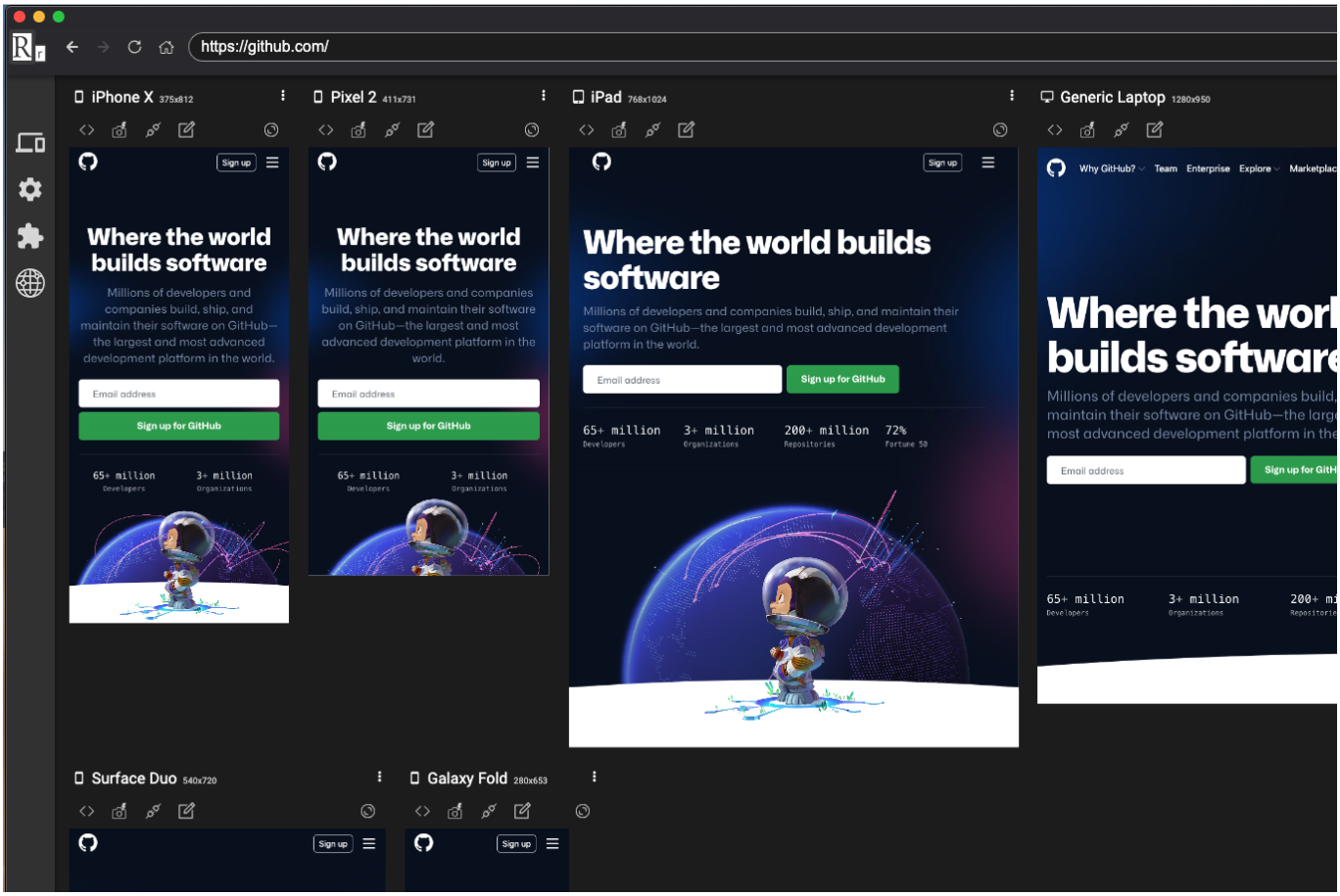
Si hacemos un inspect, nos podemos encontrar que hay entradas en la parte de estilos marcados con *@media* para aplicar estilos espefícos para ciertos tramos de resoluciones.

```
@media (min-width: 0px)
.css-1mtin4r {                                          <style>
   column-gap: 16px;
}
```

En estos *@media* le indicamos un punto de ruptura (breakpoints) en el que le indicamos a partir de que resolución o en que tramo aplicar un estilado u otro.

Tooling

Si bien con Google Chrome podemos simular diferentes dispositivos, existe una herramienta muy interesante que se llama Responsively que me permite visualizar una web simulando múltiples dispositivos a la vez, si las descargamos, podemos abrirla y poner la dirección que veamos oportuna y veremos nuestra página en varias simulaciones de dispositivos a la vez.



Ojo a todas estás soluciones, sirven a nivel de resolucíon, pero no a nivel de probar un navegador de un dispositivo físico, por ejemplo Chrome de Apple usaba el motor de safari, o el navegador de *Chrome* no es exactamente igual que el *Chrome Mobile*.

Si quieres probar tu web en *localhost* desde un dispositivo físico ppiensa mejor en usar una app de *tunneling* como *ngrok*, esto permite vincular una url de *ngrok* a nuestro *local* y desde esa *url* podemos acceder en nuestro dispositivo como si fuera un sitio web público.

Estrategias

Vamos a ver dos estrategías que podemos tomar en cuenta a la hora de arrancar con nuestro diseño:

- **Mobile first**: diseñamos nuestro sitio pensando en dispositivos móviles y luego en los otros (parto de una resolución pequeña y a partir de ahí vamos adaptando a resoluciones más grandes).
- **Desktop first**: diseñanmos nuestro sitio pensando en dispositivos de escritorio y después en los otros (parto de una resolución grande y a partir de ahí vamos adaptando a resoluciones pequeñas).

¿Qué aproximación elegir? Depende un poco del uso que va a tener tu aplicación:

- Si es crítico que funcione bien en móvil, o la mayoría de usuarios van a entrar por ahí es buena idea arrancar por mobile first, porque así estás enfocado en una aplicación ligera y un interfaz orientada a móvil.

- Si el uso principal va a ser escritorio, nos podemos plantear maquetar para escritorio y después ir adaptando, por ejemplo una aplicación en la que tenemos que rellenar la declaracíon de la renta,

tendría sentido plantearlo para desktop, adaptar a tablet, y plantear incluso si tendría sentido cubrir móvil.

Media Queries

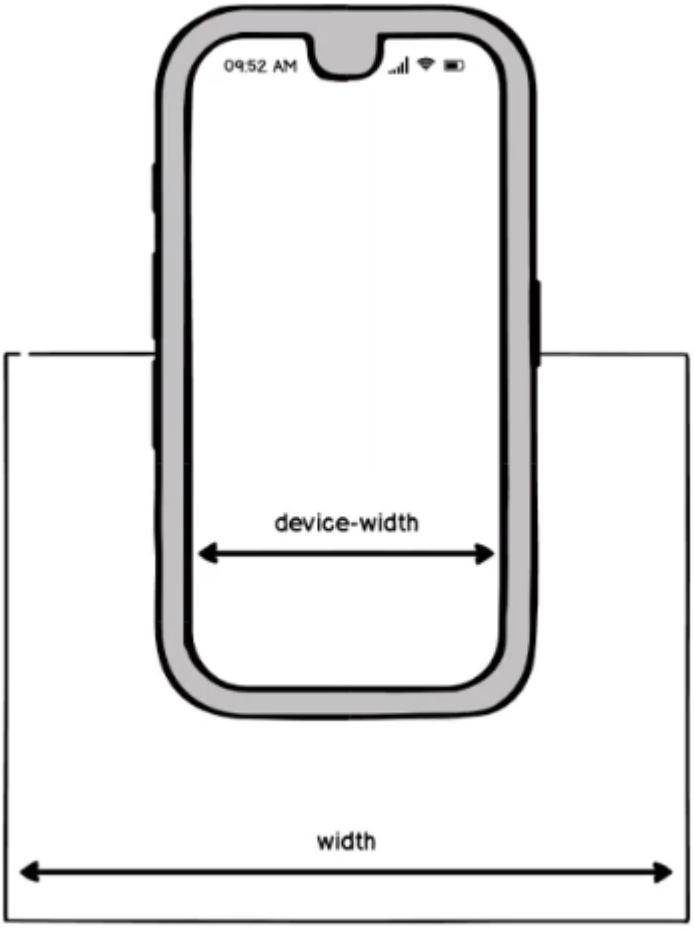
Las media queries nos permite aplicar unos estilos u otros dependiendo del tamañao de pantalla que estemos manejando, estos nos permite ajustar nuestro layout para que sea óptimo y usable tanto en escritorio, tablet, movil...

La sintaxis básica:

```
@media not|only mediatype and|not|only (condition) {
  /* reglas css */
}
```

Aquí le estamos diciendo que aplica unas condiciones a:

- *mediatype*: lo normal es que aquí elijamos *screen* (pantalla), aunque podemos elegir otras opciones como *print* para vista previa de impresión, o *speech* destinado a sintetizadores de voz, normalmente vamos a usar el *mediatype screen*, pero en algunas ocasiones puede que nos interese usar el *print* por ejemplo tenemos un banner a la derecha de nuestra ventanas que no queremos que aparezca cuando se imprima, en este caso dentro del *media print* le índicamos que ese *banner* va a tener el *display a _none*.
- *condition*: bajo que condicion queremos que se apliquen estos estilos (por ejemplo si el espacio horizontal que tengo disponible es menor de 320 pixeles).
- Estas condiciones las podemos anidar, y podemos aplicar los operadores lógicos *and only* y *not*



Una posible aproximacíon de tus media queries si sigues *mobile first* es la siguiente:

```
@media screen and (max-width: 320px) {
  /* reglas CSS para igual o menor que 320px de ancho */
}

@media screen and (max-width: 640px) {
  /* reglaCSS para igual o menor que 640px de ancho */
}
```

Aquí le estamos diciendo que si la pantalla disponible tiene como máximo 320 pixeles de ancho, que le aplique unos estilos, a partir de ese ancho que deje de aplicar los estilos, después hay otros estilos que se aplicarían hasta la resolución de 640 pixeles.

Fijate como aquí vamos quitando estilos como crecemos de resolución (vamos de menos a más)

Y una posible aproximación para orientarlo a *desktop first*

```
@media screen and (min-width: 1024px) {
  /* reglas CSS para igual o mayor que 1024px de ancho */
}
```

```
@media screen and (min-width: 640px) {
  /* reglaCSS para igual o mayor que 640px de ancho */
}
```

En este caso si tenemos un espacio mayor que 1024 pixeles, estamos aplicando los estilos de *640* y los de *1024* es decir vamos poniendo estilos encima.

¿Quiere esto decir que a partir de ahora tenemos que meter todos nuestros CSS dentro de media queries? Noooo... de hecho una gran parte del CSS va a estar fuera (el que no cambia estés en la resolución que estés), y dentro de las media queries tendrás los estilos específicos para cierto rango de resoluciones.

> Los terminos *mobile first desktop first* son un poco engañosos, hoy en día te puedes encontrar tables y móviles con resoluciones tipo desktop, y por otro lado podemos tener un navegador en escritorio pero queremos tener la ventana del navegador pequeña.

Vamos a por el *codesandbox* y a jugar con esto:

En el HTML vamos a sustituir el contenido del *body* por el siguiente:

./index.html

```
<body>
+  <header>Master Lemoncode</header>
+  <div class="grid-container">
+    <div class="item">1</div>
+    <div class="item item-2">2</div>
+    <div class="item">3</div>
+    <div class="item">4</div>
+    <div class="item">5</div>
+    <div class="item">6</div>
+  </div>
</body>
```

Aquí tenemos un contenedor (vamos a crear un CSS Grid), y seis items, uno de ellos lo marcamos con una clase para identificarlo (el numero 2).

Vamos a los estilos, borramos el contenido y vamos a introducir el siguiente:

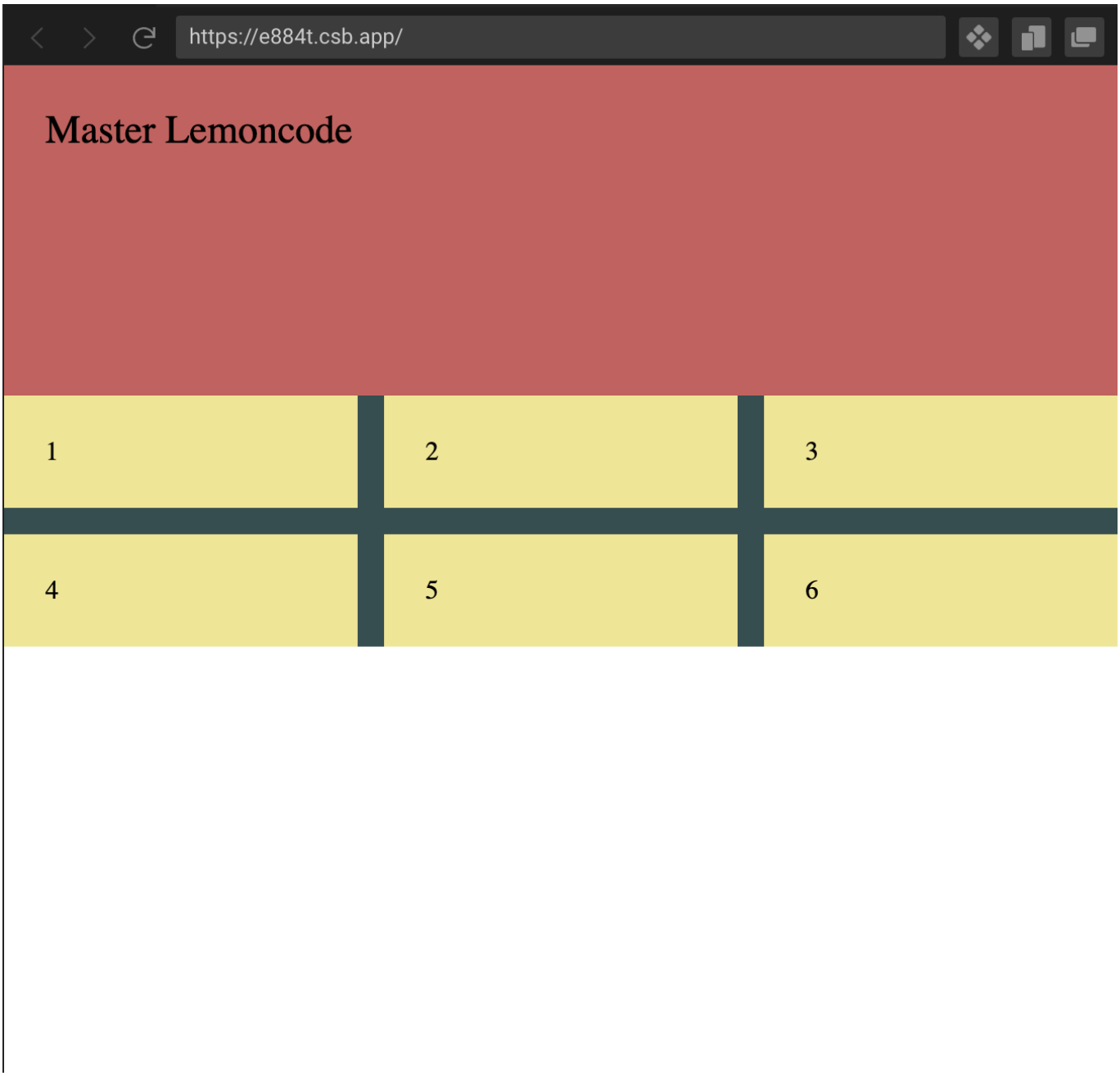
- Le eliminamos los bordes al margen.
- Creamos los estilos para un grid con 3 columnas por 2 filas.
- Le ponemos una cabecera con un tamaño considerable.

./src/styles.css

```
body {
  padding: 0;
  margin: 0;
}
```

```css
.grid-container {
  display: grid;
  background-color: darkslategrey;
  gap: 1rem;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
}

header {
  height: 150px;
  background-color: indianred;
  padding: 25px;
  font-size: 150%;
}

.item {
  background-color: khaki;
  padding: 25px;
}
```

El resultado:



> Si te fijas en el *body* siempre andamos quitando el *padding* y *margin* para evitar tener diferencias entre navegadores, hay librerías (por ejemplo ress) que ya hacen este *reset* (y otros) por ti.

Ahora mismo el resultado que tenemos es el siguiente:

Antes de seguir vamos a práctica con Flexbox ...¿y si quisieramos que la parte del *css grid* ocupará todo el espacio disponible (quitando el sitio de la cabecera) sin generar scroll vertical? Podríamos usar *calc* y restarle al *viewport height* los *150* pixeles de la cabecera, pero eso es un rollo, porque si el día de mañana meto otro elemento o cambio ese alto, esto dejaría de funcionar ¿Qué podemos hacer? ¡ *Flexbox* al rescate ! Defino el *Body* como un *flex-container* y al *grid-container* le doy un *flex-grow* de 1

./src/styles.css

```
body {
  padding: 0;
  margin: 0;
+ display: flex;
```

```
+ height: 100vh;
+ flex-direction: column;
}

.grid-container {
+  flex-grow: 1;
  display: grid;
  background-color: darkslategrey;
  gap: 1rem;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
}
```

Si ahora hacemos esto más pequeño, por ejemplo a partir de 650 pixeles, yo quiero que en vez de tener 3 columnas, quiero pasar a 2 ¿Cómo puedo hacer esto? En este caso hemos empezado por *desktop first* hemos diseñado el caso de escritorio, vamos a por nuestra primera media query para manejar este escenario:

```
.grid-container {
  flex-grow: 1;
  display: grid;
  background-color: darkslategrey;
  gap: 1rem;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
}

+ @media screen and (max-width: 650px) {
+  .grid-container {
+    grid-template-columns: 1fr 1fr;
+    grid-template-rows: 1fr 1fr 1fr;
+  }
+ }
```

Si ahora probamos podemos ver que al hacer más pequeña la ventana, hay un salto y se pasa al layout de 2 columnas por tres filas.

También podríamos aprovechar y el *height* del header dejarlo en el tamaño inicial:

```
 @media screen and (max-width: 650px) {
  .grid-container {
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 1fr 1fr;
  }

+ header {
+  height: initial;
+ }
 }
```

Un tema interesante es que sólo nos hace falta cambiar las propiedades que hace falta sobreescribir o añadir nuevas, las existentes que no cambian no hace falta tocarlas.

¿Y en tamaño móvil? Vamos a configurar el grid para que se vea en una sóla columna:

./src/styles.css

```
 @media screen and (max-width: 650px) {
  .grid-container {
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 1fr 1fr;
  }

 header {
  height: initial;
 }
}

+ @media screen and (max-width: 450px) {
+  .grid-container {
+    grid-template-columns: 1fr;
+    grid-auto-rows: 1fr;
+  }
+
+ header {
+  display: none;
+ }
+}
```

Ahora si ejecutamos podemos ver que conforme vamos haciendo más pequeña la ventana vamos pasando de tres columnas, a dos y finalmente a una y sin header.

Tal y como hemos planteado las media queries, hay un tema delicado, el orden, fijate si ahora cambiamos de sitio la última media query de *450px* y la ponemos por encima de la de *650px*.

```
body {
  padding: 0;
  margin: 0;
  height: 100vh;
  display: flex;
  flex-direction: column;
}

.grid-container {
  flex-grow: 1;
  display: grid;
  background-color: darkslategrey;
  gap: 1rem;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
}

header {
  height: 150px;
  background-color: indianred;
  padding: 25px;
  font-size: 150%;
```

```
}

.item {
  background-color: khaki;
  padding: 25px;
}

+ @media screen and (max-width: 450px) {
+  .grid-container {
+    grid-template-columns: 1fr;
+    grid-auto-rows: 1fr;
+  }
+
+  header {
+    display: none;
+  }
+ }


@media screen and (max-width: 650px) {
  .grid-container {
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 1fr 1fr;
  }

  header {
    height: initial;
  }
}

- @media screen and (max-width: 450px) {
-  .grid-container {
-    grid-template-columns: 1fr;
-    grid-auto-rows: 1fr;
-  }
-
-  header {
-    display: none;
-  }
-}
```

Si hacemos esto vemos que:

- Para desktop bien.
- Para tablet va bien.
- Para móvil... se me quedan las dos columnas 😦 en vez de una ¿Y esto por qué? El problema es que la segunda que llega gana (max-width:650px), nos podríamos dar cuenta de esto si abrimos las dev tools e inspeccionamos ese elemento.

Una opción para evitar esto es, añadir una condición previa (donde termina la anterior):

```
- @media screen and (max-width: 650px) {
+ @media screen and (min-width: 451px) and (max-width: 650px) {

  .grid-container {
    grid-template-columns: 1fr 1fr;
```

CSS Imports

Los CSS imports, me permiten importar dentro de un fichero CSS otros ficheros, así podemos estructurar nuetros ficheros CSS en distintas carpetas, etc...

Por ejemplo podemos organizar:

- Un fichero con lo estilos de base.
- Un fichero con los valores de variables y demás.
- Uno o más ficheros con temás más específicos de ventanas etc..

Vamos a ver esto con un ejemplo:

./src/base.css

```
body {
  padding: 0;
  margin: 0;
  height: 100vh;
  display: flex;
  flex-direction: column;
}
```

./src/styles.css

```
+ @import url("base.css")

- body {
-  padding: 0;
-  margin: 0;
-  height: 100vh;
-  display: flex;
-  flex-direction: column;
- }
```

SASS

Custom properties

Antes de arrancar con SASS, comentar que son las *custom properties de HTML*

Con CSS podemos utilizar variables, por ejemplo podemos definir una variable en el CSS de esta manera:

```
html {
  --back-color: darkorange;
}
```

Y como podemos usarla:

```
main {
  background-color: var(--back-color);
}

footer {
  background-color: var(--back-color);
}
```

De esta manera si un día deciden cambiar un color, no hay que reemplazarlo en mil sitios si no en uno sólo.

Un tema importante: la función *var()* sólo se puede utilizar para establecer valores de propiedades CSS, no se puede usar como nombres de propiedades, selectores, definición de media queries...

Vamos a ver esto en un *codesanbox* (vanilla JS)

En el HTML enlazamos el CSS:

index.html

```
  <head>
    <title>Parcel Sandbox</title>
    <meta charset="UTF-8" />
+    <link rel="stylesheet" type="text/css" href="./src/styles.css" />
  </head>
```

Vamos a meter contenido en el body del *html*

index.html

```
<body>
+ <header>Master Front End Lemoncode</header>
+ <main>Hello Lemoncoders!</main>
</body>
```

Y ahora en el *css* vamos a definir la variable que comentamos:

./src/style.css

```
html {
  --primary-color: darkorange;
}
```

> En los ejemplos estamos trabajando con un sólo fichero CSS, lo normal en un proyecto grande es romperlo en varios para hacerlo más mantenible, también puedes usar CSS in js.

Al meter la variable dentro del *html* le estamos definiendo un *scope* en este caso le decimos que todos los elementos que sean hijos de *html* podrán ver esta variable, si por ejemplo lo declarara dentro del *header* el elemento *main* no tendría acceso a la variable *--primary-color*

Y vamos a usarlo en el estilado que vamos a definir, tanto en el *header* como en el *main*

./src/styles.css

```
html {
  --primary-color: darkorange;
}

+ header {
+  background-color: var(--primary-color);
+  padding: 25px;
+ }

+ main {
+  color: var(--primary-color);
+  font-size: 125%;
+  margin: 25px;
+ }
```

Fijate que tanto *header* como *main* aplican el color definido en la variable.

Vamos ahora a definir una *card*, en los estilos vamos a indicarle que el color a utilizar para el borde es el de la variable.

./src/styles.css

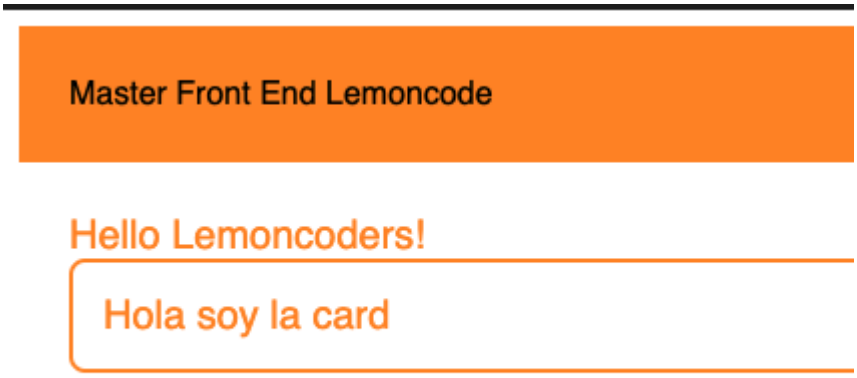
```
main {
  color: var(--primary-color);
  font-size: 125%;
  margin: 25px;
}

+ .card {
+  border: 2px solid var(--primary-color);
+  padding: 15px;
+  border-radius: 8px;
+ }
```

Y en el *html* vamos a mostrar la card:

index.html

```
  <main>
    Hello Lemoncoders!
++    <div class="card">Hola soy la card</div>
  </main>
```

El resultado:



¿Que pasaría si yo ahora dentro del scope del card cambiara el valor de la variable primary-color a verde? ¿Se aplicará el nuevo valor de forma global ó sólo en el scope de las cards?

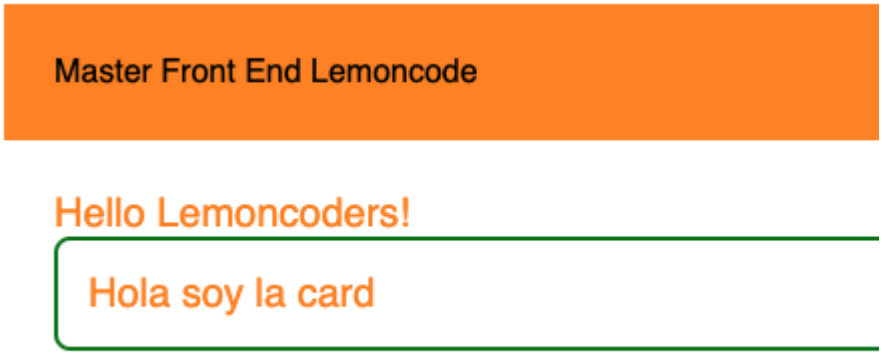
Y el texto que viene heredado del *html ¿Se quedaría verde o naranja?*

```
 .card {
+ --primary-color: green;
  border: 2px solid var(--primary-color)
  padding: 15px;
```

```
  border-radius: 8px;
 }
```

El resultado que tenemos:

- En el resto de elementos el color que se aplica es el naranje (no es el scope del *card*).
- Dentro del *card* el color del borde se cambia a verde (el de su scope).
- El color del texto al ser heredado es naranja 😡.



¿ Y por qué el texto del card sale en naranja? Porque se hereda, si explicitamente le dieramos ese color a la propiedad si aparecería en verde:

```
 .card {
  --primary-color: green;
  border: 2px solid var(--primary-color)
  padding: 15px;
  border-radius: 8px;
+ color: var(--primary-color)
 }
```

Otro tema interesnate, sería subir esa asignación de *color* al main, en este caso el valor bajaría al card y se aplicaría tanto al *border* como a la *fuente*

```
main {
+  --primary-color: green;

  color: var(--primary-color);
  font-size: 125%;
  margin: 25px;
}

.card {
-  --primary-color: green;
  border: 2px solid var(--primary-color);
  padding: 15px;
```

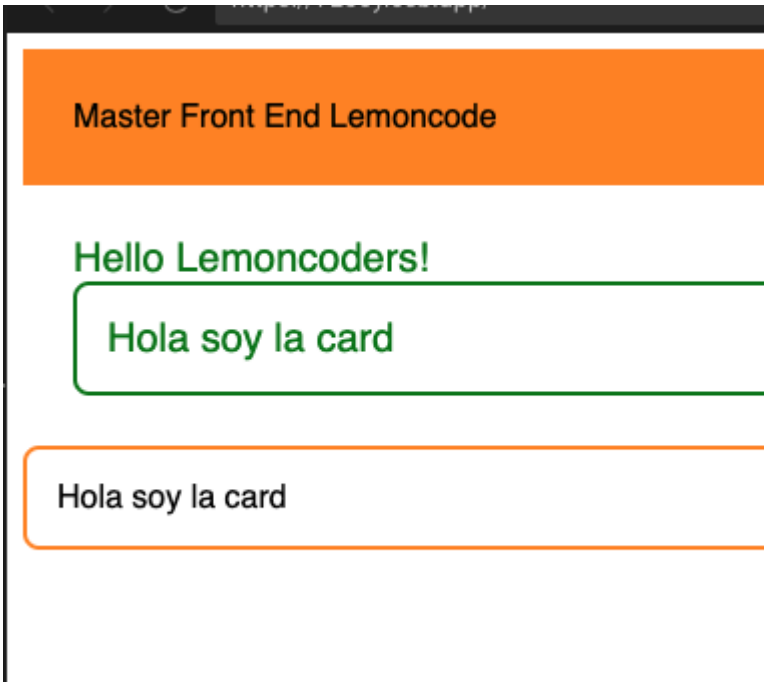
```
  border-radius: 8px;
}
```

Y... ¿ Que pasaría ahora si pongo un *card* fuera del _main?

index.html

```
  <body>
    <header>Master Front End Lemoncode</header>
    <main>
      Hello Lemoncoders!
      <div class="card">Hola soy la card</div>
    </main>
+    <div class="card">Hola soy la card</div>
  </body>
```

Pues que esa tarjeta en concreto estaría fuera del scope de main, y el borde sería naranja, y la fuente negro.



Como ves esto de las variables y la herencia es un poco lio, podría pasar que usemos una variable y no esté definida en ese scope, para curarnos en salud podríamos tener un valor por defecto, esto lo hacemos añadiendo un segundo parametro a *var(--primary-color)* en el que le indicamos el valor de fallback en caso de de que no esté definida.

Veamos este escenario:

```
+ body {
  font-family: sans-serif;
}
```

```
html {
}

header {
  --primary-color: darkorange;
  background-color: var(--primary-color);
  padding: 25px;
}

main {
  --primary-color: green;
  color: var(--primary-color);
  font-size: 125%;
  margin: 25px;
}

.card {
  border: 2px solid var(--primary-color);
  padding: 15px;
  border-radius: 8px;
}
```

En este caso si ponemos un *card* fuera del *main, no tendrá la variable _primary-color* definida, ¿Que podemos hacer para definirle un valor de fallback ?:

```
.card {
-  border: 2px solid var(--primary-color);
+  border: 2px solid var(--primary-color, red);
  padding: 15px;
  border-radius: 8px;
}
```

> Las variables de CSS no se solían usar, ya que lo normal era trabajar con preprocesadores como SASS que además te dan más libertad para usar variables, en framework de última hornada, se han vuelto a poner de moda estas variables.

SASS Conceptos

SASS viene de **S**intactically **A**wesome **S**tyle **S**heets, es un preprocesador de CSS ¿Qué quiere decir esto? Es un super lenguaje que ponemos encima de CSS que nos permite manejar variables, trozos reusables etc... es decir le añadimos "super poderes al css".

¿Y eso como se lo come el navegador? El navegador no sabe ni que existes:

- Tu defines tus hojas de estilo usando SASS.
- Cuando vas a hacer un build, esas hojas en formato SASS se convierten a CSS estándar y es lo que el navegador ve.

¿Y porque no lo implementan directamente en CSS o soltamos el SASS en el navegador?

- CSS es un estándar que va a su ritmo, incluir nuevas características lleva su tiempo y además cada navegador también emplea su tiempo en incorporarlas (algunos como IE11 jamás incorporarán lo nuevo).
- Hay transformaciones de configuración que consumen CPU y sólo se hacen una vez ¿Porque no incluirlas en el ciclo de Build? Es algo parecido a TypeScript.
- SASS le añade a CSS: variables, anidamiento de estilos, mixins, funciones, condicionales, bucles...

> Para trabajar con *SASS* podemos usar una extensión de *css* (*scss*), también ofrece una opcion con sintaxis tipo *yml* pero ésta es menos común encontrartela.

Para trabajar con SASS, vamos a aprovechar el soporte de *codesandbox* a *scss*, si lo queréis probar en local tendrías que instalarlos un *bundler* (*parcel*, o *webpack*) y realizar la configuración necesaria para poder añadir el preprocesador en el flujo de build

Empecemos con un ejemplo simple, en el *html* vamos a sustituir el contenido del *body* por el siguiente:

```
<body>
+ <div class="container">
+   <p>Máster Front End Lemoncode</p>
+   <main>
+     <p>Hello Lemoncoders</p>
+   </main>
+ </div>
</body>
```

En el lado del estilado:

- Añadimos un contenedor.
- Y le indicamos que todos los parrafos que estén dentro de ese div deben tener un estilo en concreto (una fuente más grande).

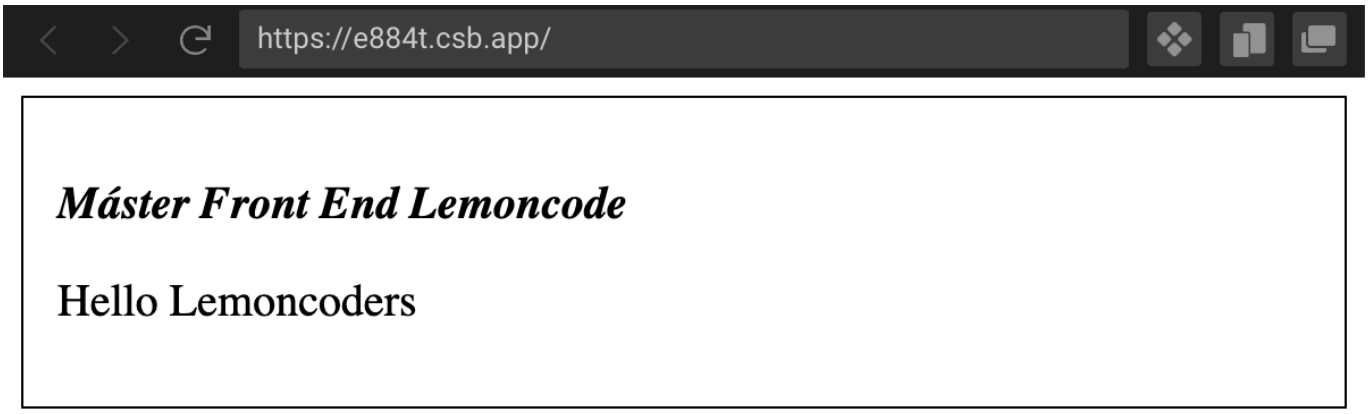
```
div.container {
  padding: 15px;
  border: 1px solid black;
}

div.container p {
  font-size: 125%;
}
```

- Vamos a indicarle que sólo a los hijos directos de ese *container* lo quiero poner como *italic* y *bold*:

```
div.container {
  padding: 15px;
  border: 1px solid black;
}
```

```
div.container p {
  font-size: 125%;
}

+ div.container > p {
+  font-style: italic;
+  font-weight: bold;
+ }
```

El resultado:



Imaginate que quisieramos decir, me gustaría poder anidar estas reglas para poder tener un código más mantenible.

Renombramos el fichero de *styles.css* a *styles.scss* y lo actualizamos en el enlace de nuestro *index.html*:

index.html

```
  <head>
    <title>Parcel Sandbox</title>
    <meta charset="UTF-8" />
-    <link href="./src/styles.css" />
+    <link href="./src/styles.scss" />
  </head>
```

Vamos a anidar los selectores utilizando la aproximación de SASS, así podemos ver como obtenemos un código más mantenible:

- Dentro de div container anidamos un párrafo y le indicamos el *font-size* a *125%*.
- También anidamos la regla que aplica a los hijos directos del div en la que le aplicamos el éstilo *italic* y *bold*.

./src/styles.scss

```
div.container {
  padding: 15px;
  border: 1px solid black;
+
+ p {
+   font-size: 125%;
+ }
+
+ & > p {
+  font-style: italic;
+  font-weight: bold;
+ }
}

- div.container p {
-  font-size: 125%;
- }
-
- div.container > p {
-  font-style: italic;
-  font-weight: bold;
- }
```

Todo esto que estamos aplicando CSS no lo permite, de ahí que nos pongamos con *scss* (SASS) si vemos el ejemplo en nuestro *codesandbox* podemos comrpobar que el resultado es el mismo, eso si en background se realiza la conversión de *scss* a *css* y lo que se sirve al navegador es un *css* estándar.

Para trabajar con esto a nivel de desarrollo en local (sin codesandbox):

- Tendríamos un proyecto web.
- Usaríamos un bundler (parcel o webpack).
- Insalaríamos un preprocesador de SASS como por ejemplo *sass* (*npm install sass --save-dev*).
- Lo configuramos en el proceso de build.

Esto veremos como hacerlo en el *módulo 3 - Bundling*.

Reglas anidadas

Como hemos visto, podemos escribir un selector y dentro de ese selector poner otro.

por ejemplo:

```
#main p {
  color: #00ff00;
  width: 97%;
}

#main p .redbox {
  background-color: ##ff0000;
```

```
  color: #000000;
}
```

Se puede escribir como

```
#main p {
  color: #00ff00;
  width: 97%;

  p .redbox {
    background-color: ##ff0000;
    color: #000000;
  }
}
```

Y si anidamos... ¿Tenemos alguna manera de que un hijo pueda refernciar al selector padre? Si, usando el caracter especial *&*.

Es decir el siguiente código *css* estandar:

```
a {
  font-weight: bold;
  text-decoration: none;
}

a:hover {
  text-decoration: underline;
}

body.firefox a {
  font-weight: normal;
}
```

Podríamos escribirlo como:

```
a {
  font-weight: bold;
  text-decoration: none;
  &:hover {
    text-decoration: underline;
  }

  body.firefox & {
    font-weight: normal;
  }
}
```

En este caso el *&* referencia al selector padre que en este caso es el *a* (el anchor padre que hemos puesto en el fichero *scss*).

De esta manera generamos un código más mantenible.

Vamos a ver esto exteniendo el ejemplo del apartado anterior, si recordamos teníamos el siguiente *html* y *sccs* (ojo con la extensión que ya no es *css*):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Parcel Sandbox</title>
    <meta charset="UTF-8" />
    <link href="./src/styles.scss" />
  </head>

  <body>
    <div class="container">
      <p>Máster Front End Lemoncode</p>
      <main>
        <p>Hello Lemoncoders</p>
      </main>
    </div>
  </body>
</html>
```

```
div.container {
  padding: 15px;
  border: 1px solid black;

  p {
    font-size: 125%;
  }

  & > p {
    font-style: italic;
    font-weight: bold;
  }
}
```

Vamos a indicarle que queremos que cuando se pasa el ratón por encima del contenedor queremos cambiar el color de fondo:

```
div.container {
  padding: 15px;
  border: 1px solid black;

  p {
```

```
    font-size: 125%;
  }

  & > p {
    font-style: italic;
    font-weight: bold;
  }
+
+ &:hover {
+  background-color: khaki
+ }
}
```

> Fíjate que esta regla la hemos anidado dentro del *div.container*

Propiedades anidadas

Para manejar propiedades parecidas en un CSS, éste nos ofrece algo de azúcar, como por ejemplo border y width que podemos decir uno, dos, tres o cuatro valores, también nos ofrece short-hands... pero hay casos en los que puedes ser pesado meter valores, por ejemplo definiendo una fuente:

```
p {
  font-style: italic;
  font-weight: bold;
  font-size: 200%;
}
```

Esto en *sass* lo podemos escribir de la siguiente manera:

```
p {
  font: {
    style: italic;
    weight: bold;
    size: 200%;
  }
}
```

> ¡OJO! fijate que en el *font* le hemos puesto los dos puntos *:* es nuestra manera de indicarle al preprocesador de que eso es *SASS*

Ejercicio: ¿ Te animas a probar esto con las propiedades de los borders? ¿Que diferencias ves con los *shorthands*?

Variables

Las variables de SASS nos permiten:

- Reutilizar valores sin tener que copiarlos.
- Actualizar cualquier valor de manera sencilla.
- Tienen contexto dentro de las reglas anidadas (como la de HTML)

¿Cómo defino las variables con *SASS*? Usando el signo *$*, por ejemplo:

```
$mycolor: #ffeedd;
```

Veamos esto con un ejemplo.

Sustituimos el contenido del *body* de nuestro *html* con el siguiente contenido:

- Un contenedor A que dentro tiene un parrafo.
- Un contenedor B (hermano del A) que dentro tiene otro párrafo.
- Un enlace exterior a los dos contenedores.

./src/index.html

```
<body>
+  <div class="container-a">
+   <p>Párrafo del contenedor a</p>
+  </div>
+  <div class="container-b">
+    <p>Párrafo del contenedor </p>
+  </div>
+  <a>Esto es un enlace</a>
</body>
```

Vamos a por la hoja de estilos, lo primero vamos a declarar variables (reemplazamos todo el contenido anterior que hubiera).

./src/styles.scss

```
$primary-color: darkorange;
$secondary-color: skyblue;
$text-color: darkblue;
$font-base-size: 1.3em;
$link-font-size: 1.5em;
```

Vamos a darle uso a esas variable en mi estilado, aprovechamos para jugar con el operador *&*:

- Decimos que vamos a estilar párrafos.
- En la regla anidada indicamos el contenedor *container-a* y con el *&* le decimos que aplique el párrafo padre.

```

```

```
$font-base-size: 1.3em;
$link-font-size: 1.5em;

+ p {
+  .container-a & {
+    font-size: $font-base-size;
+    background-color: $primary-color;
+    color: $text-color;
+  }
+ }
```

Con esto podemos ver que al párrafo que hay dentro del *container-a* se estila aplicando los valores de las variables que hemos definido.

Podemos hacer algo parecido con el párrafo que está dentro del *container-b*

```
$font-base-size: 1.3em;
$link-font-size: 1.5em;

p {
  .container-a & {
    font-size: $font-base-size;
    background-color: $primary-color;
    color: $text-color;
  }

+  .container-b & {
+    background-color: $secondary-color;
+    color: $text-color;
+  }
}
```

Y vemos que se le aplica el estilado al párrafo que está bajo el *.container-b*.

Para el enlace *a* podemos usar la variable de *link-font-size* y el *weight* podemos decirle que el *weight* es *bold*, y además vamos a jugar con el *hover* de ese *a*

```
  .container-b & {
    background-color: $secondary-color;
    color: $text-color;
  }
}

+ a {
+  font: {
+    size: $link-font-size;
+    weight: bold;
+  }
```

```
+
+  &:hover {
+    color: $primary-color;
+  }
+ }
```

Así ya tenemos nuestro enlace estilado utilizando los valores de las variable que hemos definido anteriormente, la ventaja de esto es que si ,por ejemplo, el día de mañana cambia la paleta de colores de la imagen corporativa de mi empresas, yo sólo tengo que tocar en un sólo sitio para actualizarla en mi proyecto.

La forma que tiene SASS de trabajar con las variables es distinto a como lo hace *html*, cuando las variables de *SASS* pasan por el procesador, todo se queda en constantes repetidas en mil sitios, por ejemplo si tengo una variable con un color que utilizo en 30 sitios, se me sustituye en esos 30 sitios por el literal con el color, es decir cuando tienes una variable *SASS* y pasas a *css* estás desaparecen, ya no hay variables de *sass*.

Una cosa interesante es que con *SASS* si que puedes usar variables en *media queries*, por ejemplo:

```
$breakpoint-lg: 650px;

@media screen and (min-width: $breakpoint-lg) {
  /* Tus estilos aquí */
}
```

Operaciones

Podemos sumar, mutliplicar, dividir, y combinar esto para cálculos más elaborados, por ejemplo:

```
font-size: 4px + 4; // 8px
font-size: 20px * 0.8; // 16px
color: #fff / 4; // #404040
width: (100% / 2) + 25%; // 75%
```

También podemos combinarlo con valores de variables *SASS* que hayamos creado.

> Sobre la división, te recomiendan que en vez de usar el operador / utilizes *math.div*

Funciones de color

En base a una variable que tiene un color, podemos obtener colores más oscuros, claros, saturarlos, invertirlos, sacar su complementario...

```
$primary-color: darkorange;

.baz {
```

```
  color: lighten($color, 10%);
  color: darken($color, 10%);

  color: saturate($color, 10%);
  color: desaturate($color, 10%);

  color: fadein($color, 0.1);
  color: fadeout($color, 0.1);

  color: invert($color);
  color: complement($color);
}
```

Esto es muy útil si por ejemplo vamos a crear una paleta de colores para nuestra aplicación.

Otras funciones:

Otras funciones que nos podemos encontrar en SASS y que pueden ser de ayuda:

```
.baz {
  // Permite ponerle o quitarle comillas a un valor de una variable
  content: quote($string);
  content: unquote($string);

  // Operador ternario if
  content: $if($condition, $if-true, $if-false);

  // Redondear un valor
  content: floor($number);

  // Porcentajes
  content: percentage($number);

  // Concatenar dos arrays
  content: join($list1, $list2);
}
```

Interpolación de strings

Otra funcionalidad que nos permite SASS es interpolar strings, esto depende de como lo usemos puede hacer que nuestro código sea más críptico y no nos aporte valor.

Podemos concatenarle a un selector el valor de una variable (pasandolo a string), algo así como:

Este sería el *scss*

```
$name: foo;
$attr: border;
```

```
p.#{$name} {
  #{$attr}-color: blue;
}
```

Y el *css* resultando:

```
p.foo {
  boder-color: blue;
}
```

Una aplicacion en la que puede ser util interpolar strings: por ejemplo cuando tenemos un contenedor y un montón de items sueltos, y en cada item queremos dar un estilado siguiendo unas reglas, podríamos meter un bucle (ya veremos como funciona esto) y en los items hijos ponerle como sufijo a cada item el sufijo con el *id* del bucle (y por ejemplo en cada item jugamos con que tenga cada uno un color un poco más claro).

Estos son casos muy concretos, no debemos de abusar de esta funcionalidad.

@import

La regla *@import*:

- Permite importar también archicos *scss* y *sass*.
- Cualquier *variable* o *minx* definido se mezcla en la hoja.
- Permite importar en reglas anidadas.

Por ejemplo en *css* los imports tenemos que ponerlos arriba del todo y con *sass* podemos hacer un *import* por ejemplo dentro de un elemento:

```
#main {
  @import "colors";
}
```

Hojas de estilos parciales

Cuando veamos una hoja de estilo que comienza con el caracter _ significa que su contenido no se compilará a *css ¿Y esto para que sirve?* por ejemplo si nos creamos un fichero que se llame *variables.scss*, aunque esto ya muchos bundlers detectan que ficheros *scss* no generan *css* y no los generan en la salida al procesar.

@media

Nos permite anidar *media queries* en los selectores, es decir nos permite escribir algo así como:

```
.sidebar {
  width: 300px;
  @media screen {
    width: 500px;
  }
}
```

Y esto se traduce en *css* en:

```
.sidebar {
  width: 300px;
}

@media screen {
  .sidebar {
    width: 500px;
  }
}
```

Bien usado puede ser muy interesante para tener un código mantenible.

@extend

Nos permite arreglar reglas a todos lo selectores anidados.

Puedo definir una regla base e indicarle que otros selectores extiendan de la misma (copien los valores), veamos esto con un botón:

El código *sass*

```
.button {
  color: black;
}

.submit-button: {
  @extend .button;
  border: 1px black solid;
}
```

El *css* que se generaría:

```
.submit-button {
  border: 1px solid black;
}

.button,
```

```
.submitbutton {
  color: black;
}
```

Además de esto podemos decirle que extienda de cualquier regla, y además aplicar multiple *extend*, por ejemplo:

```
.submit-button {
  @extend: a:hover;
  @extend .button;
  border 1px black solid;
}
```

Esto hay que usarlo con cuidado, ya que inspeccionando con las dev tools podemos acabar con problemas para encontrar estilos etc...

mixins

Los *mixins* nos permite agrupar código que se repite en ciertos selectores y poder soltarlos donde queramos sólo con referenciarlo, es decir:

- Tenemos un nombre de regla y argumentos opcionales.
- Pueden contener selectores, incluso referencias al selector padre.
- Nos aporta funciones.
- Se incluyen con las directiva @include

Por ejemplo podemos tener:

```
@mixin clearfix {
  display: inline-block;
  &:after {
    content: ".";
    display: block;
    height: 0;
    clear: both;
    visiblity: hidden;
  }
  * html& {
    height: 1px;
  }
}
```

Vamos a ver esto con un ejemplo simple y útil: quiero contar con un mixin que me redondee los borders: esto me puede hacer tener que copiar varios valores repetidos en *safari*, *chrome* y *firefox*.

```
```

```
@mixin rounded-corners-all($size) {
  border-radius: $size;
  -webkit-border-radius: $size; // Safari, Chrome
  -moz-border-radius: $size; // Firefox
}

#form {
  @include rounded-corners-all(5px);
}
```

Ahora cuando queremos añadir a cualquier elemento ese borde redondeado, no tenemos que ir recordando como era para cada navegador, si no que *sass* va a ir sustiyuendolo donde toque.

function

Los mixin ya hemos visto que sustituye, ... *function* es parecido pero nos permite devolver un datos (tenemos un return), por ejemplo:

```
$app-width: 900px;

@funtion column-width($scols) {
  @return ($app-width / $scols) - ($cols * 5px);
}

.col2 {
  width: column-width(2);
}

.col3 {
  width: column-width(3);
}
```

Aquí estamos calculando el ancho de unas columnas en base al número de columnas que tenemos (este es un ejemplo tonto, lo normal sería usar flexbox o css grid para esto).

Directivas de control - if

Nos permite meter estructuras *if then else* en nuestro código *css*,

Un ejemplo de lo que se puede hacer:

```
p {
  @if $type == ocean {
    color: blue;
  } @else if $type == matador {
    color: red;
  } @else if $type == monster {
    color: green;
```

```
  } @else {
    color: black;
  }
}
```

Esto nos podría valer para definir temas *dark* o *light*.

Directivas de control - bucles

Aquí tenemos bucles *for*: muestra repetidamente un conjunto de estilos. En cada repetición se utiliza el valor de una variable de tipo contador para ajustar el resultado mostrado.

```
@for $i from 1 through 3 {
   .item-#{$i} {
      width: 2em * $i;
   }
}

@for $i from 1 to 3 {
   .item-#{$i} {
      width: 2em * $i;
   }
}
```

Bucles _For each_: se recorre toda la lista o mapa y en cada iteración, se asigna un valor diferente a la variable $var antes de compilar los estilos.

```scss
@each $item in first, second, third, fourth {
  .#{$item} {
    background-image: url('/images/#{$item}.jpg');
  }
}
```

Bucles while: La directiva @while toma una expresión SassScript y repite indefinidamente los estilos hasta que la expresión da como resultado false. Aunque esta directiva se usa muy poco, se puede utilizar para crear bucles más avanzados que los que se crean con la directiva @for.

```
$i: 6;
@while $i > 0 {
  .item-#{$i} {
    width: 2em * $i;
  }
  $i: $i - 2;
}
```

Material de referencia

Responsively: https://responsively.app/

Responsive Web Design Patterns: https://developers.google.com/web/fundamentals/design-and-ux/responsive/patterns

Responsive patterns: https://bradfrost.github.io/this-is-responsive/patterns.html

Página oficial SASS: https://sass-lang.com/