Manual de Taildwind CSS

Este es el Manual de TailwindCSS un popular framework de CSS de nueva generación que aporta sencillez en el desarrollo y agilidad para la apliación de estilos.

Tailwind viene a hacer la competencia a otras alternativas como Bootstrap, con un enfoque que aporta numerosas ventajas, como la sencillez de aplicación de los estilos, la facilidad para subreescribirlos, la optimización del código resultante, etc.

El enfoque de TailwindCSS se conoce de manera genérica como "atomic CSS" y consiste en especificar los estilos con clases muy concisas, que afectan generalmente a un único atributo, aplicando cierto valor. Para crear componentes por tanto se tienen que aplicar numerosas clases, lo que puede parecer un inconveniente, pero con un estudio minucioso se transforma en una verdadera optimización del CSS de los proyectos.

En este manual iremos recorriendo las características de TailwindCSS y el modo de usarlo, para poder comenzar a sacar partido a esta estupenda herramienta de desarrollo web.

Este manual está en proceso de creación!! Vuelve pronto para encontrar más artículos publicados.

cuando el usuario interacciona con la página.

Introducción a Tailwind CSS

En estos primeros artículos queremos ofrecer una introducción al framework CSS Tailwind. Nuestro objetivo consiste en ofrecer una vista de pájaro sobre cómo se trabaja con Tailwind y cómo organizarás tu código CSS para sacar partido a las clases de utilidad pero no desesperarte por ensuciar el HTML. Después de estos artículos tendrás suficiente conocimiento para implementar Tailwind CSS en un proyecto, pero no para sacarle todo el partido que en realidad ofrece.

Primeros pasos con TailwindCSS

Primeros pasos con TailwindCSS, instalación de PostCSS, instalación del plugin de TailwindCSS, configuración del sistema de compilado del código CSS y uso de Tailwind en un archivo HTML.

Tailwind es un framework CSS que permite el diseño de sitios web de una manera ágil y muy amistosa para el desarrollador. Ofrece una interfaz para el desarrollo de las hojas de estilo en cascada muy cercana al propio lenguaje CSS, que se aplica mediante clases de utilidad.

Es una manera rápida de desarrollar, en la que se requiere escribir poco o nada de CSS, ya que el propio framework ofrece prácticamente todos los estilos mediante clases, que nosotros podemos aplicar directamente en el código HTML.

Tailwind tiene muchas ventajas con respecto a frameworks de diseño más tradicionales como Bootstrap. Más adelante explicaremos con detalle cuáles son estas ventajas y por qué Tailwind es una alternativa muy a tener en cuenta. Sin embargo, en este artículo vamos a comenzar directamente por la práctica, explicando cómo empezar a usar esta herramienta.

Modos de trabajo posibles para Tailwind

Tailwind es un framework que requiere cierto setup para comenzar a trabajar. Aunque existe la posibilidad de instalarlo en una página mediante un CDN, no ofrece ni de lejos la mejor alternativa de integración en un proyecto frontend.

De hecho, Tailwind está pensado para que, cuando lo uses, siempre tengas en mente la optimización del CSS resultante. Por ello, para usar Tailwind lo tendrás que integrar con alguna de las herramientas que uses en tu proyecto para compilar el Javascript o el CSS. Algunas alternativas de integración serían por ejemplo Webpack, PostCSS o Laravel Mix, por poner varios ejemplos.

En este artículo vamos a explicar la manera más directa de usar Tailwind, que es mediante un plugin de PostCSS.

Cómo instalar Tailwind con PostCSS

Trabajar con Tailwind sobre PostCSS se hace muy sencillo. Además, PostCSS no es nada complicado de usar y, de paso que aprendes a integrar Tailwind aprenderás también a sacarle partido a otra herramienta excelente para tus CSS, si es que no lo usas todavía.

Comenzamos configurando PostCSS en el proyecto. Esto requiere una serie de pasos, que están explicados en el artículo de las PostCSS.

Instalamos TailwindCSS

PostCSS se organiza mediante una serie de plugins que instalaremos bajo demanda, dependiendo de las transformaciones que se deseen realizar sobre el CSS del proyecto. Así pues, una vez instalado PostCSS, tendremos que instalar el plugin para realizar el trabajo con Tailwind.

Este es el comando para instalar el plugin de TailwindCSS para PostCSS.

```
npm i -D tailwindcss
```

Ahora creamos el archivo de configuración de TailwindCSS, que se llama "tailwind.config.js". Este archivo lo puedes crear a mano, pero es más cómodo hacerlo a través de la línea de comandos. Para ello, estando situados con el terminal en la raíz del proyecto, lanzamos el siguiente comando:

```
npx tailwind init
```

El archivo de configuración de Tailwind tendrá un código inicial como este:

```
module.exports = {
  future: {
    // removeDeprecatedGapUtilities: true,
    // purgeLayersByDefault: true,
  },
  purge: [],
  theme: {
```

```
    extend: {},
  },
  variants: {},
  plugins: [],
}
```

De momento lo vamos a dejar tal cual nos lo ofrecen, puesto que con la configuración predeterminada tenemos suficiente por ahora. Más adelante aprenderemos a configurar el framework, para lo que colocaremos aquí nuestras propias personalizaciones.

Ahora configuramos el plugin de Tailwind para PostCSS, en el archivo "postcss.config.js". Ese fichero deberías haberlo creado anteriormente, cuando hiciste la configuración inicial de PostCSS.

```
module.exports = {
  plugins: [
    require('tailwindcss'),
    // otros plugins de postcss que desees usar, como autoprefixer
  ]
}
```

Como puedes comprobar, en el array de plugins de PostCSS estamos haciendo el "require" del plugin "tailwindcss". Por supuesto, podemos usar otros plugins de PostCSS, como por ejemplo "autoprefixer", en cuyo caso tendríamos un código como este:

```
module.exports = {
  plugins: [
    require('tailwindcss'),
    require('autoprefixer'),
  ]
}
```

Autoprefixer sirve para asegurarnos que los prefijos necesarios se coloquen después de haber realizado el proceso de generación del CSS de Tailwind. Por supuesto, si lo usas tendrías que haber instalado autoprefixer con npm previamente **(npm install -D autoprefixer).**

Comenzar a usar Tailwind en nuestro archivo CSS

Para usar Tailwind partimos del CSS del proyecto, que si estamos comenzando será un archivo con extensión .css que inicialmente estará vacío.

En ese archivo archivo CSS es donde comenzaremos a incluir las clases que nos proporciona TailwindCSS, usando unas directivas específicas en el código CSS, que podrán ser interpretadas y resueltas por el plugin de Tailwind de PostCSS.

Para nuestro proyecto, ese archivo lo vamos a colocar en la ruta: src/css/styles.css, pero realmente podría usarse cualquier ruta que creas conveniente, es decir, lo harás de tal manera que se adapte a tu proyecto y a tus costumbres.

En todo caso, el archivo CSS tendrá el siguiente código:

```
@tailwind base;
@tailwind components;
```

```
@tailwind utilities;
```

Este código hace uso del framework TailwindCSS, creando los bloques de código CSS generados por el framework:

- **Base**: es el código base del framework, comenzando por el típico "normalize".
- **Components**: son los componentes que te ofrece de base Tailwind.
- **Utilities**: las clases de utilidad, el auténtico corazón de TailwindCSS.

Ejecutar el compilado del CSS con PostCSS

Esta parte consiste más en la configuración del propio PostCSS, que en algo específico de Tailwind. Básicamente necesitamos ejecutar PostCSS para generar el código CSS que nos ofrece el framework TailwindCSS.

Para compilar este CSS necesitamos crear un script de npm. Esto ya lo explicamos en el proceso de configuración de PostCSS, pero básicamente consistirá editar el archivo "package.json", en el que colocaremos una configuración de scripts que incluya el comando de compilación, que sería algo parecido a esto:

```
"scripts": {
  "build": "postcss src/css/styles.css --output dist/css/styles.css"
},
```

Nota: Por supuesto, este script de npm lo podrás nombrar como desees. Nosotros hemos usado "build", pero podría ser cualquier otra cosa. Además, tienes que prestar especial atención a las rutas de los archivos de origen y de destino, para adaptarlas a tu proyecto. En este comando se está asumiendo que el CSS con el código de origen está en la ruta src/css/styles.css y, una vez compilado, el CSS resultante se almacenará en la ruta dist/css/styles.css.

Ahora podemos lanzar el proceso de build de PostCSS, para que genere el CSS de nuestro proyecto:

```
npm run build
```

Se habrá generado un archivo CSS de salida en dist/css/styles.css. No te asustes porque el CSS sea enorme!! luego veremos la manera de optimizar esta compilación y quedarnos con lo que realmente estamos utilizando en nuestra aplicación.

Desarrollar el HTML apoyándonos en TailwindCSS

En este punto por fin ya podemos probar Tailwind y crear un archivo HTML que use las clases que se han generado al compilar el CSS del proyecto.

Para ello vamos a generar un archivo index.html en la carpeta "dist" que se ha generado al compilar el CSS, con un código como este:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Demo de Tailwind</title>
  <link rel="stylesheet" href="./css/styles.css">
```

```
</head>
<body>
  <h1 class="text-3xl font-light text-center text-orange-600">Hola mundo!!</h1>
  <p class="mt-4 text-xl italic text-center text-gray-800">Lorem ipsum dolor sit, amet consectetur adipisicing elit.</p>
  <div class="text-center mt-4">
    <a href="#" class="bg-orange-600 text-white px-4 py-2 font-bold uppercase rounded-xl tracking-wider">Entrar</a>
  </div>
</body>
</html>
```

> Nota: En este momento no queremos pararnos a explicar las clases de TailwindCSS que hemos usado. La verdad es que la mayoría son bastante claras y podrás imaginarte lo que hacen, pero de todos modos no te preocupes porque las iremos abordando en los siguientes artículos.

Ahora puedes guardar el archivo HTML y abrirlo con el navegador, para ver el resultado que se ha producido. Si no se ha enlazado bien el CSS mira que el proceso de build te haya dejado el archivo de estilos en el lugar adecuado y que esté bien enlazado con la etiqueta LINK desde el HTML.

Como ves en el HTML hemos colocado 3 elementos, un encabezamiento, un párrafo y una división con un enlace. Todos ellos tienen clases generadas por el CSS de Tailwind, que se aplican para dar estilos a los elementos.

Como habrás podido imaginar, existen cientos (en realidad miles) de clases de utilidad en Tailwind para hacer prácticamente cualquier cosa que puedas necesitar, sin tener que editar los estilos, simplemente agregando las clases adecuadas.

Conclusión

Hemos podido realizar todos los pasos para configurar PostCSS en nuestro proyecto, con el plugin de Tailwind, de modo que podamos comenzar a compilar el CSS del framework. Luego hemos creado el primer HTML de un proyecto, simplemente para ver cómo se aplican las clases que nos ofrece TailwindCSS.

El proceso de configuración probablemente te parezca un poco largo, pero una vez lo tienes listo el flujo de desarrollo se realiza de manera muy ágil. En artículos sucesivos veremos todavía cómo optimizar el proceso un poco más y por supuesto, como reducir drásticamente el peso del código CSS generado.

En lo que respecta al HTML, la dificultad ahora radica en acostumbrarse a usar estas clases, aunque la verdad es que tienen nombres bastante intuitivos y es fácil deducir cuál tienes que usar, después de un tiempo trabajando con el framework. Eso sí, necesitarás conocer CSS porque la mayoría de clases aplican directamente a un atributo CSS y un valor determinado. Al principio se hará más necesario consultar la documentación constantemente, pero más adelante podrás naturalizar el proceso de asignación de clases.

En el siguiente artículo te explicaremos mejor el enfoque de Tailwind y por qué consideramos un framework mucho mejor diseñado que otros más tradicionales como Bootstrap. También te daré explicaré por qué te gustará y te resultará especialmente apropiado, en el caso que ya tengas cierto dominio sobre las CSS.

Tal como hemos dejado el ejemplo hasta aquí, el código con todos los archivos del proyecto se puede ver en este enlace de GitHub.

Videotutorial de TailwindCSS

Te dejamos con un videotutorial de Tailwind CSS que seguro te va a interesar. Es la primera clase del Curso de Tailwind CSS de EscuelaIT que impartimos en abierto en diciembre de 2020, que aborda Tailwind 2.0.

En este vídeo avanzamos bastante en el conocimiento del framework, abordando algunos temas adicionales a los que hemos tratado en este artículo y que veremos más adelante en el Manual de TailwindCSS. Son los siguientes:

- Qué es Tailwind CSS
- Arrancar con PostCSS
- Arrancar con Tailwind CSS
- Flujo de desarrollo
- Creación de componentes

Link de video de curso de taillwind

- https://www.youtube.com/watch?v=aUuYGDK6Oio&list=RDCMUCQZyH9sRRjvtZhMw-ldRvXQ&start_radio=1&rv=aUuYGDK6Oio&t=33

Por qué usar Tailwind CSS

Motivos por los que usar Tailwind CSS como framework de diseño. Cuáles son sus ventajas y puntos fuertes que te deben animar a usarlo para tus desarrollos.

En este artículo vamos a explicar algunos de los motivos por los que pienso que Tailwind es una excelente alternativa como framework de diseño web. Algunos de estos argumentos son absolutamente objetivos y están reflejados perfectamente en la página de inicio del framework. Otros son meramente personales y los cito en base a mi experiencia.

¿Por qué necesito un framework CSS / framework de diseño?

Hay una pregunta que me hago yo hace tiempo ¿por qué necesito Tailwind CSS o cualquier otro framework de diseño como Bootstrap? La verdad es que me siento muy cómodo desarrollando con CSS y nunca he visto la necesidad de usar algo por encima de CSS para mejorar mi desempeño.

> Nota: como herramientas de diseño hay que decir que los preprocesadores me conquistaron hace ya años pero la verdad es que cada vez los veo menos necesarios, en la medida que CSS ha ido evolucionando y existen otras herramientas como PostCSS que permiten un flujo de desarrollo un poco más personalizable y versátil. Por lo tanto, en mi último proyecto ya ni siquiera he implementado Sass con scss, sino directamente PostCSS y Tailwind.

Lo cierto es que nunca pude adaptarme a Bootstrap. Lo intenté algunas veces, pero no lo conseguí, no por el framework en sí sino por su enfoque. Incluso sirva de demostración que desarrollé la nueva versión de DesarrolloWeb.com en 1999 con Materialize CSS, un framework con enfoque idéntico a Bootstrap. A mitad del proyecto decidí eliminarlo del todo del desarrollo y enfocarme en el trabajo con CSS desde cero, para hacerlo totalmente mío y alcanzar la personalización y optimización que quería, sin tener que luchar contra los elementos.

Nunca me gustó colocar tantas clases CSS sobre las etiquetas, parecía casi como colocar estilos inline. Ni tener que aprenderme un nuevo sistema de rejilla, cuando ya conozco CSS y con flexbox o grid layout tengo todas las herramientas para hacer cualquier tipo de distribución completamente adaptada a las necesidades, y adaptable a los dispositivos.

Pero sin duda lo peor creo que era luchar contra el framework para poder modificar algunos estilos de los componentes. Buscando siempre selectores con más especificidad de los usados en el framework, que me permitieran cambiar los estilos a ciertas partes, o que mi css tuviera que crecer y crecer para poder personalizar los estilos que el framework aportaba.

Todo ello sin hablar de la mala optimización a la que condenas tu proyecto al usar estos frameworks, si es que no te lo curras para conseguir quedarte solo con las partes que necesitas y quitar el resto, que no siempre es tan fácil como podríamos creer.

Cómo conocí Tailwind CSS

Habitualmente uso Laravel como framework PHP para desarrollo web. Me parece un framework muy útil y sencillo de usar, pero no me gustaba demasiado el stack de tecnologías frontend que usaba hasta la versión 7. De hecho, con cada proyecto nuevo de Laravel lo primero que hacía era eliminar Bootstrap y VueJS para hacer mi propio CSS y usar Web Componente estándar en vez de componentes basados en una librería.

Entonces, me enteré de que Laravel iba a cambiar de Bootstrap a Tailwind y la verdad no esperaba que eso me hiciera cambiar la opinión que tenía, basada en los frameworks de diseño que ya conocía. Pero al estudiar Laravel 8 para usarlo en un nuevo proyecto decidí darle una oportunidad a Tailwind, para ver si me adaptaba. El resultado fue una grata sorpresa! En este artículo pretendo explicar por qué.

Clases de utilidad

Para captar las ventajas de Tailwind CSS hay que entender su enfoque de clases de utilidad. Básicamente consiste en que puedes crear cualquier estilo a cualquier elemento mediante un conjunto de clases sencillas y atómicas, que afectan a algo muy particular.

En Tailwind no hay tanto componentes como un ejército de clases de utilidad, que puedes usar para componer cualquier estilo. Los componentes los hace el desarrollador, bajo demanda para el proyecto en el que se encuentre y gracias a ello obtiene entre otras cosas versatilidad y diseños únicos.

Este enfoque de clases de utilidad ya está explicado en la categoría de Tailwind CSS. Si no lo entiendes te aconsejo su lectura. En cualquier caso, gracias a las clases de utilidad puedes mantener un control total sobre el sitio, algo que no consigues con otros frameworks CSS anteriores, como Bootstrap.

Para los que tenemos un control muy preciso de las CSS las clases de utilidad son un aliado, porque no nos obligan a aprender a hacer las cosas de nuevo. Algo tan sencillo para mi, como hacer un layout responsive, ya sea con grid o flex, y posicionar elementos de una manera radicalmente diferente para distintas anchuras de pantalla, lo puedo hacer en Tailwind usando los mismos conceptos y herramientas que manejo con CSS.

Personalización

Las clases de utilidad permiten personalización, a la hora de aplicarlas al diseño, para conseguir cualquier aspecto de un sitio web. Pero es que además, gracias a cómo está construido Tailwind, consigo también la posibilidad personalizar estas mismas clases.

Por ejemplo, TailwindCSS permite cambiar los nombres de las clases que sirven para hacer diseños adaptables (responsive) y agregar los breakpoints que sean necesarios para adaptarlos a cada proyecto. En otras palabras, es muy sencillo crear un breakpoint intermedio, entre dos saltos de los que ellos predefinen en el framework.

En resumen, gracias al archivo de configuración de Tailwind yo tengo absoluto control sobre qué clases de utilidad se generan, en número y variantes. Ajustando el generador de código CSS, es posible personalizar las clases de utilidad para agregar aquellas que nosotros necesitemos, para configurar colores, espaciados, saltos de mediaqueries, animaciones, etc.

Flujo más optimizado que otros frameworks

Estoy convencido de que los desarrolladores acostumbrados a Bootstrap encontrarán muchas ventajas al usar Tailwind. Para ilustrarlo usaré un ejemplo que seguramente os haya pasado.

Por ejemplo, tenemos un componente de acordeón o de lista en Bootstrap o Materialize. Entonces queremos que el margen sea tal o cual, en vez del configurado de manera predeterminada, o una negrilla en determinada anchura de pantalla. Lo que ocurre en muchos de estos casos es que te tienes que meter muy a fondo con el uso de selectores, para conseguir llegar a mandar sobre los estilos marcados por el framework. ¿No os ha pasado eso? A veces parece una lucha entre tú y el framework por llevarse un estilo determinado y acabas generando selectores locos para conseguir tus objetivos.

Esto no ocurre con Tailwind gracias a su enfoque de clases de utilidad. Como cada elemento tiene una serie de clases determinada y por separado, si algo no nos interesa en uno en concreto, simplemente quitamos esa clase y punto. No hay drama.

Además, Tailwind no te obliga a escribir más y más CSS, aparte de los componentes de base del framework, por lo que el peso de las hojas de estilo en cascada se mantiene siempre muy ajustado.

Es cierto que Tailwind te obliga a escribir más y más clases en tu HTML. Algo que con Bootstrap solucionabas con un "btn btn-primary", en Tailwind requiere colocar por lo menos media docena de clases muy específicas. En este sentido es una perversión aún mayor del paradigma de separación de contenido y presentación. Sin embargo, @apply viene al rescate en Tailwind CSS.

Optimización

Tailwind se usa mediante PostCSS y en el flujo de trabajo con este framework se incorporar numerosas herramientas para optimizar el código.

Al final, tu proyecto solamente tendrá el CSS que requiera, en función de las clases de utilidad que hayas usado en el código HTML o en tus templates. Esta optimización la veremos más adelante, en artículos sucesivos del Manual de Tailwind CSS.

Cómo Tailwind CSS ayuda a mitigar las desventajas de frameworks de diseño

Bueno, no dudo que un framework de diseño puede ofrecernos productividad en lo que respecta a la definición del aspecto, ya que no necesitamos editar el código CSS y el código HTML a la vez. Solamente añadiendo las clases adecuadas en el HTML conseguimos el objetivo, que es dejar la página bonita. En la medida que obtengamos experiencia, podremos alcanzar un mayor rendimiento.

Pero también veo algunas desventajas. La primera es ensuciar el HTML. El código HTML debería servir para especificar el contenido de las páginas y no la presentación. Con la declaración de tantas clases de utilidad estamos transfiriendo el peso de la presentación al mismísimo HTML, perdiendo la potencia que existe en la separación de contenido y presentación desde que tenemos CSS. Afortunadamente, Tailwind nos ofrece vías para mitigar esto. Estoy hablando de la directiva @apply, que abordaremos más adelante.

Frameworks CSS y Web Components

Finalmente, y esto lamentablemente no tiene solución, a mi me gusta usar Web Components, gracias a los cuales puedo construir botones, tarjetas, navegadores, listas... Muchos de los componentes de mis sitios web son Custom Elements, del estándar de Web Components.

En Web Components no siempre es posible o aconsejable basarse en estilos globales. Al contrario, usamos habitualmente estilos locales a cada componente, lo que nos permite una completa encapsulación y saber en qué lugar preciso están los estilos CSS que afectan a cada pieza del diseño.

Frameworks como Bootstrap o Tailwind CSS resultan de poca utilidad en componentes que usan Shadow DOM, ya que este marcado está completamente encapsulado.

Por eso considero importante que estos frameworks puedan optimizarse bien, como es el caso de Tailwind, para que no representen un peso muy representativo, más del imprescindible para los pocos estilos globales que pueda acabar usando en algunos proyectos.

Conclusión

Espero que esta disertación haya ayudado a más de uno a entender las ventajas, y las desventajas de los frameworks CSS y cómo Tailwind nos ofrece algunas mejoras con respecto a lo que antes había.

Después de tanto rollo, en el artículo siguiente cambiaremos al lado práctico, abordando algo tan importante como es la creación de componentes usando clases personalizadas y la directiva @apply de Tailwind CSS.

- Video de porque me gusta taildwind?? https://www.youtube.com/watch?v=5A-1A4-33fk

Directiva @apply en TailwindCSS

Cómo crear componentes con Tailwind CSS. mediante el uso de la directiva @apply, de modo que podemos reutilizar elementos aplicando el nombre de una única clase personalizada

En este artículo vamos a abordar una de las herramientas importantes que ofrece Tailwind CSS para la creación de componentes reutilizables, la directiva @apply, que aporta sencillez a la hora de aplicar estilo en ciertos elementos de diseño, que se van a usar en repetidas ocasiones a lo largo de todo un sitio web.

Como he dicho, lo que me parece horrible de los frameworks de diseño es el uso abusivo de clases en el HTML. Este detalle, aparte de echar por tierra el paradigma de separación de contenido y presentación, representa un problema serio cuando repetimos código constantemente.

Imagina que tienes 8 clases para definir un botón, o un encabezamiento, que has usado repetidas veces a lo largo de todo el sitio. ¿Qué pasaría si deseas cambiar esos componentes a lo largo de todo el sitio web?

Por eso existen las clases CSS, justamente para evitar que tengas que repetir dos veces una declaración de estilos, sin embargo, los frameworks CSS basados en "utility first" no aportan mucho en lo que respecta a reusabilidad de los componentes. En el caso de Tailwind CSS esta situación se ha solucionado de manera estupenga por medio de @apply, una directiva que permite englobar una serie de clases de utilidad bajo un mismo nombre, de modo que podamos usar eas clases para poder definir los estilos de nuestros componentes reutilizables.

Problema de la repetición de clases

Si observas el código HTML de ejemplos abordados anteriormente en el Manual de Tailwind observarás que habíamos definido un enlace con aspecto de botón usando todas estas estas clases:

```
<a href="#" class="bg-blue-600 text-white px-4 py-2 font-bold uppercase
rounded-xl tracking-wider">Entrar</a>
```

Gracias a esas clases conseguimos realizar un sencillo botón. Como imaginarás, es un estilo que pensamos repetir con frecuencia a lo largo de todo el sitio web. Sería un incordio tener que escribir todas esas clases, una a una, en nuestro HTML cada vez que queremos usar ese componente ¿no?.

Pero es más, como habíamos hecho ver, si por cualquier motivo se decide cambiar el botón, el color de todos los botones, o el tamaño del texto, tendrás que ir cambiando las clases por todo tu HTML, una y otra vez, por todo lugar donde hayas repetido el código. Creo que esto acabaría llevándonos a una experiencia de desarrollo poco amistosa.

Cómo solucionar la componetización con @apply

Usando @apply conseguimos solucionar el problema planteado de una manera sencilla. Podemos definir un código CSS como el siguiente para conseguir una clase CSS que tenga estos mismos estilos

```
@tailwind base;
@tailwind components;

.button {
    @apply bg-blue-600 text-white px-4 py-2 font-bold uppercase rounded-xl tracking-wider;
}

@tailwind utilities;
```

Nuestra clase se llama .button y tiene exactamente las mismas clases que las indicadas en el enlace anterior.

Gracias a esta definición ahora el botón lo podemos expresar con una única clase.

```
<a href="#" class="button">Entrar</a>
```

Ahora, si necesito cambiar el botón más adelante, solamente tendré que hacerlo en un único sitio!!

Organizar correctamente el código CSS basado en Tailwind

Es importante que tus clases, en las que usas @apply, se coloquen entre "@tailwind components" y "@tailwind utilities", ya que así tendrás una ventaja importante, que es la de sobrescribir el CSS por medio de las clases de utilidad.

Imagina que ahora deseas que determinado botón tenga un padding superior hacia la derecha y la izquierda. Sería simplemente asignarle la correspondiente clase de TailWind.

```
<a href="#" class="button">Entrar</a>
<a href="#" class="button px-8">Entrar</a>
```

Tenemos dos botones. El segundo tiene un padding diferente. Hemos reutilizado la clase "button" pero es perfectamente personalizable, dado que la definición de @apply la hemos colocado antes que la definición de "@tailwind utilities", que tomará precedencia gracias a la cascada del CSS.

Crear especializaciones de los componentes

También puede ocurrir que cierto botón tenga un color especial. Puedes tener botones verdes que expresen acciones de un tipo y rojos que expresen acciones peligrosas. En estos casos en los que quieres reutilizar el código del botón y hacer ciertas personalizaciones, puedes ir a una estrategia como esta:

```
.button {
    @apply px-4 py-2 font-bold uppercase rounded-xl tracking-wider;
}
.button-blue {
    @apply bg-blue-600 text-white;
```

```
}
.button-green {
    @apply bg-green-200 text-gray-800;
}
```

Hemos definido una clase "button" para el botón, con los estilos comunes a todos los botones, y una clase "button-green" y "button-blue" para dos tipos de botones en el sitio web. Vaya, es más o menos lo mismo que harías con tus declaraciones normales en CSS.

@apply aplica a todas las variantes de clases

No hemos hablado todavía de las variantes de Tailwind CSS (variants), que nos sirven para aplicar estilos para un determinado estado de los componentes, como por ejemplo "hover" o "focus", pero también para diseño responsive.

Este es un tema que tocaremos dentro de poco, pero para hacernos una idea, este estilo permite una caja sombreada, que cuando pasamos el ratón por encima cambia su borde y se levanta el sombreado.

```
<div class="p-4 border border-gray-200 rounded shadow hover:border-gray-300 hover:shadow-md">Este elemento tiene un hover</div>
```

Ahora podemos componetizar este elemento con @apply de esta manera:

```
@apply {
    p-4 border border-gray-200 rounded shadow hover:border-gray-300 hover:shadow-md;
}
```

Tailwind 2.0 en adelante, acepta cualquier tipo de variante en la directiva @apply, lo que nos permite una declaración sencilla de componentes complejos.

Nota: Si ves tutoriales en Internet que te dicen que esto no es posible, es que están desactualizados y abordan la versión 1.x de Tailwind CSS.

Conclusión @apply en TailwindCSS

Hemos visto cómo trabajar con la directiva @apply para mitigar la principal desventaja de los frameworks CSS para diseño web, que es la aplicación masiva de clases en los elementos para producir estilos determinados. Gracias a que Tailwind dispone de @apply podemos de una manera sencilla declarar estos conjuntos de clases, unificando bajo un componente con un nombre de clase personalizado.

Por cierto, @apply es una de varias directivas presentes en Tailwind. Otra es por ejemplo @tailwind que hemos visto nos sirve para volcar cierto CSS de una parte del framework en el CSS generado. Veremos otras directivas más adelante.

Pero antes de continuar con otras directivas, vamos a explicar algo tan importante y básico hoy en día como es el diseño responsive con Tailwind.

Variantes de las clases de utilidad

La magia de Tailwind CSS está detrás de las clases de utilidad y lo más impresionante del framework aparece cuando conoces las variantes que éstas ofrecen. Todas las clases de utilidad del framework se multiplican, en número y posibilidades, cuando sabes qué son las variantes y cómo aplicarlas para poder hacer diseños adaptables y conseguir aplicar estilos a todas las pseudoclases. En los siguientes artículos aprenderás a aplicar variantes para diseños responsive y para definir estilos cuando el usuario interacciona con la página.

Diseño responsive en Tailwind CSS

En este artículo explicamos cómo realizar diseño responsive, adaptable a todos los dispositivos, mediante las clases de utilidad de Tailwind CSS y con el enfoque Mobile First.

Una de las ventajas que nos ofrece Tailwind consiste en diseñar de modo adaptable sin tener que lidiar con las mediaqueries de CSS. No es que las mediaquieres sean difíciles, pero sí que agregan algo de dificultad al CSS básico y ajustarlas bien a veces ralentiza algo el flujo de desarrollo de un sitio web.

Tailwind soluciona el diseño adaptable por medio de clases de utilidad. Encontramos toda una serie de clases que nos permiten realizar el diseño adaptable o también "diseño responsive" como le solemos llamar.

En este artículo del Manual de TailwindCSS vamos a abordar todos los detalles sobre el diseño responsive, tal como lo hacemos usando este framework CSS.

Mobile first

Antes de ver cuáles son las clases que Tailwind nos ofrece para realizar el diseño adaptable, vamos a centrarnos en explicar un concepto aplicado en el desarrollo de Tailwind: "Mobile first".

Mobile First no es algo específico de Tailwind, sino es una manera de trabajar que aporta orden en la definición de un diseño y al desarrollo de la adaptabilidad a todas las pantallas. Por tanto, es algo más conceptual, que te puede ayudar a ti como desarrollador, para ser más ordenado y sistemático a la hora de desarrollar el responsive.

En Mobile First primero vamos a especificar los estilos pensando en móviles con pantallas pequeñas. Una vez tengamos un diseño que encaje bien en las anchuras reducidas de los dispositivos móviles vamos a ir pensando en pantallas cada vez mayores. Cuando el diseño se ve de manera fea para ciertas dimensiones, aplicaremos estilos para mejorarlo. Así hasta llegar a los monitores de grandes dimensiones de los ordenadores de sobremesa.

En el fondo, es algo tan sencillo como que al diseñar nos vamos a centrar primeramente en aplicar los estilos para los móviles y, a medida que apliquemos clases de Tailwind, iremos modificando los estilos desde un tamaño de pantalla dado hacia todos los superiores.

- Si no indicamos ninguna clase de utilidad de diseño adaptable, los estilos se aplicarán a todas las pantallas
- Si aplicamos una clase responsive, con alguna de las clases de utilidad de Tailwind, entonces aplicará ese estilo a ciertas dimensiones de pantalla y todas las pantallas de dimensiones superiores a esa. Es decir, una clase responsive afecta a una dimensión de pantalla y a todas las superiores.

Por ejemplo, si defino un estilo para pantallas medianas, este estilo aplicará a pantallas medianas, grandes y extra-grandes, pero no a las pantallas pequeñas.

Dimensiones de pantallas definidas de manera predeterminada en Tailwind

Tenemos definidas varias dimensiones de pantallas en Tailwind de manera predeterminada. Los nombres de las clases para cada una de estas dimensiones son los siguientes:

- **sm: Aplica a pantallas con dimensiones de 640 px para arriba.**
- **md: Para pantallas con dimensiones de 768 px en adelante.**
- **lg: Aplica a pantallas grandes, que comienzan en 1024 px.**
- **xl: Son para las pantallas muy grandes, que comienzan en 1280 px. 2xl: Este estilo afecta a pantallas de 1536 px y superiores.**

Nota: Ten en cuenta que el responsive 2xl solamente apareció en Tailwind 2. Es fácil caer en la trampa de pensar que "sm" sería el modificador para pantallas pequeñas. En realidad no es así, porque pantallas de 640 píxeles en adelante no son realmente pantallas de dispositivos móviles. Si quieres poner estilos para pantallas pequeñas en realidad no hace falta ningún modificador responsive, ya que la estrategia mobile first nos dice que los estilos asigandos sin identificadores responsive son aplicados a todos los tamaños de pantalla, incluidos por supuesto los móviles.

Por medio de la definición de nuestro propio tema personalizado estas medidas se pueden cambiar, así como podríamos definir estilos para tamaños intermedios de pantallas. Este tema lo veremos más adelante. Así como también sería posible cambiar el nombre de estas clases a nombres como "tablet", "desktop", etc. En este punto del manual no vamos a abordar configuración todavía, pero más adelante lo explicaremos.

Aplicación de clases responsive

Los nombres de clases responsive, como "sm", "md", etc. se combinan con cualquier otra clase del framework para generar clases como estas: "sm:text-center", "lg:font-bold" o "md:text-xl". Estas clases son las mismas que conocemos para centrar o cambiar el tamaño del texto, pero restringidas a determinadas dimensiones de pantallas. Por ejemplo:

- **sm:text-center hará que ese elemento tenga el texto centrado en dimensiones superiores a 640 px.**
- **lg:font-bold hará que el texto aparezca en negrita cuando la pantalla tenga 1024 px o más**

Podemos poner todas las clases responsive que sean necesarias en un elemento:

```
<h2 class="text-lg md:text-xl xl:text-2xl">Encabezado</h2>
```

Este encabezado tendrá tengo grande en todas las pantallas (text-lg), pero en pantallas medianas para arriba se ampliará a extra-large (md:text-xl) y ya en pantallas super grandes será de dos extra-large (xl:text-2xl)

Las clases responsive las podemos mezclar de todas las maneras posibles. Algo muy típico es mostrar u ocultar elementos según las dimensiones de la pantalla.

```
<div class="hidden md:block">Ocultar en pequeño y mostrar de mediano para arriba</div>
```

Esa división solamente se verá en pantallas de 768 px en adelante. Como puedes ver, el estilo sin modificadores responsive siempre se aplica "mobile first" y a continuación lo modificamos con los estilos necesarios para pantallas grandes.

Ejemplo de contenido responsive

En la documentación de Tailwind hay buenos ejemplos de elementos responsive que puedes tomar en cuenta para aprender muchos usos interesantes de sus clases de utilidad. Yo voy a hacer algo muy sencillo donde ver algunas clases en funcionamiento y practicar con ellas, a la vez que conocemos alguna otra alternativa para hacer diseño adaptable.

```
<div class="max-w-md p-4 mx-auto mt-4 bg-gray-200 sm:shadow-md sm::rounded-md sm:bg-gray-100 sm:p-6 md:bg-green-100">
  <h2 class="text-lg font-semibold text-center text-blue-500 sm:text-xl">Me adapto a todo</h2>
  <p class="mt-3 text-gray-600">Esta caja es adaptable. Diseño primero para las dimensiones pequeñas y voy aumentando para las grandes.</p>
</div>
```

Este bloque es un bloque adaptable, en el sentido que si es muy pequeño se muestra sin márgenes ni nada, al 100% del espacio disponible. En el momento que hay más espacio en la página el bloque se muestra como una tarjeta, con los bordes redondeados y una sombra.

Tiene una propiedad que no habíamos visto hasta ahora que es "max-w-md" que sirve para asignar una anchura máxima a este elemento (max-w = max-width), en este caso de tamaño "md", lo que equivale a 448 píxeles (28 rem). El color de fondo de la caja también cambia a medida que la anchura de la pantalla va creciendo.

Ya dentro tenemos un titular H2 cuyo texto aumenta de tamaño si la anchura de la página es de 640 px o superior.

Conclusión

El diseño responsive resulta bastante sencillo en Tailwind CSS, pero sobretodo ofrece un flujo de desarrollo muy ágil para el desarrollador.

Aplicar clases responsive permite que podamos definir de una manera super rápida el estilo para cada tipo de dispositivo y además nos obliga a trabajar siempre con mobile first, lo que hará que nuestras prácticas sean muy sistemáticas y ordenadas.

Usar variantes en Tailwind para aplicar estilos a pseudo-clases CSS

En este artículo aprenderás a usar variantes en TailwindCSS que afectan a estilos de las pseudo-clases, como hover, focus, etc. Además verás cómo configurar Tailwind para que estén disponibles las variantes en las clases de utilidad que necesites.

Vamos a abordar un tema muy interesante dentro de TailwindCSS, que consiste en la aplicación de variantes, las cuales nos pueden dar muchas posibilidades de configuración del aspecto de los elementos. Lo cierto es que ya habíamos visto ejemplos de uso de variantes en el Manual de TailwindCSS, aunque solo por encima. Ahora vamos a abordar de manera más detallada el uso de variantes y explicaremos cómo configurar el framework para aceptar las variantes que vayamos a usar.

En Tailwind disponemos clases de utilidad para prácticamente todo lo que podamos imaginar. Estas clases de utilidad además tienen una gran cantidad de variantes, de modo que se puede definir el aspecto de un elemento y especificar estilos distinos que se aplicarán en ese mismo elemento, pero en el momento en el que el usuario ha colocado el ratón por encima, cuando el elemento tiene el foco de la aplicación, etc.

La mayoría de las variantes, se apoyan en las conocidas pseudo-clases de CSS. Son las que vamos a tratar ahora. Aunque también hay variantes responsive como las que vimos en el artículo pasado.

> Las pseudo-clases de CSS sirven para implementar estilos en situaciones como el hover, active, focus, etc. El funcionamiento de las variantes de pseudo-clases es muy similar al de las variantes de responsive design que vimos en el artículo anterior. Simplemente se coloca el nombre de la variante, seguido de dos puntos (:) y la clase de utilidad que queremos aplicar.

```
<span class="hover:font-bold">Hover en negrita!</span>
```

El texto del span anterior se mostrará en negrita cuando el usuario pase el ratón por encima.

Ejemplo de uso de variantes hover, focus y active

Veamos el siguiente HTML, en el que aplicamos varias variantes a unos enlaces de una barra de navegación, para que tengan estilos aplicados a sus pseudo-clases.

```
<nav class="mt-4 bg-gray-200 shadow-lg">
  <ul class="flex">
    <li><a href="#" class="inline-block px-4 py-4 mr-1 font-semibold text-blue-500 hover:bg-gray-300 hover:text-blue-700 focus:font-bold focus:outline-none focus:bg-gray-500 focus:text-indigo-100 active:text-red-400">Productos</a></li>
    <li><a href="#" class="inline-block px-4 py-4 mr-1 font-semibold text-blue-500 hover:bg-gray-300 hover:text-blue-700 focus:font-bold focus:outline-none focus:bg-gray-500 focus:text-indigo-100 active:text-red-400">Contacto</a></li>
    <li><a href="#" class="inline-block px-4 py-4 mr-1 font-semibold text-blue-500 hover:bg-gray-300 hover:text-blue-700 focus:font-bold focus:outline-none focus:bg-gray-500 focus:text-indigo-100 active:text-red-400">Sobre</a></li>
  </ul>
</nav>
```

Las variantes de pseudo-clases que estamos usando tienen la forma como hover:text-blue-700, focus:font-bold o active:text-red-400. Creo que son bastante auto-explicativas. Así conseguimos un comportamiento bastante interesante de los estilos definidos para este navegador, de una manera rápida y sencilla.

Aquí no obstante hay dos factores importantes que nos dan pie a aprender más cosas de Tailwind:

- **Las variantes no siempre están instaladas en la configuración predeterminada. Por ejemplo, el caso de active:text-red-400 verás que no se aplica. Esto es porque el procesado predeterminado de Tailwind no la tiene en cuenta. Así que tendremos que aprender a configurar el framework para conseguir que exista esa clase.**
- **Recordar que Tailwind nos ofrece la directiva @apply para poder reutilizar estilos, algo que sería ideal aplicar en los enlaces de la barra de navegación. Si estás en Tailwind 1 tienes que tener en cuenta que la sintaxis varía un poco a lo que explicamos en el artículo de la directiva @apply. Luego lo veremos también.**

Configurar las variantes que deseamos en Tailwind

Recuerda que Tailwind tiene un archivo de configuración que nos permite especificar el comportamiento del generador de CSS. La configuración inicial del framework crea una cantidad de clases de utilidad gigantesca, pero no están todas las variantes de las clases de utilidad.

Ese archivo lo generamos en el capítulo inicial de Tailwind, donde explicamos cómo instalar y comenzar a usar el framework. Dicho archivo se llama "tailwind.config.js" y deberías encontrarlo en la raíz del proyecto.

> Más adelante hablaremos de la optimización y sobre cómo reducir el número de clases generadas por Tailwind. De momento vamos a ver lo básico para agregar nuevas variantes y más adelante explicaremos más detalles sobre cómo mantener y personalizar tus clases generadas, además de cómo reducir el peso del código CSS de Tailwind con otros plugins de PostCSS.

Dado que "active" no es una pseudo clase muy utilizada, el equipo de Tailwind ha decidido **no incorporarla de manera predeterminada en el CSS generado**. Para poder definir su existencia tenemos que especificarlo de manera explícita. Dentro de tailwind.config.js encontrarás una propiedad llamada "variants".

En ella colocamos todas las variantes que queremos que estén disponibles para cada clase del framework. Por ejemplo, nosotros queremos usar "active:text-red-400". Por lo tanto, tendremos que agregar la variante "active" a la clase del color de texto. Para ello usaremos un código como este:

```
variants: {
  textColor: ['responsive', 'hover', 'focus', 'active'],
},
```

Observarás que, además de 'active', hemos declarado que vamos a usar las variantes 'responsive', 'hover' y 'focus'. Esto es importante porque, en el momento que quieres agregar una variante, necesitas indicar toda la lista de variantes que vas a requerir aplicar a los colores de textos y no solo aquella en específico que quieres agregar.

Gracias a la declaración de estas variantes podrás pseudo-clases como md:text-blue-500, focus:font-bold outline-none, hover:text-blue-700, active:text-red-400 y cosas similares.

Variantes soportadas por Tailwind

Existen multitud de variantes soportadas por Tailwind que puedes ver listadas en la página de la documentación de configuración de variantes. Algunas de ellas, aparte de las que ya hemos visto, son "first", "last", "odd", "even", "visited"...

Además puedes instalar o crear tú mismo plugins de Tailwind que eventualmente puedan crear otras variantes que añadir en el framework.

Configurar estilos con @apply para pseudo-clases de CSS

Actualmente, desde el lanzamiento de Tailwind 2, es posible aplicar variantes también en la directiva @apply sin ninguna restricción, más allá que estén activadas en la configuración, tal como acabamos de decir.

El código que podrás tener en una declaración @apply será el mismo que colocas como clases en el elemento.

```
.button {
    @apply block px-6 py-3 mb-3 text-lg tracking-wider text-center text-gray-800
uppercase bg-yellow-100 rounded hover:bg-white sm:mr-5 sm:text-xl;
}
```

Cómo aplicar variantes en la directiva @apply en TailwindCSS 1

Sin embargo, en Tailwind 1 no era posible hacer este uso tan cómodo de las variantes de pseudo-clases en las declaraciones @apply. Si estás comenzando con TailwindCSS seguramente esto te resultará indiferente, porque no tengas que trabajar con la versión 1 del framework, pero si estás en proyectos poco actualizados te podría interesar. Igualmente, si estás leyendo tutoriales poco actualizados en Internet probablemente te encuentres código como el que vamos a ver que, a día de hoy, sería innecesario.

Entonces, en Tailwind 1 debes tener en cuenta que la directiva @apply que vimos con anterioridad no acepta el uso de pseudo-clases como focus, hover, etc. En cambio tienes que usar un poco de selectores de CSS estándar.

El código te quedará así:

```
.navlink {
  @apply inline-block px-4 py-4 mr-1 font-semibold text-blue-500;
}
.navlink:hover {
  @apply bg-gray-300 text-blue-700;
}
.navlink:focus {
  @apply font-bold outline-none bg-gray-500 text-indigo-100;
}
.navlink:active {
  @apply text-red-400;
}
```

Recuerda que este código deberías colocarlo en tu CSS antes de "@tailwind utilities; y que solamente sería necesario trabajar de esta manera en la versión primera del framework".

Como ves, en tu CSS estás definiendo la clase .navlink y a la vez diversas pseudo-clases para cada variante.

Ahora tus enlaces podrían tener este marcado mucho más limpio:

```
<nav class="mt-4 bg-gray-200 shadow-lg">
  <ul class="flex">
    <li><a href="#" class="navlink">Productos</a></li> <li><a href="#"
class="navlink">Contacto</a></li> <li><a href="#" class="navlink">Sobre</a></li>
  </ul>
</nav>
```

Sin duda ha mejorado el código HTML ¿no? esta es una excelente ventaja de Tailwind con respecto a cualquier otro framework de CSS.

Pero un detalle más. Gracias a que has usado el CSS para definir las pseudo-clases, no es necesario que tengas configurada ninguna variante en especial para este caso. Es decir, dado que hastas definiendo a mano el CSS para .navlink:active, ya sería un poco innecesario hacer esa configuración para agregar la variante active que aprendimos en el pasado punto de este artículo, ya que en el HTML no estamos usando finalmente esa pseudo-clase.

Conclusión

Hemos visto cómo trabajar con variantes en el framework Tailwind para aplicar de una manera muy ágil y cómoda estilos basados en pseudo-clases de CSS, sin salirnos del HTML.

Hemos explicado que no todas las variantes están activadas para todas las clases de utilidad, por lo que es posible que tengas que realizar cambios en la configuración del framework para disponer de ellas.

Finalmente hemos mencionado que antes (Tailwind 1) era imposible usar variantes en las directivas @apply, por lo que teníamos que acudir a una fórmula distinta. Esto ya no ocurre en la versión actual de TailwindCSS.

En el siguiente artículo continuaremos aprendiendo la magia del framework, sacando partido a la personalización de Tailwind CSS.

Personalización y optimización

Resulta fundamental aprender a personalizar Tailwind CSS. El framework está pensado para llegar a la mayoría de las necesidades de los desarrolladores, pero sabemos que los diseños son y deben ser únicos. Tailwind permite llegar a lo que tú quieras gracias a la personalización del framework. Cuando aprendes todo lo que es posible configurar verás que Tailwind es como un generador de tu propio framework personalizado. Pero además, observarás que la enorme cantidad de clases de utilidad es abrumadora y entenderás que sin optimización no se llega a ningún lado. Lo bueno es que Tailwind CSS está desarrollado encima de PostCSS y todas las herramientas para optimizar el código de tus estilos están a tu disposición para dejar el peso de tu framework reducido exactamente a lo que necesitas.

Cómo personalizar nuestro tema de diseño con Tailwind

Cómo personalizar el tema de diseño para Tailwind se adapte a las necesidades de cada sitio web. Aprenderás a sobreescribir los estilos predeterminados del framework y a extender las clases de utilidad creando todas las que necesites.

Hasta el momento en el Manual de Tailwind CSS hemos aprendido a usar las clases de utilidad que nos facilita el framework, lo que ha estado muy bien. Sin embargo, queda mucho por ver y en este artículo comenzamos con una de las partes más interesantes de la herramienta, que distingue a Tailwind de cualquier otro framework CSS existente hasta la fecha: su capacidad de personalización.

Tailwind ofrece una cantidad enorme de clases de utilidad. Su enfoque es ofrecer gran cantidad de clases para que tú no tengas que editar prácticamente nada de CSS, salvo cuidarte para no repetir código. En esencia este workflow ya lo explicamos en el artículo anterior, dedicado a explicar el enfoque de clases de utilidad.

Ahora bien, muchas de las clases de Tailwind están pensadas para servir de modo general, para que encajen en la mayoría de los proyectos, sin embargo puede que no sea exactamente lo que necesitamos para algún diseño en particular. Por ello en ocasiones es necesario **personalizar el generador de código CSS para que se adapte a unas necesidades concretas**. Esta tarea la realizaremos dentro del archivo de configuración de Tailwind, que habíamos generado anteriormente en el proyecto, llamado "**tailwind.config.js**".

Este archivo de configuración ofrece una gama de posibilidades impresionante. Ahora vamos a explicar varias de sus posibilidades, centradas en la declaración "themes".

Configuración predeterminada de Tailwind

Recuerda que el archivo de configuración de Tailwind lo generamos con el comando:

```
npx tailwind init
```

Como resultado se generó el archivo "tailwind.config.js" con unas declaraciones pero sin ningún dato. Tal como está, es lo mismo que no tener nada y básicamente lo que hará el framework es tomar la configuración predeterminada.

Es una buena idea ver cómo es esa configuración por defecto, ya que nos va a servir de guía para que nosotros podamos crear las configuraciones personalizadas. Para ello lanzamos el mismo comando npx tailwind init, indicando otro nombre de archivo donde almacenar la configuración predeterminada y el flag --full.

```
npx tailwind init tailwind-full.config.js --full
```

Esto generará un archivo llamado tailwind-full.config.js que no servirá para nada más que permitirnos observar la configuración completa y predeterminada de la generación de clases de utilidad presentes en Tailwind. Son más de 800 líneas de código en las que se modelizan absolutamente todas las posibilidades del framework para la generación de las clases.

Esas posibilidades se mezclan para generar miles de combinaciones de clases distintas, las que ofrece Tailwind si no lo optimizamos.

Para editar estas configuraciones tenemos dos alternativas posibles.

- Sobreescribir configuraciones existentes, o
- Añadir nuevas configuraciones personalizadas, que se unen a las existentes.

Sobreescribir configuraciones de Tailwind

Vamos a comenzar explicando cómo sobreescribir las configuraciones ofrecidas de manera predeterminada.

Al principio del archivo de configuración global encontramos la declaración "theme", que empieza con este código:

```
theme: {
    screens: {
      sm: '640px',
      md: '768px',
      lg: '1024px',
      xl: '1280px',
    },
    // ...
}
```

Bien, los "screen" definidos en el theme son los distintos tamaños de los breakpoints que podemos usar para el diseño responsive, que el framework transformará en mediaqueries sobre las que puedes definir estilos personalizados a distintos tamaños de pantallas.

Para modificar este apartado de configuración tenemos toda la flexibilidad imaginable:

- Podemos cambiar los nombres de los saltos, por otros que nos resulte más sencillo de recordar. Por ejemplo, cambiar "md" por "tablet" o "lg" por "desktop".
- Podemos cambiar el valor en píxeles de cada breakpoint. Por ejemplo, igual necesitamos que el primer breakpoint se produzca a los 500px en vez de 640px.
- Podemos eliminar saltos si no los vamos a utilizar
- Podemos agregar nuevos breakpoints si los necesitamos.

Obviamente, cuantos más saltos tengamos, más combinaciones de todas las clases de utilidad de Tailwind existirán y por tanto aumentará el tamaño del archivo de CSS. Por tanto, una buena vía para comenzar la optimización sería reducir el tamaño de breakpoints. Pero tranquilo, tampoco pasa nada si los dejas, puesto que luego con PostCSS podremos hacer un vaciado de todos los estilos que no vamos a utilizar.

Por ejemplo, podríamos definir estos breakpoints (solo a modo de ejemplo, no digo que sean o no los adecuados para un diseño en particular).

```
theme: {
  screens: {
    bigMobile: '420px',
    tablet: '768px',
    desktop: '1024px',
  },
  extend: {},
},
```

Ten en cuenta que después de cambiar el fichero de configuración tendrás que recompilar el código CSS generado para que tenga efecto y si usas "watch" tendrás que volverlo a arrancar.

Gracias a esta configuración tendrás algunas clases como: *tablet:border-red-700, tablet:hover:bg-green-900, bigMobile:py-3, desktop:mt-5…* Sin embargo, **se habrán eliminado las antiguas convenciones del framework para breakpoints** como "sm", "md", etc., ya que no están ya definidas como propiedad en el objeto "screens".

Añadir nuevas configuraciones personalizadas al tema

Ahora vamos a aprender cómo podemos **agregar configuraciones al tema existente**, en vez de sobre escribir las que se ofrecen de manera predeterminada. Esto es muy útil en gran cantidad de casos, porque muchas veces simplemente queremos añadir un color que no está presente en Tailwind, por ejemplo, pero no queremos perder todas las definiciones de colores que ya existen de antemano.

Vamos a imaginar que queremos trabajar con un color primario y uno secundario. Esos colores puede que no estén justamente en la paleta de Tailwind (siempre habrá uno parecido, pero quizás necesites usar un RGB exacto forzado por la imagen de marca de tu cliente), pero además es muy probable que quieras llamarles justamente así "primary" y "secondary" a lo largo de todo tu diseño.

Podríamos pensar que esta situación se solucionaría de la siguiente manera:

```
theme: {
```

```
  colors: {
    primary: '#fe3',
    secondary: '#ba0',
  },
},
```

Sin embargo, si pruebas esa configuración verás que, aunque funciona, se habrán eliminado todas las clases de todos los colores existentes. Solucionarlo es fácil, mediante la extensión en vez de la sobreescritura.

```
theme: {
  extend: {
    colors: {
      primary: '#fe3',
      secondary: '#ba0',
    },
  },
},
```

En el código anterior habrás visto que hay justamente una propiedad llamada "**extend**", que nos sirve para agregar configuraciones al tema. Justamente lo que necesitamos. Ahora podemos usar clases con los colores "primary" y "secondary", pero todavía podemos seguir trabajando con los colores definidos en el framework como "gray-400".

```
<div class="p-4 bg-primary text-gray-400">
  Esto es un <span class="text-secondary">test!!</span>
</div>
```

Existen muchos otros casos en los que la extensión de un tema **será la solución más óptima**, por ejemplo cuando necesitamos definir una medida extra para una anchura. Por ejemplo necesitamos definir medidas de unos elementos en un tamaño que no está definido en el framework. Deseamos agregarlo, pero no deseamos prescindir de las medidas que ya estaban representadas anteriormente en las clases de utilidad.

> Como decíamos, no necesitas preocuparte porque la extensión produzca todavía más código CSS de clases de utilidad y por lo tanto pese más el archivo de CSS generado por Tailwind. Más adelante te explicaremos cómo optimizarlo para que se vacíe de todas las clases que no hayas llegado a usar en tu proyecto.

Conclusión

Hemos aprendido a mejorar los temas de diseño de Tailwind por medio de la configuración definida en la propedad "theme" de tailwind.config.js. De esta manera **podemos aumentar la versatilidad que nos ofrece el framework** de diseño, sin quedarnos restringidos a un conjunto de opciones predefinidas.

Cuando vi estas posibilidades por primera vez me resultaron especialmente interesantes, ya que me ayudan a mantener un control estrecho de los estilos y posibilidades de diseño que se pueden conseguir. Creo que representan una diferencia fundamental con otros frameworks, que hacen que Tailwind gane bastantes enteros.

Para acabar sólo me queda comentar que podríamos pretender ser muy minuciosos y quitar manualmente del archivo de configuración todas las declaraciones que no pensamos utilizar, como colores, tamaños, breakpoints, etc. Con eso ganaríamos en optimización. La verdad es que no deja de ser cierto, pero no es la mejor manera de conseguir ahorrar espacio en el CSS generado. En el siguiente artículo dedicado a la optimización de Tailwind veremos opciones mejores y más sencillas de producir

Optimización del CSS con Tailwind

Cómo optimizar el CSS generado por Tailwind, para asegurarse que, al llevar a producción el sitio web, no se incluya más CSS que el estrictamente necesario, consiguiendo que el archivo de estilos pese muy poco.

Si observamos el código CSS generado al usar Tailwind quizás nos quedemos con la boca abierta. De manera predeterminada ocupa nada más y nada menos que varios megas!!! 3 MB o más, dependiendo de la versión del framework y de las personalizaciones de Tailwind CSS que vengas realizando. Ese tamaño para un archivo CSS es inadmisible para un proyecto en producción, por muy grande que sea el sitio web y aunque estemos usando muchas de las clases de utilidad de Tailwind.

El motivo de ese enorme peso es la **cantidad de clases de utilidad que nos proporciona Tailwind.** Por mucho que nos empeñemos en sacarle partido al framework y usar cientos de clases **es muy probable que no lleguemos a emplear ni un 1% de su código**!! Por ello es esencial que podamos realizar las debidas **optimizaciones del código CSS generado**.

Como explicamos en el artículo anterior, una posible idea sería optimizar el tema gráfico, con lo que podríamos eliminar muchas clases que no necesitamos. Sin embargo, nos llevaría un gran trabajo manipular de manera tan minuciosa el archivo de configuración y seguramente sería complicado mantenerlo al día, cada vez que necesitemos incorporar nuevas clases. No sería práctico y todavía sería más complejo en proyectos donde participasen varias personas.

Otro detalle que se nos podría ocurrir sería minimizar el código y, aunque resulte un ahorro representativo, sigue sin ser suficiente. En este artículo veremos cómo conseguir optimizar de una manera más radical, para llegar al menor peso posible en nuestro CSS.

> Actualmente en Tailwind 2 se ha simplificado muchísimo la operativa para la optimización del CSS generado por el framework. Afortunadamente, ahora ya no es necesario configurar PostCSS y el plugin de "Purge CSS", ya que viene incorporado a Tailwind de manera predeterminada. Si quieres aprender a configurar PostCSS a mano te recomendamos acceder a este enlace: PostCSS.

PurgeCSS

Comenzaremos usando una herramienta llamada PurgeCSS, que nos permite **eliminar del código CSS todas las clases que no están siendo usadas desde nuestro HTML**.

Esta herramienta se encuentra disponible como plugin en PostCSS y su documentación la puedes encontrar en purgecss.com/plugins/postcss.html

Cómo funciona PurgeCSS

Para que PurgeCSS haga bien su trabajo debemos comenzar por entender cómo funciona. Este plugin hace un análisis de todo el texto de los archivos que puedan tener HTML, buscando cadenas.

> No queremos decir que analice solamente los archivos .html, sino todos los archivos que puedan tener código HTML dentro, como los .php. Por supuesto, esto se puede configurar, como veremos enseguida.

En la búsqueda de cadenas Purge CSS se encargará de localizar todas las clases de utilidad de Tailwind que estés utilizando, para poder luego optimizar el CSS generado. Ten en cuenta que **no hace análisis de aquello que está escrito en los atributos class, sino cadenas en general en todos los archivos**.

Cómo escribir el código HTML para que PurgeCSS localice todos los nombres de clases usadas

Por lo tanto, los nombres de las clases de CSS que va a dejar en el archivo de código generado, deben estar escritos, tal cual, en el texto del archivo compilado. Esto quiere decir que no deberíamos hacer algo como esto:

```
div class="{{ $variable_clase_atributo }}">Elemento</div>
```

Puede que al procesarse el template unas veces se interpole $variable_clase_atributo colocando la clase "text-red-500" y otras veces "text-green-400". Sin embargo, PurgeCSS no tiene cómo saber los posibles nombres de esa variable $variable_clase_atributo. Por tanto, fallará y no mantendrá esas clases en el archivo CSS purgado.

Sería mejor un código como este:

```
<div class="@if $estado == 'error' text-red-500 @else text-green-400
@endif">Elemento</div>
```

Elemento

De este modo, el nombre de la clase estará escrito en el template y PurgeCSS será capaz de detectarlo y hacer que dicha clase permanezca en el código resultante optimizado. Espero que eso se entienda, para que no haya sorpresas!

Cómo configurar los nombres de los archivos con el código HTML

Ahora veamos cómo indicarle a Tailwind qué nombres de archivos debe de procesar para buscar nombres de clases en su texto. Básicamente tendrás que indicarle todos los patrones de rutas donde haya código HTML. Estos archivos pueden ser:

- Ficheros HTML
- Ficheros PHP que tengan código HTML embebido
- Ficheros de vistas de un framework como Vue, React o Angular

La configuración de este listado de patrones de rutas se hace dentro del archivo **tailwind.config.js**, con un código como el siguiente:

```
module.exports = {
  purge: [
    './src/**/*.html',
    './otra_ruta/**/*.jsx',
  ],
  theme: {},
  variants: {},
  plugins: [],
}
```

Ejecutar el build en modo producción

Tailwind ya contiene la funcionalidad para activar automáticamente PurgeCSS por nosotros, a fin de ponerlo en marcha para limpiar de clases no utilizadas el código resultante. El único detalle es que **PurgeCSS sólo se activa en modo producción**.

> Esto tiene mucho sentido porque al procesarse PurgeCSS Tailwind se toma mucho más tiempo para generar el CSS, lo que provoca tiempos de espera innecesarios para ver el código en funcionamiento cuando introducimos cambios durante la etapa de desarrollo.

De modo que, para conseguir que PurgeCSS entre en funcionamiento, vamos a tener que definir una variable de entorno de NodeJS en "NODE_ENV" con el valor "production".

Para facilitarnos la labor vamos a cambiar el script npm para build. Con estos tres scripts npm tenemos una buena cantidad de variantes de ejecución de PostCSS:

```
"scripts": {
  "build": "NODE_ENV=production postcss src/css/styles.css --output
dist/css/styles.css",
  "dev": "postcss src/css/styles.css --output dist/css/styles.css",
  "watch": "postcss src/css/styles.css --output dist/css/styles.css --watch"
},
```

El primero produce el build, enviando la variable de entorno "NODE_ENV=production" para que se ejecute PurgeCSS. El segundo quizás es un poco innecesario, pero generaría el CSS sin hacer ninguna purga. El tercero es el watch, que ejecuta la generación de estilos cada vez que cambian los archivos fuente del CSS.

Ahora al hacer "npm run build" realizaremos la **optimización del CSS**, básica para que nuestro sitio web no tenga cargas innecesarias de clases que no se están utilizando.

Cambios en Tailwind 2.0 para usuarios de Tailwind 1.x

Como ya hace tiempo que se lanzó Tailwind CSS 2 es muy probable que no te interese el siguiente bloque, que afectará más a los usuarios de Tailwind 1.x que no hayan migrado todavía a la nueva versión.

> La principal novedad de Tailwind 2 con respecto a Purge CSS es que este plugin de PostCSS ya se encuentra instalado de casa con Tailwind. Pero los usuarios de Tailwind 1 lo tienen que instalar de manera explícita.

En Tailwind 1 al usar purge, si ejecutas el build podrás encontrar un "warn" con un mensaje como este:
The conservative purge mode will be removed in Tailwind 2.0.

Esto quiere decir que Tailwind en la versión 2.0 va a cambiar de funcionamiento en lo que respecta a la purga de código CSS. Actualmente solo se hace el purge de la capa "utilities", pero en la versión 2 pasará a purgar todas las "layers".

> Nota: las "layers" o capas de Tailwind son las que colocas en las directivas @tailwind en el archivo CSS: "base", "components" y "utilities". Cada una contiene una cantidad de CSS. Hablaremos de ellas con detalle un poco más adelante en este manual.

Para quitar ese warning en Tailwind 1.x simplemente tienes que descomentar una línea en el atributo "future", de este modo:

```
future: {
  // removeDeprecatedGapUtilities: true,
  purgeLayersByDefault: true,
},
```

Esto quiere decir que ahora va a purgar todas las "layers" por defecto. Sería básicamente lo que haría en la versión 2 de Tailwind, purgar todas las layers existentes. Sin embargo, si quieres mantener la configuración anterior y purgar solamente una de las capas, utilities, sigues pudiendo hacerlo. Para ello simplemente necesitas activar el array "layers" dentro de Purge.

El código del purge te quedaría más o menos así:

```
purge: {
  layers: ['utilities'],
  content: [
    './src/**/*.html'
  ],
},
```

Conclusión

Con estas configuraciones habrás podido **realizar una optimización muy relevante** en tu código y podrás distribuir un CSS con el tamaño perfectamente ajustado. De esta manera, el archivo de código CSS resultante de usar Tailwind ocupará unas pocas KB. Para un sitio pequeño puede que en torno de 30 KB, para uno grande podría duplicarse ese tamaño, aunque todo depende de la cantidad de clases de utilidad que estés usando verdaderamente en el código de tu proyecto.

Esta posibilidad de optimización de Tailwind, ya incorporada de casa en el framework, es una de las **ventajas fundamentales que encontramos en Tailwind con respecto a otros frameworks tradicionales basados en**

componentes como Bootstrap. Es decir, con Tailwind CSS no es necesario enviar a tus usuarios una cantidad de código CSS enorme, sino solamente el necesario.

Profundizando en Tailwind CSS

Ahora vamos a dedicarnos a abordar temas importantes del framework, que aparecen cuando quieres profundizar un poco y conseguir dar respuesta a diversas necesidades comunes en el diseño web.

Crear un Layout con Tailwind CSS

Cómo hacer layouts con Tailwind CSS para la distribución del contenido de la página por columnas, adaptable a los distintos tamaños de pantalla de los usuarios, para un diseño responsive.

En este artículo del Manual de Tailwind CSS vamos a hacer una práctica fundamental para quien está comenzando un desarrollo de un sitio web con el framework y necesita **crear la estructura básica del layout del sitio**, es decir, la división del contenido mediante filas y columnas.

Afortunadamente para todos los que ya conocen CSS, **el diseño de un layout con Tailwind no difiere mucho de lo que haríamos con nuestros propios conocimientos de CSS**, simplemente se trata de saber qué clases de utilidad debemos aplicar.

Obviamente, a la hora de diseñar tu propio esquema de distribución del contenido, tendrás que usar tus propias columnas con medidas personalizadas a tu gusto. En este artículo vamos simplemente a aportar algunos ejemplos y conceptos básicos que debes conocer para poder realizar este trabajo. Posteriormente, con los conocimientos adquiridos, podrás adaptar el layout a tus necesidades.

Contenedor principal

La clase por la que tenemos que comenzar cualquier layout es "**container**". De hecho es la típica en cualquier diseño que hayas realizado antes, ¿no?

La clase container es adaptable al tipo de pantalla y ya hace los saltos automáticamente en los breakpoints definidos por el framework.

- Para pantallas pequeñas tiene un width: 100%;
- Para pantallas sm o superior está definido como max-width: 640px;
- Para pantallas md en adelante: max-width: 768px;
- Para pantallas en adelante: max-width: 1024px;

- Para pantallas en adelante: max-width: 1280px;

Además, lo más normal es que quieras centrar el contenedor en la pantalla, por lo que no solemos usar esa clase de manera única, sino que se combina con otra clase llamada "mx-auto".

```
<div class="container mx-auto bg-blue-500">
  Contenedor centrado
</div>
```

En el ejemplo anterior hemos agregado además un fondo, más que nada para que al probar el ejemplo puedas redimensionar y ver perfectamente el tamaño del contenedor, a medida que vas haciendo mayor o menor la ventana del navegador.

Dado que el centrado es una configuración muy frecuente, si deseamos que los contenedores estén centrados de manera predeterminada, podemos especificarlo en la configuración del tema:

```
theme: {
  container: {
    center: true,
  },
},
```

> Recuerda que ya aprendimos en un artículo anterior los pasos para personalizar el tema de diseño del CSS generado mediante Tailwind.

Personalizar el contenedor con una variante responsive

Si no te gusta que el contenedor ocupe solamente una parte de la anchura disponible, puedes perfectamente eliminar la clase container. El único problema real ocurrirá con las pantallas que son muy anchas, que tu página se va a estirar demasiado hacia los lados y seguramente no quede muy bonita.

Por lo tanto, al menos lo recomendable sería que la clase container se aplicase para las pantallas muy grandes. Esto lo puedes conseguir mediante los modificadores responsive, que ya conoces.

```
<div class="px-4 mx-auto xl:container">
  Contenedor centrado
</div>
```

En este caso, el contenedor ocupará toda la ventana del navegador, excepto para pantallas de 1280 píxeles, a partir de las cuales el contenedor se limitará a 1280px.

Podríamos haber conseguido este mismo efecto con estas otras clases de utilidad:

```
<div class="w-full max-w-screen-xl mx-auto px-4">
  Contenedor centrado con 1280px de máxima anchura
</div>
```

En este caso hemos usado w-full para aplicar la anchura al 100% y luego max-w-screen-xl que fuerza el estilo "max-width: 1280px".

Crear un layout con distribución en columnas con TailwindCSS

Como sabrás si dominas las hojas de estilo en cascada, las clases de utilidad de Tailwind no hace nada de magia, simplemente aplican determinados estilos CSS. Por tanto, para crear una estructura de columnas para un layout se hace de manera similar a como lo harías con CSS escrito a mano.

Simplemente debemos escoger el sistema de rejilla estándar con el que deseamos trabajar y tener en cuenta el **enfoque Mobile First de Tailwind**. Veamos todos estos detalles con calma.

Escoger el display para la división en columnas

Dadas las especificaciones del CSS actual, los dos display más apropiados para hacer un layout en columnas son "flex" o "grid". Cualquiera de los dos es estupendo para los objetivos de prácticamente cualquier distribución de contenido.

Quizás Flex Box es un poco más sencillo de utilizar, aunque lo cierto es que **Grid Layout sería más apropiado para lo que es una distribución de rejilla**. Sin embargo, Flex tiene un poco más de tiempo y como consecuencia está mejor soportado por navegadores antiguos como Internet Explorer. Aunque, si no nos preocupa IE, entonces Grid y Flex son prácticamente equivalentes en términos de soporte.

En cualquier caso, como TailwindCSS usar el propio sistema de CSS para la distribución en filas y columnas, **no requiere que tengamos que aprender a usar un sistema de rejilla distinto**, sino simplemente usar las clases de utilidad apropiadas.

Enfoque Mobile First

Para aplicar la distribución de columnas a partir de determinada anchura de página hay que tener en cuenta el enfoque "mobile first".

La clase container será suficiente para anchuras de pantalla pequeñas, en las que generalmente no deseamos que exista ninguna distribución por columnas. Lo normal será aplicar el layout de columnas a partir de un breakpoint desde el cual tengamos la anchura suficiente para que esas columnas se vean bien, en función del contenido que debamos mostrar.

Ejemplo práctico de distribución de columnas

Para este ejemplo vamos a usar Flexbox y además realizaremos el salto a dos columnas en dimensiones "md", que equivale a 768px.

```
<div class="container md:flex">
  <main class="px-4 mb-6" flex-grow>
    <p class="mb-4">Lorem ipsum dolor sit amet, consectetur adipisicing elit. Culpa sed quas non voluptates labore commodi eos neque dicta cupiditate, ipsam libero, beatae vel suscipit nihil nemo cumque magnam similique doloremque?</p>
    <p class="mb-4">Nesciunt magnam excepturi tenetur eum magni mollitia amet at neque. Minima placeat maiores laudantium quisquam molestiae corporis et possimus iusto suscipit illum?</p>
  </main>
  <aside class="px-4 md:flex-none md:w-64">
    <div class="px-4 py-2 mb-2 bg-red-300">Titulo de aside</div>
    <p>Este es el aside</p>
  </aside>
</div>
```

- El elemento principal tiene la clase "container md:flex". Quiere decir que se mostrará como un contenedor, pero que a partir del breakpoing "md" su display pasará a ser "flex".
- El elemento MAIN tiene una clase "flex-grow" que hace que este elemento crezca todo lo que pueda. Es decir, si Flexbox encuentra que tiene espacio disponible sobrante, lo asignará a esta columna.
- El elemento ASIDE tiene una clase "flex-none" que hace que la anchura indicada para este elemento "w-64" mande sobre la distribución de espacios de Flexbox. Por lo tanto, la anchura de este elemento será siempre fija. Sin embargo, nos fijamos que la anchura está expresada con "md:w-64". Esto quiere decir que esa anchura solo se aplicará cuando el tamaño de la ventana sea de "md" (768px) para arriba. Esto es importante porque no queremos alterar la anchura del elemento para pantallas menores de "md", de modo que ocupe todo el ancho disponible en pantallas pequeñas.

La que acabamos de ver es **una entre mil posibles configuraciones del sistema de columnas**, ya que somos nosotros los encargados de elegir los tamaños adecuados y la disposición de cada columna, por medio de las clases de TailwindCSS adecuadas para cada caso. **Tenemos toda la flexibilidad para elegir nuestra propia configuración**, ya que no necesitamos ceñirnos a un sistema de rejilla previamente creado. Por supuesto, si Tailwind no tiene las personalizaciones de diseño exactas que nosotros queremos, las podemos añadir en la personalización del tema.

Configurar anchuras proporcionales

Otra de las posibilidades de creación del layout es escoger anchuras proporcionales para cada columna, algo que se puede hacer con otras clases de utilidad.

Para ello tenemos las clases w-{fraccion} como w-1/2 o w-1/4. Esto nos puede venir bien en diseños fluidos o en general al dividir los espacios proporcionalmente, en vez de maquetar una de las columnas con un ancho fijo y asignar el resto del espacio a la otra, como vimos en el ejemplo anterior.

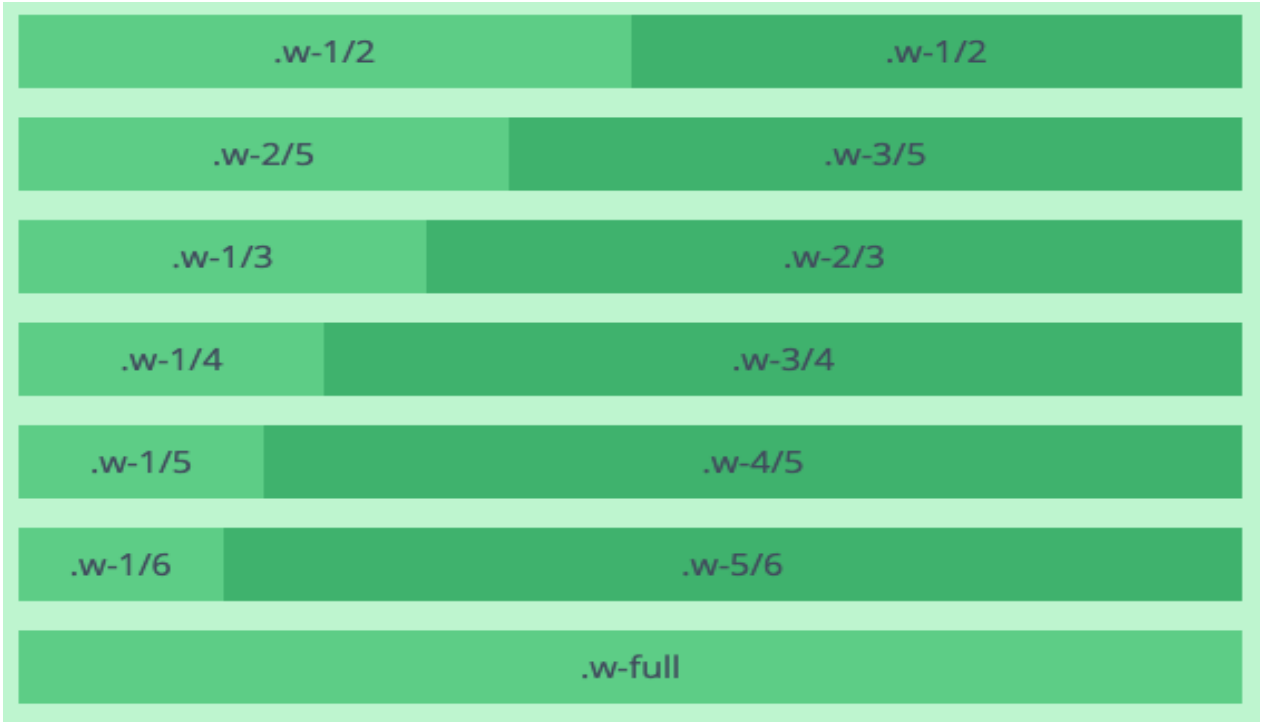
Por ejemplo, si queremos hacer dos columnas que dividan el espacio al 50% se haría así:

```
<div class="flex mb-4">
  <div class="w-1/2 p-2 text-center bg-green-400">.w-1/2</div>
  <div class="w-1/2 p-2 text-center bg-green-500">.w-1/2</div>
</div>
```

Podemos usar más columnas que distribuyan el espacio de manera fluida mediante entre más elementos. Por ejemplo así tendríamos un elemento que ocupa la mitad del espacio y dos elementos que ocupan la mitad restante.

```
<div class="flex mb-4">
  <div class="w-1/2 p-2 text-center bg-blue-400">.w-1/2</div>
  <div class="w-1/4 p-2 text-center bg-blue-500">.w-1/4</div>
  <div class="w-1/4 p-2 text-center bg-blue-600">.w-1/4</div>
</div>
```

En realidad puedes usar combinaciones de cualquier tipo, con fracciones que van hasta sextos. Algunos otros ejemplos los puedes ver en esta imagen:



Y por supuesto, en la configuración del framework, en el archivo "tailwind.config.js" puedes organizar cualquier división en fracciones que veas conveniente, para que se creen las correspondientes clases de utilidad.

Conclusión

Hemos visto algo tan importante como la distribución de espacios en Tailwind, para crear layouts con distintas columnas. Lo que has aprendido lo podrías aplicar a un sitio web completo, o al área de un componente determinado, como un navegador o una tarjeta.

Esperamos que te haya resultado de utilidad. No olvides consultar el Manual de Tailwind para seguir aprendiendo a usar este completo framework CSS.

Organización del código CSS en Tailwind CSS con la directiva @layer

Explicamos la directiva @layer de TailwindCSS, que nos sirve para definir estilos en cada una de las capas del framework y así organizar mejor el código CSS del proyecto.

La directiva @layer nos permite **agregar algunas reglas de estilos a una capa determinada del framework**, de modo que podamos escribir CSS en cualquier lugar del proyecto, asegurándonos que ese CSS se coloca en el lugar donde nosotros queremos, una vez realizado el proceso de build de los estilos.

Capas de Tailwind

Para comenzar conviene recordar cuáles eran las tres capas de TailwindCSS:

- **Base**: con código fundamental, como el típico reset.
- **Components**: con código que implementa componentes.
- **Utilities**: las potentes y numerosas clases de utilidad de Tailwind.

Estas tres capas se colocan en el orden en el que las hemos listado y sobre ellas se aplica la regla de la cascada de CSS, de modo que los estilos definidos antes son sobreescritos con cualquier código CSS que se defina después. Por tanto, los estilos definidos en "base" se sobreescriben por cualquier declaración siguiente (capas "components" y "utilities") y los estilos de "components" se sobreescriben por los que hay en "utilities".

Dada esta organización de Tailwind, el CSS inicial se compone ordenando las tres capas con las correspondientes directivas @tailwind base, @tailwind components.... Todo esto fue materia de estudio del artículo de primeros pasos con TailwindCSS.

Ejemplo de organización del código por capas

Ahora vamos a suponer que queremos agregar algo de código a los estilos de la capa "base". Para ello tendríamos que escribir el CSS del proyecto de esta manera:

```
@tailwind base;

h1 {
  @apply font-light text-2xl mb-4;
}
h2 {
  @apply font-light text-xl mb-3;
}

@tailwind components;
@tailwind utilities;
```

Nos salimos del tema pero ¿Recuerdas qué significado tiene la directiva @apply? Simplemente sirve para aplicar en el CSS estilos que vienen de las clases de TailwindCSS. Por tanto, el código anterior sería equivalente a escribir:

@tailwind base;

```
@tailwind base;

h1 {
  font-size: 1.5rem;
  font-weight: 300;
  margin-bottom: 1rem;
}
h2 {
```

```
  font-size: 1.25rem;
  font-weight: 300;
  margin-bottom: .75rem;
}

@tailwind components;
@tailwind utilities;
```

Organizar el CSS aplicando la directiva @layer

Como el código CSS que queremos asignar a los encabezados H1 y H2 debería ser incluido como estilos de base, hemos tenido que escribir estas reglas de personalización propias después de "@tailwind base" y antes de "@tailwind components". Así cualquier estilo definido posteriormente sobrescribirá a los estilos de base.

Ahora vamos a ver este mismo código haciendo uso de la **directiva @layer**, que nos permite **escribir el código CSS en cualquier lugar y asegurarnos que se incluya dentro de la capa correcta**.

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer base {
  h1 {
    @apply font-light text-2xl mb-4;
  }
  h2 {
    @apply font-light text-xl mb-3;
  }
}
```

Simplemente usamos la directiva, indicando en qué capa queremos colocar el CSS, en este caso en la layer "base". Luego abrimos llaves y colocamos todas las reglas de estilo que el framework se encargará de situar en el lugar correcto, independientemente de dónde hayas escrito el anterior CSS.

Podríamos pensar que no tiene mucha diferencia y en parte es verdad. Sin embargo hay un detalle relevante y es que gracias a esta directiva podemos dejar el código de Tailwind, con las directivas @tailwind donde sea y el código adicional lo podré colocar en cualquier archivo CSS externo, importando ese archivo en donde me apetezca, pero asegurándome que se procese en la capa donde debe.

Ahora veremos un ejemplo para ser más claros, pero en resumen podría tener un @import que define estilos en los headings, que aparezca en cualquier parte del código CSS, independientemente de dónde haya colocado las directivas @tailwind.

Organización del código CSS

Ahora vamos a ver cómo podemos organizar el código CSS de un proyecto, beneficiándonos de la directiva @layer de Tailwind CSS. Para ello, colocaremos cada bloque de código CSS en archivos independientes, dado que es la mejor manera de mantener el código sencillo, escueto y en definitiva, mantenible.

Como cada parte del código CSS del proyecto estará en un archivo aparte, tendremos que incluirlo por medio de los tradicionales @import de CSS. Podríamos todavía hacer una organización un poco más interesante por medio de varias carpetas, donde tengamos alojadas las modificaciones que hacemos en cada capa.

.

> Recuerda que para poder usar las directivas @import y que sean tratadas también con PostCSS para su procesamiento completo, necesitas usar el plugin de importado "postcss-import". El modo de uso de este plugin está explicado en nuestra página de PostCSS

Por ejemplo podríamos tener la carpeta "components" con componentes diversos del estilo de botones, tabs, etc. que usemos habitualmente. Por otra parte, podríamos tener una carpeta "base" con modificaciones base a los estilos sobre etiquetas como headings, listas, etc.

En esta arquitectura de carpetas podríamos tener un archivo raíz con todos los imports a los archivos donde tenemos separado el código por ámbitos.

```
@import './tailwind.css';
@import './components/button.css'
@import './base/headings.css';
```

Este sería el código del archivo "tailwind.css":

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Este podría ser el archivo "base/headings.css":

```
@layer base {
  h1 {
    @apply font-light text-2xl mb-4;
  }
  h2 {
    @apply font-light text-xl mb-3;
  }
}
```

Este podría ser el código del archivo "components/buttons.css":

```
@layer components {
  .button {
    @apply inline-block px-3 py-2 text-sm tracking-wider uppercase rounded;
  }

  .button-blue {
```

```
    @apply text-white bg-blue-600;
  }
}
```

Fíjate que cada archivo inyecta código en una capa distinta, los headings en la layer "base" y los botones irían en la layer "components".

Conclusión

El enfoque de Tailwind es que tu código CSS no crezca mucho, dado que principalmente se usan las clases de utilidad dentro del código HTML. Sin embargo, hay motivos más que de sobra para crear ciertas clases dentro del proyecto con elementos que vamos a usar en diversas ocasiones, así como la asignación de código de base a ciertas etiquetas que nos venga bien tener de entrada.

Con estas guías podrás organizar el código CSS de tu proyecto de una manera avanzada, pues, aunque no vayas a crear demasiados archivos, siempre es útil tener el código por separado, en diversos ficheros pequeños, en vez de uno grande con todas las modificaciones de estilos.

Aprende a usar la directiva @variant de Tailwind CSS

Cómo trabajar con la directiva @variant de Tailwind CSS. Veremos de manera práctica cómo crear nuestras propias clases a las que podremos aplicar variantes como focus, active y otras.

Tailwindcss nos permite **definir nuestras propias variantes a través de una directiva llamada @variant**, que nos resulta muy útil cuando queremos aplicar estilos CSS condicionales a un estado del elemento, como hover, focus, etc.

Recuerda que el concepto de variante y las explicaciones sobre cómo utilizarlas, incluso cómo activarlas o desactivarlas para determinadas clases en la configuración de TailwindCSS ya la vimos en uno de los primeros artículos de este manual de Tailwind CSS.

En este artículo vamos a seguir ese mismo concepto, componiendo clases que se pueden combinar en múltiples variantes, para que las podamos aplicar en momentos muy concretos.

Ahora verás que la **directiva @variant en sí es muy sencilla de utilizar**. Por eso en este artículo vamos a trabajar desde un enfoque mayormente práctico, viendo una aplicación útil de estas variantes.

Una clase CSS para producir un sacudido de un elemento

Vamos a tener para nuestro ejercicio una clase llamada "shake", que básicamente lo que hace es crear una animación que permite sacudir un elemento de la página. Es decir, es una clase que, una vez aplicada sobre cualquier elemento, produce que éste se mueva de manera repetitiva.

Esta animación, para no tenerla que crear yo mismo, me he permitido la licencia de tomarla de una librería que tiene una lista de animaciones "shake" muy divertidas. Os la recomiendo, se llama CSShake.

Esa librería incluye varias clases para animación del movimiento de sacudida, pero yo he tomado únicamente uno de ellos, que produce un efecto más suave. Básicamente el código de esta animación sería el siguiente:

```
.shake {
  transform-origin: center center;
  animation-name: shake-slow;
  animation-duration: 5s;
  animation-timing-function: ease-in-out;
  animation-iteration-count: infinite;
}

@keyframes shake-slow {
  2% {
    transform: translate(6px, -2px) rotate(3.5deg); }
  4% {
    transform: translate(5px, 8px) rotate(-0.5deg); }
  6% {
    transform: translate(6px, -3px) rotate(-2.5deg); }
  8% {
    transform: translate(4px, -2px) rotate(1.5deg); }
  10% {
    transform: translate(-6px, 8px) rotate(-1.5deg); }
  12% {
    transform: translate(-5px, 5px) rotate(1.5deg); }
  14% {
    transform: translate(4px, 10px) rotate(3.5deg); }
  16% {
    transform: translate(0px, 4px) rotate(1.5deg); }

  0%, 100% {
    transform: translate(0, 0) rotate(0); } }
```

Importante, para no aburriros con un código de animación muy largo he cortado los pasos. El código completo lo puedes obtener directamente visitando la mencionada librería CSShake o bien en el repositorio de código del proyecto Tailwind que estoy poniendo en GitHub para este manual, que estará enlazado en el final de este artículo.

> Nota: un detalle extra sería comentar que esta animación de sacudida, tal como la he incorporado yo sólo se podría realizar para elementos que no sean "inline".

Creación de las variantes

Ahora, dentro del código CSS de nuestro proyecto, vamos a crear las variantes de esta manera.

```
@variants focus, hover, active {


}
```

Dentro de las llaves colocamos la clase o clases que queramos que se generen con sus posibles variantes, en este caso focus, hover y active.

Para nuestro caso sería algo como esto:

```
@variants focus, hover, active {
  .shake {
    transform-origin: center center;
    animation-name: shake-slow;
    animation-duration: 5s;
    animation-timing-function: ease-in-out;
    animation-iteration-count: infinite;
  }
}
```

Date cuenta que la animación es la misma para todas las variantes, por lo que no es necesario que la englobes dentro de la directiva @variants.

En nuestro caso tenemos una única clase que podemos hacer con sus variantes, pero podríamos haber generado **varios tipos de clases de sacudida combinadas con cada una de las variantes** deseadas.

Ahora podríamos tener un elemento de la página que aparezca siempre en movimiento así:

```
<div class="shake">
  Me muevo!!
</div
```

Si queremos que solamente se mueva cuando el ratón esté encima del elemento podríamos usar la variante "hover".

```
<div class="p-6 bg-red-300 hover:shake">
  Me muevo!!
</div>
```

Le he puesto un poco de color a la división para que sea un poco más divertida!

Si queremos que se mueva cuando tiene el foco, es mejor que usemos un enlace.

```
<a href="#" class="button button-blue focus:shake">Vamos!</a>
```

Nota: Este botón utiliza las clases definidas en el artículo dedicado a las @layer.

Si queremos que se mueva cuando hacemos clic encima del enlace, entonces colocamos la variante "active".

```
<a href="#" class="button button-blue active:shake">Vamos!</a>
```

Las variantes se crean en el orden en el que las declaras

Como nos advierten en la documentación, en el código CSS resultante, **las variantes estarán creadas en el orden en el que las has especificado en la directiva @variants**.

Por eso, ten en cuenta que los estilos de ciertas variantes pueden sobrescribirse cuando estemos aplicando variantes distintas.

Es importante que coloques por último las que deben tomar prioridad sobre las que están antes. Es decir, que por ejemplo la variante "focus" venga después de la variante "hover".

Conclusión

Gracias a la directiva @variant hemos podido **enriquecer todavía más el framework**, creando clases de utilidad que tienen diferentes alternativas en las que pueden ser aplicados.

Esto puede ser útil en diversas situaciones, ya que la cantidad de variantes del framework es muy amplia. Por ejemplo podemos tener un estilo definido en una clase, que aplicaremos cuando sea el primer elemento de un conjunto de hermanos (variante "first") o el último (variante "last"). Los límites los pone tu imaginación.

Tailwind CSS

Tailwind CSS es un framework CSS que permite un desarrollo ágil, basado en clases de utilidad que se pueden aplicar con facilidad en el código HTML y unos flujos de desarrollo que permiten optimizar mucho el peso del código CSS.

Tailwind CSS es una potente herramienta para el desarrollo frontend. Está dentro de la clasificación de los frameworks CSS o también llamados frameworks de diseño. Permite a los desarrolladores y diseñadores aplicar estilos a los sitios web de una manera ágil y optimizada.

Tailwind permite escribir los estilos por medio de clases que se incluyen dentro del código HTML y que afectan a un aspecto muy concreto y específico de las CSS, por ejemplo, el fondo de un elemento, el color del texto o simplemente el margen por la parte de arriba. Este enfoque se conoce como "Atomic CSS", por aplicarse mediante estilos muy determinados y simples. En Tailwind CSS a estas clases se les llama "utility classes" o clases de utilidad en español.

Por tanto, Tailwind CSS no aporta muchos componentes. De hecho ofrece muy pocos componentes y son los desarrolladores los que los tienen que realizar bajo demanda del proyecto. En cambio lo que propone es entregar una enorme cantidad de clases de utilidad que combinadas en distintas variantes ofrecen prácticamente un número ilimitado de variantes de diseño, que permite una personalización del aspecto realmente única para cada proyecto.

Además Tailwind es una herramienta que se apoya en PostCSS para todo lo que es la generación del codigo CSS. Gracias a PostCSS se alcanza un flujo de desarrollo muy avanzado, personalizable, agil y sobre todo, extremadamente optimizado. Con ello, el código CSS resultante de un proyecto es realmente ajustado, porque con PostCSS se consige hacer que las clases que finalmente estén en el código de producción sean solamente las que el proyecto está usando realmente, ni una más.

Menu:

Enfoque de clases de utilidad

En Tailwind le llaman "Utility-first workflow" y básicamente consiste en el modo de trabajo que nos propone el framework, en el que tenemos que modificar el HTML para insertar las clases que sean necesarias para aplicar estilo a los componentes.

Las clases de utilidad son atómicas, es decir, afectan a un estilo muy concreto, aportando un valor determinado. Por ejemplo:

- p-8: altera el padding para que sea de 2rem (0.25 * 8 rem)
- mt-2: altera el margin-top para que sea 0.5rem (0.25 * 2 rem)
- flex: pone display a flex
- items-center: aplica el estilo "aling-items: center"
- text-orange-300: pone el color del texto a naranja más bien claro.
- bg-gray-800: pone el color de fondo gris muy oscuro.

Estas clases se puede aplicar todas a la vez, o en cualquier subconjunto de ellas, sobre un elemento, de esta manera:

```
<div class="p-8 mt-2 flex items-center text-orange-300 bg-gray-800">
    Contenido del elemento...
</div>
```

Utility-first workflow obliga a escribir muchas clases de utilidad en un elemento, cargando el HTML de numerosas clases que definen el estilo. En cierto modo quiebra la separación entre contenido y presentación que se tiene con HTML y CSS, pero tiene algunas ventajas importantes:

- Los estilos se pueden escribir sin necesidad de tocar el CSS
- El CSS de un proyecto no aumenta cuando se realizan especializaciones de estilo para algún elemento en particular.
- En las clases de utilidad podemos escribir estilos que sólo afectarán en determinadas variantes de pseudo-clases, como hover o focus

- Mediante las clases de utilidad podemos aplicar estilos que solo afectarán a un contenido en determinada anchura de pantalla.
- Las clases de utilidad se pueden personalizar, creando un tema gráfico que se acompaña durante todo el diseño.
- Tailwind CSS permite crear componentes por medio de especificación de diversas clases de utilidad, de modo que se puede reducir el uso de las mismas en los casos en los que se juzgue oportuno.
- El desarrollo se hace mucho más rápido.

Todo esto en detrimento de ensuciar el y engordar el HTML. Aunque en lo que respecta al peso, el hecho de que las páginas viajan gzippeadas, hace que ese aumento no resulte tan problemático, porque repetir textos como los nombres de clases en un documento varias veces es un patrón que se presta a una buena compresión.

Tailwind CSS vs Bootstrap

Bootstrap y otros frameworks como Materialize CSS tienen un enfoque orientado a componentes. Ofrecen clases que permiten definir componentes complejos de interfaz gráfica. Por su parte Tailwind tiene un enfoque orientado hacia clases de utilidad.

En Bootstrap podemos definir una tarjeta de esta manera:

```
<!-- BOOTSTRAP -->
<div class="card" style="width: 18rem;">
  <img src="..." class="card-img-top" alt="...">
  <div class="card-body">
    <h5 class="card-title">Título de la tarjeta</h5>
    <p class="card-text">Estamos definiendo las diferencias entre frameworks CSS</p>
    <a class="btn btn-primary" href="#">Entrar</a>
  </div>
</div>
```

Como puedes ver, para definir la tarjeta Boostrap usa clases orientadas a componentes, como "card", "card-tile", etc. Para definir un botón usa "btn" y "btn-primary".

En Tailwind CSS este mismo ejemplo lo podríamos haber definido con un HTML como este:

```
<!-- TAILWIND CSS -->
<article class="mb-5 bg-teal-100 border border-gray-200 rounded shadow-md">
<img src="..." alt="..." class="rounded rounded-b-none">
<div class="p-3">
  <h3 class="mb-1 text-xl font-bold">Los megalitos del Alentejo</h3>
  <p class="mb-4">Acercarse al megalitismo más impactante de la Península Ibérica en el Alentejo</p>
  <a class="bg-orange-600 text-white px-4 py-2 font-bold uppercase rounded-xl tracking-wider" href="#">Primary</a>
</div>
</article>
```

En este caso las diferencias son palpables. No tenemos un componente "card", sino que hemos generado una tarjeta a base de incorporar diversas clases. En vez de "card" tenemos algo como "mb-5 bg-teal-100 border border-gray-200 rounded shadow-md".

Bajo una impresión superficial podríamos pensar que resulta menos interesante, que necesitamos trabajar más y que hemos ensuciado más el HTML. En parte es cierto, pero las clases de utilidad nos ofrecen diversas ventajas, de las cuales ya hemos hablado. Pero además, en la comparativa con frameworks orientados a componentes como Bootstrap, las ventajas de Tailwind CSS son las siguientes:

- El componente de tarjeta es completamente personalizable. Simplemente aplicando alguna clase de utilidad diferente podemos cambiar el borde redondeado, para hacerlo más o menos sutil, el sombreado de la tarjeta, para realzarla más o menos, el grosor de su borde o cualquier otra cuestión.
- No necesitamos luchar contra el framework para conseguir escribir estilos que sobreescriban la tarjeta, lo que muchas veces es una tarea tediosa.
- El código CSS no crece sin control. Es decir, Bootstrap te obliga a seguir escribiendo tu propio CSS para personalización, mientras que Tailwind no necesita que escribas más CSS, porque cualquier estilo ya te lo ofrecen las clases de utilidad.

Pese a lo visto el código anterior, cabe decir que Tailwind también es amigo de la componetización. Si lo deseas puedes pasar a lo que tinenes en Bootstrap, creando clases propias que agrupan varias clases de utilidad, gracias a su directiva @apply, de modo que puedes reutilizar estilos bajo demanda.

Por último Tailwind se usa encima de herramientas de frontend sólidas como PostCSS, de modo que el enfoque es que puedas apdatar tu flujo de desarrollo con Tailwind CSS en torno de diversas herramientas de optimización. Así, el código que llevas a producción con Tailwind CSS siempre es reducido y acotado a las partes del framework que realmente estás usando.

Por último, Tailwind es un framework para quien sabe CSS, ya que para usarlo necesitas saber CSS y entender cómo funcionan sistemas como Flexbox o Grid Layout. En Bootstrap necesitas aprender el framework por ejemplo para saber usar su sistema de rejilla para crear un layout, mientras que en Tailwind CSS usas el propio CSS mediante las clases de utilidad para hacer lo mismo. No solo es que no te obligue a aprender algo nuevo, sino que te permite una mayor adaptabilidad a cualquier necesidad del proyecto.

Bibliotecas de componentes en Tailwind CSS

Podemos encontrar sitios web con bibliotecas de componentes listos para usar en Tailwind. Estos componentes vienen bien cuando queremos tener opciones rápidas para crear interfaces de usuario, solo con copiar y pegar un código HTML con sus clases aplicadas.

Recordemos que en Tailwind CSS no tienes componentes "de casa", sino que los puedes crear tú mismo. Esto puede considerarse una ventaja, porque los diseños son siempre personalizados, pero también una desventaja porque la aplicación de Tailwind a la creación de interfaces no es tan directa como en otros frameworks clásicos, basados en componentes.

DaisyUI

Es un repositorio de componentes creados por un único autor, que mantienen una buena consistencia de diseño, aparte de ser muy atractivos visualmente.

Funciona como un plugin de Tailwind que, una vez instalado permite usar todos los componentes disponibles con una cantidad de clases de CSS menores, que si tuvieras que hacerlos directamente con clases de utilidad de Tailwind CSS.

Además son altamente personalizables vía clases de utilidad y tiene una gestión de tema gráfico, mediante el cual puedes incorporar estilo a tus componentes de manera global y consistente en todo un proyecto.

Desde luego, si estás buscando componentes de TailWind CSS es una de las mejores opciones, ya que encima su uso es completamente gratuito. Más información en la página de DaisyUI

Tailwind UI

El sitio más conocido para encontrar componentes de Tailwind es Tailwind UI. Este sitio es oficial de Tailwind, creado por los mismos desarrolladores del framework. Sin embargo, es un complemento de pago.

Este sitio no se limita solo a crear componentes, sino que también te ofrece layouts completos de aplicaciones creados por completo con Tailwind.

Tailwind Components

TailwindComponents es un sitio está muy bien para encontrar componentes creados por la comunidad en Tailwind. Todos los componentes son gratuitos, creados por los mismos usuarios del sitio. Usarlos es solo copiar y pegar el HTML proporcionado. Hay componentes de todo tipo, para hacer interfaces simples de todo tipo, como botones, navegadores, barras de acciones, campos de textos, etc.

Ofrece también una sección para obtener layouts completos de sitios web con un diseño variado y aplicable a todo tipo de webs.

POSTCSS

Todo un set de herramientas y plugins para transformaciones del CSS mediante Javascript útil para proyectos frontend.

PostCSS es una herramienta moderna para la gestión del código CSS que permite aumentar la productividad a la vez que libera al desarrollador de trabajos adicionales así como conocimientos técnicos del estado actual del lenguaje y el soporte en los navegadores.

PostCSS por si solo no hace trabajo alguno sobre el CSS. En cambio es necesario instalar plugins que nos permitan indicar qué tipo de transformaciones serán realizadas sobre el CSS que tenemos de entrada, generando un CSS de salida acorde con las necesidades de cada proyecto o las tecnologías con las que se trabaje.

Existen más de 200 plugins de PostCSS para realizar todo tipo de transformaciones, que se pueden encontrar en la página de GitHub de PostCSS.

PostCSS se puede integrar con herramientas frontend existentes en un proyecto dado, como pueden ser Webpack o Parcel, o también se puede usar directamente por medio de su propia CLI.

Menu

Instalar PostCSS

PostCSS se instala en el proyecto donde lo desees utilizar. Es una herramienta que funciona bajo NodeJS, por lo que previamente tendrás que haber instalado esa plataforma en tu equipo. Por cierto, PostCSS también es compabible con Deno.

Lo puedes instalar como un añadido a cualquier sistema de build que vengas utilizando, como podría ser Webpack, aununque también lo puedes usar de manera autónoma gracias a su CLI. Ahora vamos a explicar cómo instalar PostCSS con PostCSS-CLI.

Lo instalas vía npm, por lo que necesitas haber inicializado el proyecto con npm:

```
npm init
```

Vas contestando el asistente de inicialización y luego tienes que instalar dos dependencias, como dependencias de desarrollo, en el proyecto:

```
npm install -D postcss postcss-cli
```

Además, necesitas instalar todos los plugins que consideres necesarios de PostCSS para tu proyecto. Una alternativa muy popular es "autoprefixer", que se encarga de procesar tu CSS y colocar los prefijos en los atributos CSS que sean necesarios.

```
npm install -D autoprefixer
```

Este paso de instalación lo debes completar luego con la configuración de PostCSS, en la que indicarás qué plugins se deben usar.

Configuración de PostCSS

Una vez tienes instaladas todas las dependencias, incluido el número de plugins que veas conveniente, debes crear un archivo de configuración de PostCSS, llamado "postcss.config.js". Ese archivo contendrá el código Javascript que NodeJS debe utilizar para configurar la ejecución de PostCSS. En él básicamente indicarás los plugins que se van a usar y su configuración.

Un ejemplo sencillo de configuración de PostCSS en postcss.config.js sería el siguiente:

```
module.exports = {
  plugins: [
    require('autoprefixer'),
  ]
}
```

Como puedes apreciar, PostCSS se ejecuta con NodeJS, por eso usamos la sentencia "module.exports", que es la manera de exponer un módulo de Javascript hacia afuera que se usa en NodeJS, que se llama "CommonJS"

Uso de PostCSS

Para comenzar a usar PostCSS nos queda un último paso, que es crear un script de npm para ejecutar PostCSS cada vez que necesites procesar tu CSS. Ese script de npm lo tienes que colocar dentro del archivo "package.json".

El archivo "package.json", tiene una propiedad llamada "scripts", en la que puedes colocar cualquier número de scripts npm que puedas necesitar en un proyecto. Su forma sería como esta:

```
"scripts": {
    "build": "postcss src/css/estilos.css --output dist/estilos.css"
},
```

En nuestro caso hemos creado un script llamado "build", que hace la invocación al CLI de PostCSS, indicando dos informaciones:

- **src/css/estilos.css sería la ruta inicial donde se encuentra el archivo CSS que se desea procesar.**
- **--output dist/estilos.css sería la ruta de destino donde se colocará el CSS una vez procesado.**

Una vez creado el script npm, lo ejecutas con la consola, desde la raíz de tu proyecto con el comando "npm run" seguido del nombre del script que hayas puesto.

```
npm run build
```

Esto realizará el procesamiento del archivo de origen, aplicando todas las transformaciones de todos los plugins que hayas configurado, y escribiendo el archivo de destino en la ruta indicada.

Por ejemplo, si tuvieras un CSS como este:

```
myGrid {
  display: grid;
  grid-template-columns: 12px 12px 12px;
  grid-template-rows: 12px 12px 12px;
  column-gap: 1rem;
}
```

Lo que tendríamos como salida es esto:

```
._myGrid {
  display: grid;
  grid-template-columns: 12px 12px 12px;
  grid-template-rows: 12px 12px 12px;
  -moz-column-gap: 1rem;
       column-gap: 1rem;
}
```

Sin embargo ten en cuenta que esta salida puede ser variable, ya que autoprefixer tiene en cuenta un rango de navegadores de destino y, a medida que los navegadores se actualizan hay ciertos prefijos que ya no son necesarios.

Si deseas cambiar los navegadores de destino puedes hacerlo de diversas maneras, por ejemplo creando un archivo ". browserslistrc" en la raíz de tu proyecto. Mira la documentación para encontrar más detalles.

Lanzar el proceso de build de manera automática

En tiempo de desarrollo realizamos cambios en el código CSS con mucha frecuencia. Como nuestro código CSS es generado, para ver los cambios en el CSS necesitaríamos lanzar el proceso de build, con el comando que ejecuta el correspondiente script npm.

Resulta muy incómodo tener que lanzar el proceso de build con cada cambio que hagamos para poder ver el nuevo aspecto que tiene nuestra página en el navegador, así que lo ideal es crear un sistema de "watch" que esté pendiende de los cambios en los archivos de origen, para compilarlos de nuevo automáticamente cada vez que se modifican.

Esto se puede configurar de una manera muy sencilla, con el flag --watch en la llamada a PostCSS. Lo ideal es tener un script npm para el build y otro para el watch, lo que nos quedaría más o menos así.

```
"scripts": {
  "build": "postcss src/css/styles.css --output dist/css/styles.css",
  "watch": "postcss src/css/styles.css --output dist/css/styles.css --watch"
},
```

Ahora, para lanzar el proceso de watch y realizar cambios automáticamente en el CSS generado ejecutaremos el comando siguiente:

```
npm run watch
```

Plugin de importado de PostCSS

Una de las utilidades más interesantes de PostCSS es la capacidad de hacer imports de distintos archivos de CSS dentro del CSS principal del proyecto. De este modo podemos desarrollar el código CSS de un proyecto en varios archivos pequeños, en lugar de una grande y difícil de mantener.

Para hacer el importado de otros archivos CSS se usa el plugin postcss-import. Se debe de instalar así:

```
npm install -D postcss-import
```

Una vez instalado lo podemos incluir en nuestro archivo de configuración postcss.config.js.

```
module.exports = {
  plugins: [
    require('postcss-import'),
    // otros plugins...
  ]
}
```

Generalmente lo querrás incluir como primer elemento, para construir un gran archivo CSS con todo el código junto, antes de pasarle las siguientes transformaciones.

Los imports en CSS se realizan con la directiva @import, indicando la ruta donde esta el archivo que deseas importar. Preprocesadores como Sass usan la misma técnica. El código CSS sería más o menos como este:

```
@import "./componentes/navegador.css";
```

A la hora de importar otros archivos de código CSS necesitas tener en cuenta:

- Los imports se deben realizar siempre al principio de los archivos
- Para organizar tus imports es muy habitual hacer un archivo raíz en el que no tengas CSS, solo los imports de otros archivos.
- Puedes importar cosas instaladas en node-modules, simplemente usando los nombres de los packages y la ruta del archivo dentro de ese package.

Plugin PurgeCSS

El plugin PurgeCSS permite eliminar de un archivo de código CSS todas las reglas de estilo que no se están utilizando. Al ponerse en marcha este plugin de PostCSS se hace un análisis del código CSS y posteriormente el código de la página o aplicación web, realizando transformaciones en el código CSS para eliminar todas las reglas de selectores que no se están usando en el código.

Es un plugin esencial cuando se trabaja con frameworks CSS como Bootstrap o Tailwind, puesto que estos frameworks incluyen mucho CSS que en realidad no se llega a usar en los proyectos. Gracias a este plugin podemos asegurar que el CSS que se lleva a producción incluye única y exclusivamente código CSS que se usa en el sitio web.

Se instala de la siguiente manera:

```
npm i -D @fullhuman/postcss-purgecss
```

Luego tienes que configurar el plugin dentro del archivo postcss.config.js, indicando con una expresión regular los archivos donde se encuentra el código HTML de tu proyecto, es decir todos los ficheros donde puede haber código HTML, para que lo analice en función del código CSS del proyecto, para saber todos los selectores que verdaderamente importan para el sitio web.

En un sitio web sencillo podríamos tener solamente páginas HTML en una carpeta, pero podrías tener templates en archivos Javascript o código HTML en archivos .php, por ejemplo. Para indicar todos los tipos de archivos y carpetas donde buscar se indican distintos patrones de rutas en el array "content".

El código para usar PurgeCSS nos quedaría así:

```
const purgecss = require('@fullhuman/postcss-purgecss');

module.exports = {
  plugins: [
    purgecss({
      content: [
      './public/**/*.html',
```

```
      './public/**/*.php',
      './public/js/**/*.js',
    ]}),
  ]
}
```

Una vez modificado el archivo de configuración de PostCSS lanzamos el proceso de ejecución de PostCSS de manera habitual y observaremos que el código CSS generado habrá sido reducido a lo esencial. Por supuesto, dependiendo de si usamos frameworks o no puede que la diferencia de peso sea más o menos notoria.

Plugin de PostCSS para minimizar el CSS

Gracias a PostCSS podemos conseguir realizar cómodamente otra de las tareas que nos ayudarán a reducir el peso del CSS generado. Consiste en pasarle un minimizador de CSS, que compacte el código, elimine comentarios y cosas como esas.

Esto lo podemos conseguir mediante un plugin de PostCSS llamado "cssnano". Para usarlo tienes que comenzar instalando el plugin en el proyecto con npm.

```
npm i -D cssnano
```

Luego lo configuramos como un plugin más dentro del archivo postcss.config.js.

```
const purgecss = require('@fullhuman/postcss-purgecss');

module.exports = {
  plugins: [
    require('tailwindcss'),
    require('autoprefixer'),
    purgecss({
      content: [
      './dist/**/*.html'
    ]}),
    require('cssnano')({
      preset: 'default',
    }),
  ]
}
```

En este caso estamos usando el preset de cssnamo llamado "default", que será el ideal en la mayoría de los casos.

Las reducciones obtenidas por medio de la minimificación pueden ser relevantes, a veces mayores del 50% del peso del archivo, aunque depende lógicamente cómo se ha codificado el CSS original.

Plugin para anidación de reglas CSS

En PostCSS tenemos la posibilidad de usar dos plugin distintos para anidación de reglas de CSS. Eligiremos uno u otro en función de la sintaxis que queramos usar.

- PostCSS Nesting: Este soporta la sintaxis de la especificación de CSS Nesting
- PostCSS Nested: En este caso soporta la sintaxis de anidación de CSS implementada en Sass

PostCSS Nested

Por ejemplo, una vez instalado PostCSS en el proyecto, para usar PostCSS Nested realizaremos el siguiente comando de instalación:

```
npm install -D postcss-nested
```

Configuramos el postcss.config.js con el siguiente código:

```
module.exports = {
    plugins: [
      require('postcss-nested'),
    ]
}
```

A continuación podremos hacer anidación de reglas CSS, en este caso con la sintaxis de Sass. Sirva este código de muestra:

```
h1 {
    color: red;

    &.main {
        color: green;
        font-size: 2.5rem;
    }
}
```

PostCSS Nesting

Si preferimos usar la especificación de CSS Nesting podemos usar el plugin PostCSS Nesting, que instalamos con el comando:

```
npm install -D postcss-nesting
```

Luego se incluye el plugin en la configuración de PostCSS dentro del archivo postcss.config.js.

```
module.exports = {
    plugins: [
      require('postcss-nesting'),
    ]
}
```

Referencias

https://desarrolloweb.com/manuales/manual-de-tailwindcss

https://desarrolloweb.com/home/tailwind-css

https://desarrolloweb.com/home/postcss

Toda la documentación esta en este archivo XD 😎😎😎😎

Creado por el de programador Ariel Zarate https://www.linkedin.com/in/ariel-zarate/