Unidad 4: Grids

4.1.Introducción a Grids

Propiedades Básicas de CSS Grid Aplicadas al Contenedor (Padre)

En este módulo, exploraremos las propiedades fundamentales de CSS Grid que se aplican al contenedor (padre). Estas propiedades permiten definir la estructura y disposición de los elementos hijos dentro de una grilla. A continuación, se detallan cada una de estas propiedades junto con ejemplos de código CSS y su resultado visual.

display: grid

La propiedad `display: grid` convierte un elemento en un contenedor de grilla, lo que permite organizar sus elementos hijos en filas y columnas.

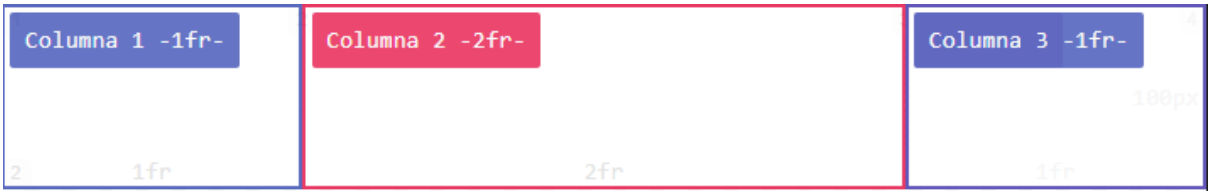
```
.container {
    display: grid;
}
```

grid-template-columns

La propiedad **`grid-template-columns`** define la estructura de las columnas en la grilla. Puedes especificar el tamaño de cada columna usando unidades como **`px`**, **`%`**, **`fr`**, entre otras.

```
.container {
    display: grid;
     grid-template-columns: repeat(3, 1fr);
}
```

Ejemplo Visual



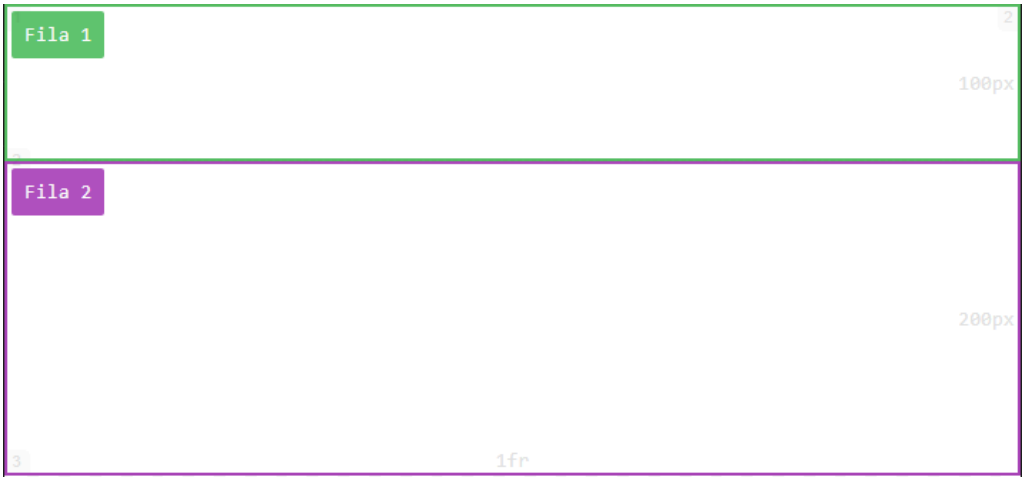
Nota: Para que puedas visualizar la grilla de manera más clara en tu proyecto, recordá declarar la propiedad border en cada área o contenedor. Así podrás asignarle un grosor y un color (por ejemplo, border: 1px solid red;) y distinguir fácilmente los límites de cada sección.

grid-template-rows

La propiedad **`grid-template-rows`** define la estructura de las filas en la grilla, similar a cómo **`grid-template-columns`** define las columnas.

```
.container {
    display: grid;
    grid-template-rows: 100px 200px;
}
```

Ejemplo Visual

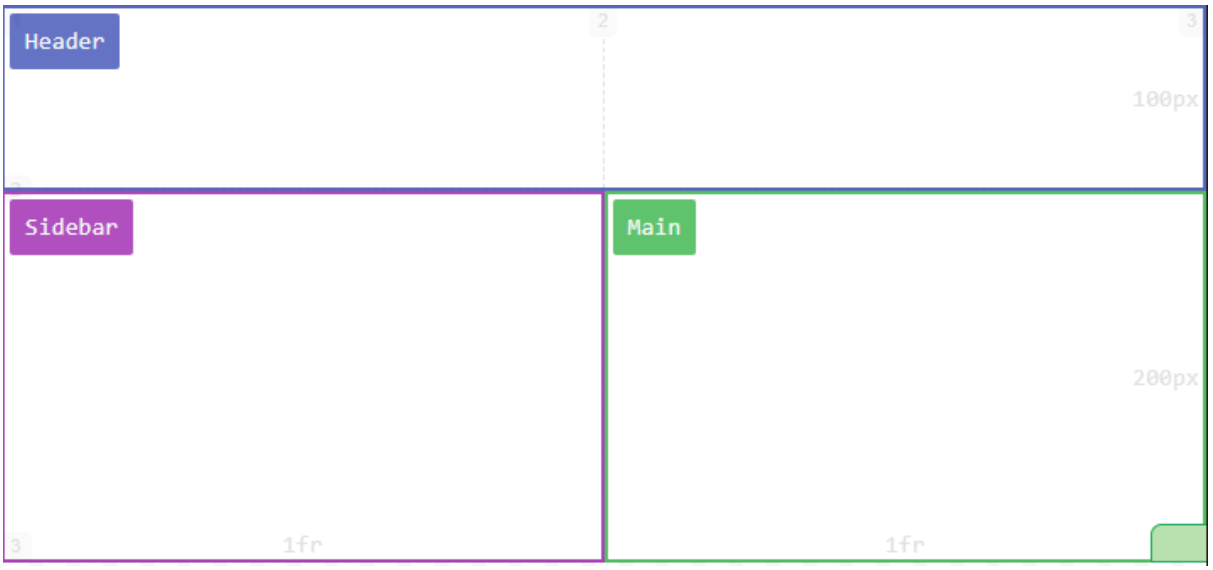


grid-template-areas

La propiedad **`grid-template-areas`** permite nombrar áreas dentro de la grilla para facilitar el posicionamiento de los elementos hijos. Se usan nombres de áreas definidos en los elementos hijos.

```
.container {
    display: grid;
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 100px 200px
    grid-template-areas:
        "header header"
        "sidebar main";
}
.header {
    grid-area: header;
}
.sidebar {
    grid-area: sidebar;
}
.main {
    grid-area: main;
}
```

Ejemplo Visual

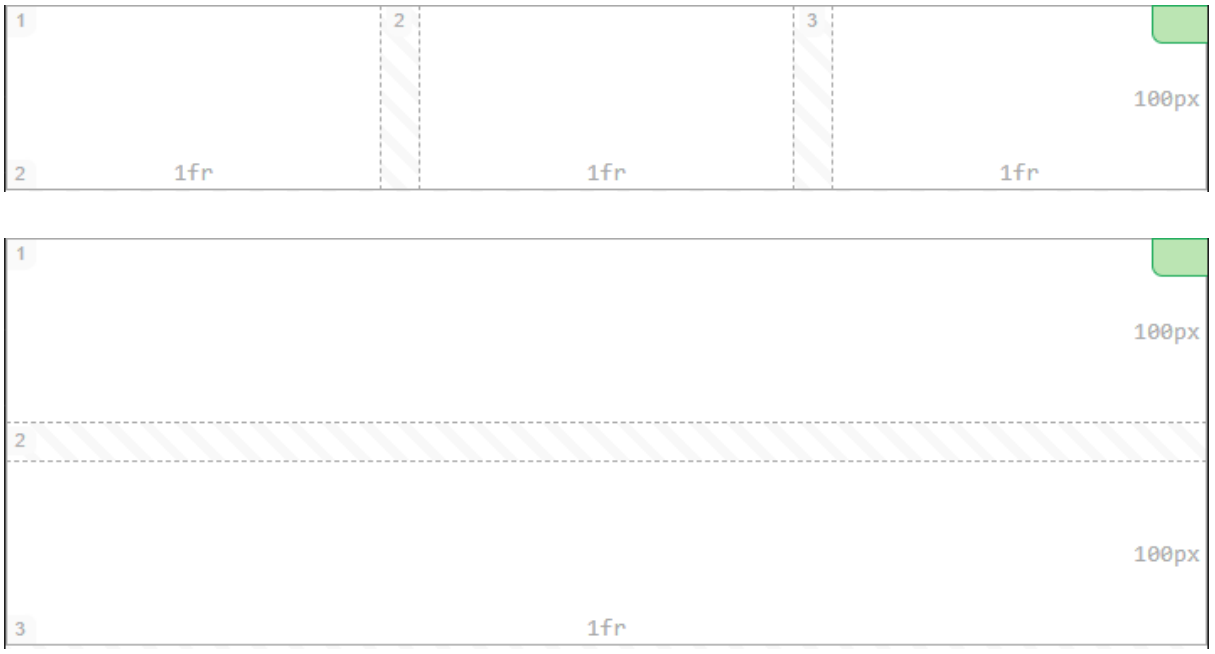


🧱 Espaciado entre filas y columnas: gap

La propiedad gap permite definir **de forma simple y unificada** el espacio entre las filas y las columnas de una grilla.
Es la forma actual recomendada por CSS (totalmente soportada en todos los navegadores modernos) y reemplaza el uso separado de row-gap y column-gap.

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 20px 10px; /* columna / fila */
}
```

Ejemplo Visual



👉 En este ejemplo:

- 20px corresponde al **espacio entre columnas**.
- 10px corresponde al **espacio entre filas**.
- Si se declara un solo valor (por ejemplo, gap: 10px;), se aplica el mismo espaciado tanto en filas como en columnas.

💡 Usar gap simplifica la escritura y evita repetir propiedades.

📜 Propiedades anteriores: `column-gap` y `row-gap`

Antes de que `gap` fuera ampliamente soportado, se usaban las propiedades específicas:

```
/* Sintaxis anterior (histórica) */
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  column-gap: 20px; /* antes llamada grid-column-gap */
  row-gap: 10px;    /* antes llamada grid-row-gap */
}
```

Estas siguen funcionando por compatibilidad, pero **ya no se recomiendan como la forma principal de enseñar ni de construir layouts modernos**.

Podés mencionarlas en clase como "lo que se usaba antes" para que los alumnos reconozcan el código si se lo cruzan en proyectos antiguos, pero **todo el material y ejemplos deben centrarse en `gap`**.

Ejemplo Completo

A continuación, se presenta un ejemplo completo que combina varias de estas propiedades para ilustrar cómo se pueden usar juntas para crear una grilla compleja.

HTML

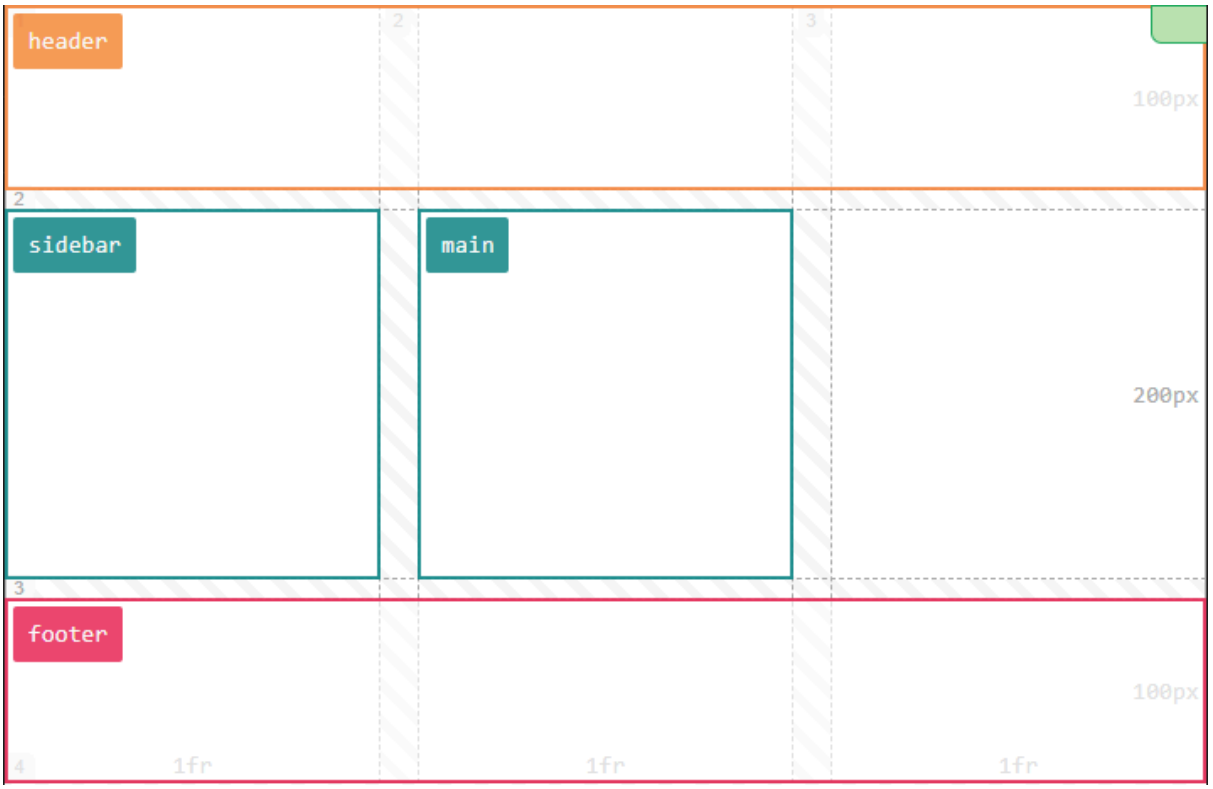
```
<div class="container">
  <header class="header">Header</header>
  <main class="main">Main</main>
  <aside class="sidebar">Sidebar</aside>
  <footer class="footer">Footer</footer>
</div>
```

CSS

```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;
  grid-template-rows: 100px 200px 100px;
  grid-template-areas:
```

```
    "header header header"
    "sidebar main ."
    "footer footer footer";
  gap: 10px 20px;
}

.header {
    grid-area: header;
}
.sidebar {
    grid-area: sidebar;
}
.main {
    grid-area: main;

}
.footer {
    grid-area: footer;
}
```

Ejemplo Visual Completo



Explicación:

- En este ejemplo, tenemos un contenedor de grilla con tres columnas y tres filas.
- Las áreas se definen con nombres como "header", "sidebar", “main” “footer” y "." (un punto vacío).
- Se aplica un espacio de 20px entre columnas y 10px entre filas.
- Cada elemento hijo (header, sidebar, main, footer) se posiciona en el área correspondiente usando la propiedad grid-area.

💡 **Recomendación:** Aplicá colores de fondo o bordes diferentes a cada área para visualizar la estructura de la grilla.

4.2.Ejemplos Prácticos de Propiedades de Grids

Items y Áreas de la Grilla

Flexibilización de Ítems Individuales en CSS Grid

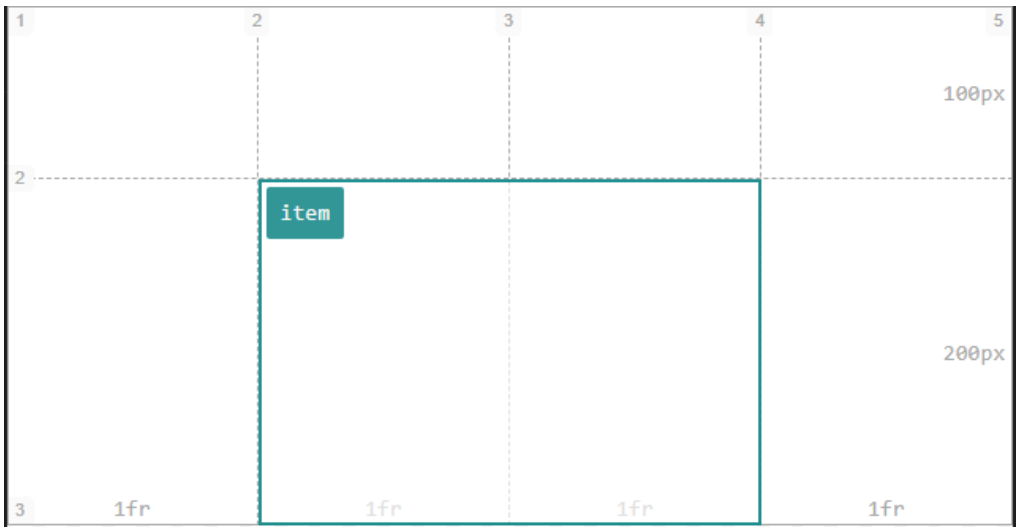
En este módulo, aprenderemos cómo flexibilizar y posicionar ítems individuales dentro de una grilla de CSS Grid. Utilizaremos propiedades como `grid-column`, `grid-row`, y `grid-area` para ajustar la posición y el tamaño de elementos específicos. A continuación, se explica cada propiedad junto con ejemplos de código CSS y su resultado visual.

grid-column

La propiedad **`grid-column`** se utiliza para definir en qué columna comienza un ítem y cuántas columnas ocupa. Esta propiedad combina **`grid-column-start`** y **`grid-column-end`**.

```
.item {
    grid-column: 2 / 4;
}
```

Ejemplo Visual



En este ejemplo tu grilla tiene 3 columnas,al aplicar `grid-column: 2 / 4;` el ítem se extiende **desde la columna 2 hasta antes de la 4**, es decir, **ocupa las columnas 2 y 3**. El segundo número es exclusivo.

Recordatorio
En este punto ya contamos con una **grilla previamente creada** en el contenedor. La propiedad `grid-column` no construye una grilla nueva, sino que se aplica a los ítems que forman parte de esa grilla. Es decir, lo que estamos definiendo es cómo se ubica un elemento dentro de la estructura ya existente.

grid-row

La propiedad **`grid-row`** se utiliza para definir en qué fila comienza un ítem y cuántas filas ocupa. Esta propiedad combina **`grid-row-start`** y **`grid-row-end`**.

```
.item {
    grid-row: 1 / 3;
}
```

Ejemplo Visual

En este ejemplo si tu contenedor tiene varias filas definidas, al usar `grid-row: 1 / 3;` le estás diciendo al ítem que comience en la fila 1 y se extienda hasta la fila 2, ocupando ese espacio en la grilla creada previamente.

Recordatorio
Al igual que con las columnas, es importante aclarar que la propiedad `grid-row` no crea una grilla nueva. Esta propiedad se aplica sobre un **ítem que ya forma parte de una grilla existente**, y lo que hace es indicar **desde qué fila hasta qué fila debe extenderse** dentro de esa estructura.

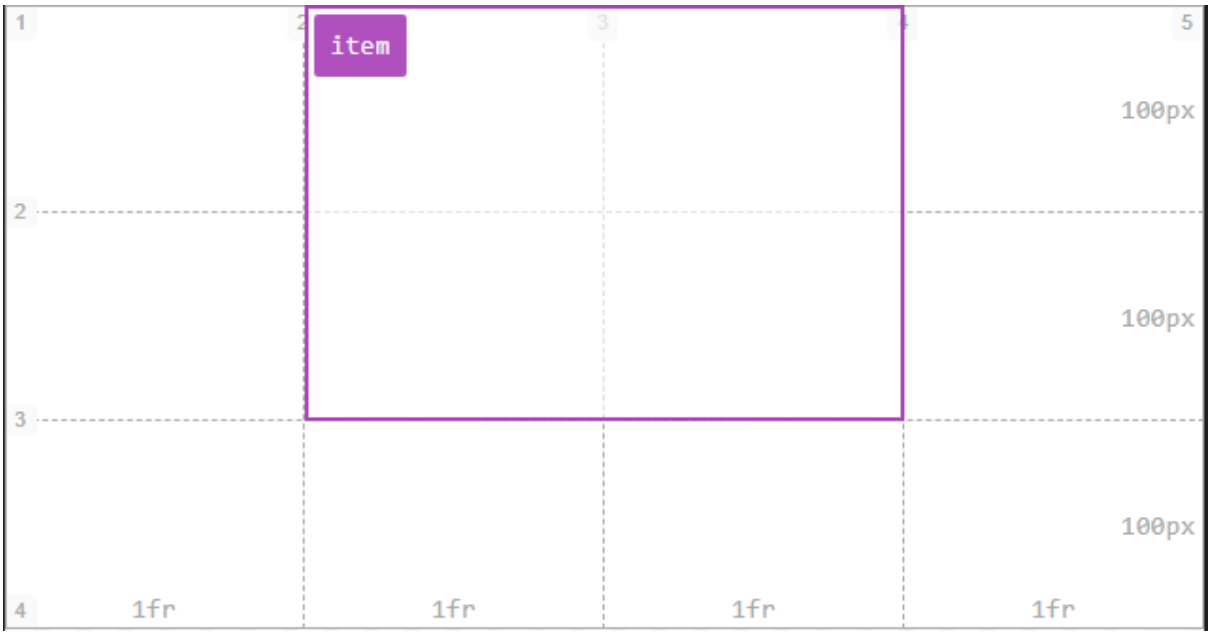
grid-area

La propiedad **`grid-area`** es una forma abreviada de definir **`grid-row-start`**, **`grid-column-start`**, **`grid-row-end`** y **`grid-column-end`** en una sola propiedad.

```
.item {
    grid-area: 1 / 2 / 3 / 4;
}
```

👉 Ocupa desde la fila 1 hasta antes de la 3 (filas 1 y 2), y desde la columna 2 hasta antes de la 4 (columnas 2 y 3).

Ejemplo Visual



El `item` ocupa **desde la fila 1 hasta la fila 2** (porque termina en la 3) y **desde la columna 2 hasta la 3** (porque termina en la 4).

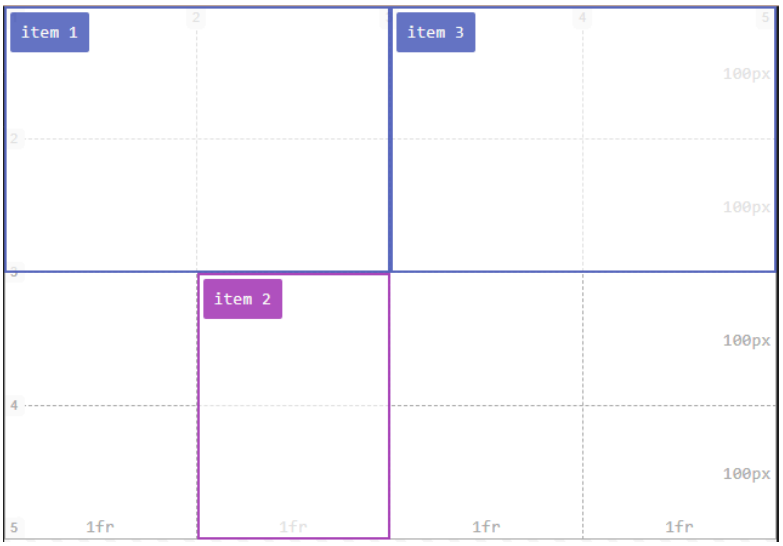
Recordatorio
Cuando usamos `grid-area` no estamos creando una grilla nueva, sino **ubicando un elemento dentro de una grilla que ya existe**. Lo que hace esta propiedad es combinar en una sola línea las instrucciones de **fila inicial, columna inicial, fila final y columna final**.

Ejemplo Completo

A continuación, se presenta un ejemplo completo que combina varias de estas propiedades para ilustrar cómo se pueden usar juntas para posicionar elementos específicos dentro de una grilla.

```
.container {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    grid-template-rows: repeat(3, 100px);
    gap: 10px;
}
.item1 {
    grid-column: 1 / 3;
}
.item2 {
    grid-row: 2 / 4;
}
.item3 {
    grid-area: 1 / 3 / 3 / 5;
}
```

Ejemplo Visual Completo



Nota: para que se te visualice la grilla con bordes de colores deberás declarar la propiedad border con el grosor y el color de tu preferencia.

En este ejemplo:

- **`Item 1`** ocupa las columnas 1 y 2 de la primera fila.
- **`Item 2`** ocupa las filas 2 y 3 de la segunda columna.
- **`Item 3`** ocupa desde la fila 1, columna 3 hasta la fila 2, columna 4.

Los colores son parte del ejemplo pero no se visualizar de esa forma a no ser que le declares

Estos conceptos y ejemplos proporcionan una base sólida para comenzar a trabajar con CSS Grid y flexibilizar ítems individuales dentro de una grilla.

Áreas de la grilla

⚙️Actividad Práctica

Galería con CSS Grid

Crear una galería de imágenes organizada en una grilla de 3 filas x 3 columnas, aplicando las propiedades grid-column, grid-row y grid-area para ubicar y flexibilizar ítems individuales dentro de la estructura.

Consignas Paso a Paso

1) **Estructura HTML básica**

- Crea una sección en tu index.html o en cualquier otro documento con un contenedor <section class="gallery">.

- Dentro de ese contenedor, agrega 9 elementos (pueden ser imágenes <img> o bloques de color con texto de ejemplo: “Item 1”, “Item 2”… “Item 9”).

```
<section class="gallery">
   <div class="item item1">1</div>
   <div class="item item2">2</div>
   <div class="item item3">3</div>
   <div class="item item4">4</div>
   <div class="item item5">5</div>
   <div class="item item6">6</div>
   <div class="item item7">7</div>
   <div class="item item8">8</div>
   <div class="item item9">9</div>
 </section>
```

2. **Definir la grilla en CSS**

    - Aplica **CSS Grid** a la galería:

```
.gallery {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(3, 150px);
```

```
  gap: 10px;
}

.item {
  border: 2px solid black;
  display: flex;
  align-items: center;
  justify-content: center;
  font-size: 20px;
  background-color: lightgray;
}
```

3. **Aplicar propiedades de posicionamiento**
    - Usa **grid-column**, **grid-row** y **grid-area** para personalizar la ubicación de algunos ítems:
        - `item1`: que ocupe las **columnas 1 y 2** de la primera fila.
        - `item5`: que ocupe las **filas 2 y 3** en la segunda columna.
        - `item9`: que ocupe desde la **fila 2, columna 3 hasta la fila 3, columna 3** (o sea, dos filas de la última columna).
4. **Visualización**
    - Para que la grilla se entienda mejor, asigna colores de fondo distintos a cada item o usá imágenes diferentes.
    - Usa la propiedad `border` para ver los límites de cada celda.

💡 **Consejo:** Probá mover distintos ítems usando grid-column, grid-row o grid-area para entender cómo cambia la distribución.

4.3.Propiedades de Distribución en CSS Grid

En este módulo, exploraremos cuatro propiedades fundamentales de **CSS Grid** que controlan la distribución de los elementos en la grilla: `justify-items`, `align-items`, `justify-content` y `align-content`. Estas propiedades nos permiten alinear y distribuir los ítems y las áreas de la grilla de manera precisa. A continuación, te presentamos una explicación actualizada de cada propiedad junto con ejemplos claros y visuales que ilustran sus efectos.

`justify-items`

La propiedad `justify-items` alinea los elementos hijos dentro de sus celdas a lo largo del eje horizontal (izquierda a derecha). Los valores posibles son:

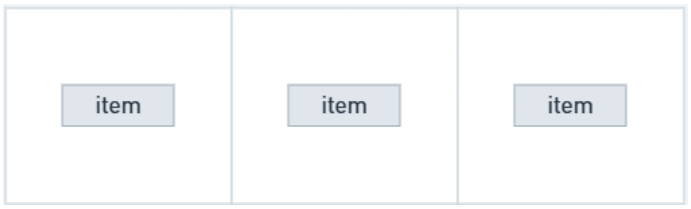
- `start` (inicio)
- `end` (fin)
- `center` (centro)
- `stretch` (ocupa todo el ancho disponible)

Ejemplo de Código

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    justify-items: center;
}
```

Visualización Actualizada

Cada ítem está centrado horizontalmente en su celda:



`align-items`

La propiedad `align-items` alinea los elementos hijos dentro de sus celdas a lo largo del eje vertical (arriba a abajo). Los valores posibles son:

- `start` (arriba)
- `end` (abajo)
- `center` (centro)
- `stretch` (ocupa toda la altura disponible)

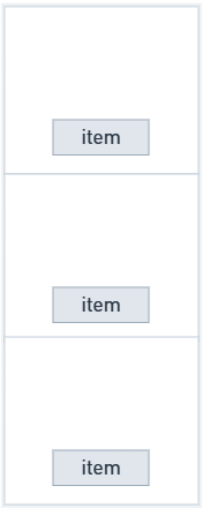
Ejemplo de Código

```
.container {
```

```
    display: grid;
    grid-template-rows: repeat(3, 100px);
    align-items: end;
}
```

Visualización Actualizada

Cada ítem está alineado al fondo de su celda:



`justify-content`

La propiedad `justify-content` alinea y distribuye las columnas de la grilla a lo largo del eje horizontal del contenedor. Los valores posibles son:

- `start` (inicio)
- `end` (fin)
- `center` (centro)
- `space-between` (espacio entre columnas)
- `space-around` (espacio alrededor de columnas)
- `space-evenly` (espacios iguales entre y alrededor de columnas)

Ejemplo de Código

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    justify-content: space-between;
}
```

Visualización Actualizada

Las columnas están distribuidas con espacio entre ellas:

`align-content`

La propiedad `align-content` alinea y distribuye las filas de la grilla a lo largo del eje vertical del contenedor. Solo tiene efecto cuando el contenedor de la grilla tiene un espacio mayor al necesario para las filas. Los valores posibles son:

- `start` (arriba)
- `end` (abajo)
- `center` (centro)
- `space-between` (espacio entre filas)
- `space-around` (espacio alrededor de filas)
- `space-evenly` (espacios iguales entre y alrededor de filas)

Ejemplo de Código

```
.container {
    display: grid;
    grid-template-rows: repeat(3, 100px);
    align-content: space-evenly;
}
```

Visualización Actualizada

Las filas están distribuidas con espacios iguales entre y alrededor:

Ejemplo Completo

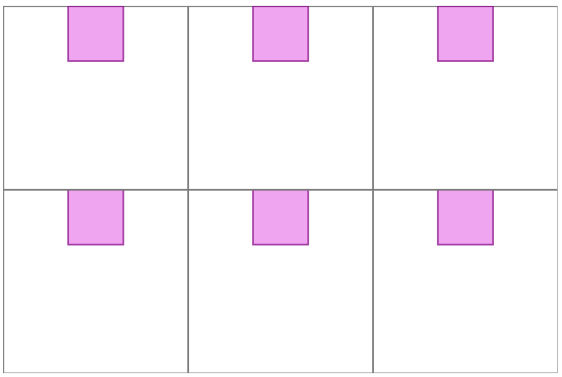
Combinemos varias propiedades para mostrar cómo afectan la disposición de la grilla.

Ejemplo de Código

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows: repeat(2, 100px);
    justify-items: center;
    align-items: start;
    justify-content: space-around;
    align-content: space-between;
}
```

Visualización Actualizada

El resultado es una grilla organizada de la siguiente forma:



En este ejemplo:

- **`justify-items: center`** alinea los ítems horizontalmente al centro de sus celdas.
- **`align-items: start`** alinea los ítems verticalmente al inicio de sus celdas.
- **`justify-content: space-around`** distribuye las columnas con espacio alrededor.
- **`align-content: space-between`** distribuye las filas con espacio entre ellas.

4.4.Introducción al Diseño Responsive y su Importancia en el Desarrollo Web

El diseño responsive es un enfoque en el desarrollo web que garantiza que los sitios web se vean y funcionen bien en una amplia variedad de dispositivos y tamaños de pantalla, desde teléfonos móviles hasta computadoras de escritorio. La importancia del diseño responsive radica en la creciente diversidad de dispositivos utilizados para acceder a la web, lo que hace esencial que los sitios sean accesibles y fáciles de usar en cualquier dispositivo.

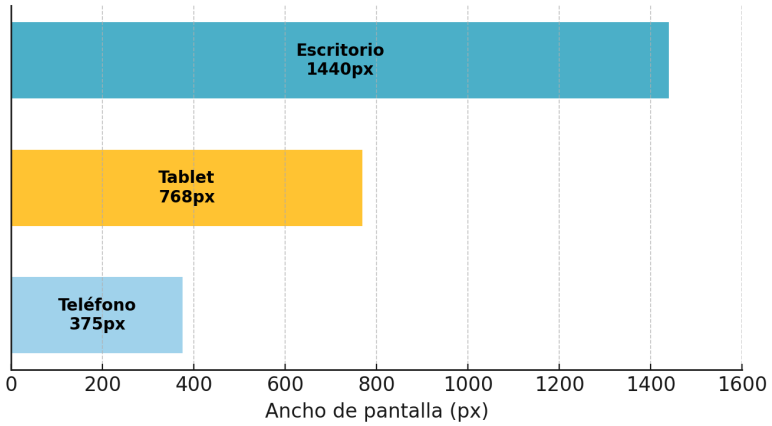
Importancia del Diseño Responsive

- **Mejora la Experiencia del Usuario**: Un sitio web que se adapta a diferentes tamaños de pantalla proporciona una mejor experiencia de usuario, lo que puede aumentar el tiempo de permanencia y la satisfacción del usuario.
- **Optimización para Motores de Búsqueda**: Google y otros motores de búsqueda favorecen los sitios web que son móviles y responsivos, lo que puede mejorar el ranking en los resultados de búsqueda.
- **Mayor Alcance**: Un diseño responsive garantiza que el sitio sea accesible para una mayor audiencia, incluidos los usuarios de dispositivos móviles, tabletas y computadoras de escritorio.
- **Reducción de Costos de Desarrollo**: En lugar de crear y mantener múltiples versiones de un sitio para diferentes dispositivos, un diseño responsive permite gestionar un solo sitio que se adapta a todos los dispositivos.

Uso de Media Queries en CSS

Las *media queries* son una característica de CSS que permiten aplicar estilos específicos en función de las características del dispositivo, como el ancho y la altura de la pantalla. A continuación, se presentan ejemplos de cómo utilizar *media queries* para adaptar el diseño a diferentes tamaños de pantalla.

Si acaso te estás preguntando cuáles son las medidas “exactas” que tenés que usar para el responsive, la respuesta es: **no existen medidas únicas ni definitivas**. Lo que tenemos son **estándares de referencia** (ej. 375px para teléfonos, 768px para tablets, 1024px/1440px para escritorios), pero siempre pueden variar según tu proyecto y los dispositivos en los que quieras que se visualice. De igual forma, a medida que vayas practicando vas a notar que si trabajas bien flex, grid, medidas relativas, etc., el diseño se va a adaptar de una forma excelente.



Estos valores son de referencia y te pueden ayudar a entender los breakpoints más comunes en media queries.

Sintaxis

1. Media query simple (ancho máximo)

Cuando queremos que algo cambie **solo en pantallas más pequeñas a la que se está desarrollando** (por ejemplo: tablets o celulares), usamos `max-width`. Útil si tu diseño arranca en dispositivos grandes de escritorio y necesitas realizar las adaptaciones a dispositivos más pequeño.

```
@media (max-width: 600px) {
  body {
    background-color: lightblue;
  }
}
```

Explicación: En este caso, si la pantalla mide 600px o menos, el fondo será azul. Es la forma más directa de decir *“aplicá este estilo cuando la pantalla sea chica”*.

2. Media query con ancho mínimo

Si en cambio queremos que un estilo se aplique **a partir de cierto tamaño hacia arriba** (ejemplo, tablets y desktops), usamos `min-width`. Útil si tu diseño arranca en dispositivos móviles y necesitas realizar las adaptaciones a dispositivos mas grande.

```
@media (min-width: 768px) {
  body {
    background-color: lightgreen;
  }
}
```

Explicación: Esto significa que en pantallas de 768px o más, el fondo pasará a verde. Con esto podemos agregar complejidad a medida que crece el tamaño de pantalla.

3. Rango de anchos

También podemos aplicar estilos solo en un rango específico de tamaños de pantalla, combinando **min-width** y **max-width**. Esto es útil si queremos que un estilo se aplique únicamente, por ejemplo, en tablets, sin afectar ni a celulares más chicos ni a pantallas de escritorio más grandes.

```
@media (min-width: 600px) and (max-width: 1024px) {
  body {
    background-color: lightcoral;
  }
}
```

Explicación: En este caso, si la pantalla mide entre **600px y 1024px**, el fondo será coral. Es la forma de decir: “aplicá este estilo solo en este rango de dispositivos, ni más chicos ni más grandes”.

Ejemplo visual de grilla responsive:

En este ejemplo de responsive **vamos a continuar trabajando sobre la misma grilla que construimos previamente** (con `header`, `sidebar`, `main` y `footer`). La idea no es arrancar desde cero, sino **adaptar lo que ya hicimos** para que pueda visualizarse correctamente en distintos dispositivos: escritorio, tablets y móviles.

De esta manera, podés ver cómo el mismo esquema de grilla se transforma con media queries, sin perder la organización que ya construiste antes.

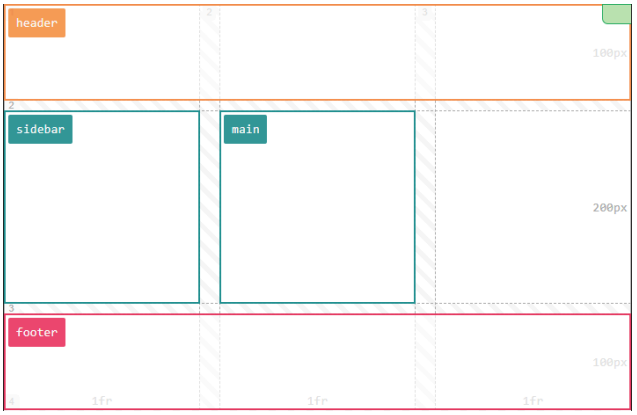
HTML

```
<div class="container">
  <header class="header">Header</header>
  <main class="main">Main</main>
  <aside class="sidebar">Sidebar</aside>
  <footer class="footer">Footer</footer>
</div>
```

Base (DESKTOP)

(se mantiene igual)

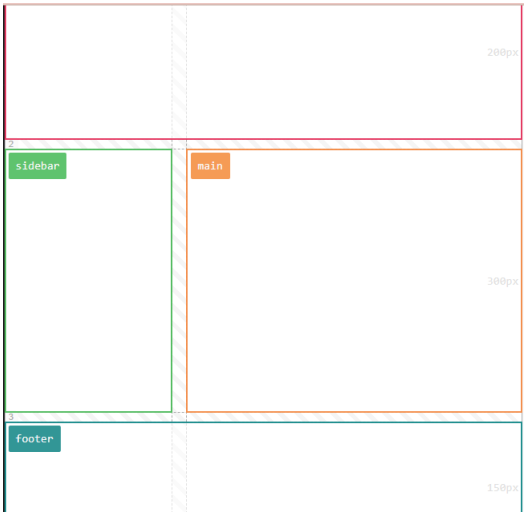
```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;
  grid-template-rows: 100px 200px 100px;
  grid-template-areas:
    "header header header"
    "sidebar main ."
    "footer footer footer";
  column-gap: 20px;
  row-gap: 10px;
}
.header  { grid-area: header; }
.sidebar { grid-area: sidebar; }
.main    { grid-area: main; }
.footer  { grid-area: footer; }
```



Tablet (<= 1024px)

Pasamos a **2 columnas**: *header* y *footer* siguen a lo ancho; *sidebar* a la izquierda y *main* a la derecha.

```
@media (max-width: 1024px) {
  .container {
    grid-template-columns: 1fr 2fr;
    grid-template-rows: auto 1fr auto;
/* más flexible en tablet */
    grid-template-areas:
      "header header"
      "sidebar main"
      "footer footer";
    column-gap: 16px;
    row-gap: 10px;
  }
}
```



Mobile (<= 600px)

Una **sola columna** para lectura vertical: *header* → *main* → *sidebar* → *footer*.

```
@media (max-width: 600px) {
  .container {
    grid-template-columns: 1fr;
    grid-template-rows: auto;
/* que fluya con el contenido */
    grid-template-areas:
      "header"
      "main"
      "sidebar"
      "footer";
    column-gap: 0;
    row-gap: 12px;
  }
}
```

Tips rápidos (manteniendo el enfoque desktop-first)

- Podés conservar las filas fijas en desktop (`100/200/100px`) y hacerlas **más flexibles** en tablet/mobile con `auto`/`minmax()` dentro de los media queries.
- Asegurá que las imágenes escalen: `img { max-width: 100%; height: auto; }`.
- Si necesitás que *sidebar* quede debajo de *main* también en tablet, reordená las áreas.

Con esto, tu layout **respeta la referencia de la grilla original** (desktop) y se adapta ordenadamente a tablet y teléfono usando solo overrides con `max-width`.

Escenarios de Diseño Responsive

El diseño responsive no se limita únicamente a las grillas: **se aplica a todos los elementos del proyecto** que necesiten adaptarse a distintos tamaños de pantalla.

Esto significa que, además de reorganizar la estructura general, puede ser necesario **ajustar la distribución de componentes, modificar tamaños de tipografía, redimensionar imágenes o incluso cambiar el orden de ciertos bloques de contenido**.

En otras palabras, el objetivo es garantizar que la experiencia de usuario sea clara y usable tanto en un celular como en una tablet o una computadora de escritorio.

1. Barra de navegación

En móviles, lo ideal es que los enlaces se apilen en columna para ser fáciles de tocar.

En pantallas más grandes, se organizan en fila.

```
.navbar {
  display: flex;
  flex-direction: column; /* móviles */
}

@media (min-width: 768px) {
  .navbar {
    flex-direction: row; /* tablet y desktop */
    justify-content: space-between;
  }
}
```

2. Contenido principal

En celulares conviene mostrar todo en una sola columna (lectura más cómoda).

En pantallas grandes podemos dividir en dos columnas para aprovechar el espacio.

```
.container {
  display: flex;
  flex-direction: column; /* móviles */
}

@media (min-width: 768px) {
  .container {
    flex-direction: row; /* tablet y desktop */
```

```
  }
}
```

3. Imágenes

Las imágenes deben escalar para no romper el diseño.

- En móvil: ocupar todo el ancho.
- En escritorio: reducir su tamaño para dejar espacio a otros elementos.

```
img {
  width: 100%;  /* móviles */
  height: auto;
}

@media (min-width: 1024px) {
  img {
    width: 50%;  /* escritorio */
  }
}
```

Conclusión:
El responsive design no se trata solo de ajustar la **grilla**: también implica **adaptar cada parte del sitio (menús, textos, imágenes, bloques de contenido)** para que se visualicen de forma clara y usable en cualquier dispositivo.

Recomendaciones:

Evitá crear una media query por cada necesidad puntual. En su lugar, agrupá un conjunto de cambios coherentes (layout, tipografías, espacios, navegación) dentro de los mismos breakpoints. Así mantenés un diseño estable, fácil de mantener y sin “parches” que se pisan entre sí.

Buenas prácticas

- Definí **pocos breakpoints claros** (ej. 600px, 768px, 1024px) y **probá que el diseño no se rompa** ni justo antes ni justo después del corte.
- Usá **unidades fluidas** (`fr`, `%`, `rem`) para reducir la cantidad de media queries necesarias.
- En cada breakpoint, **agrupá cambios relacionados** (layout + tipografía + espaciamiento + navegación).
- **Documentá** cada breakpoint con un comentario y mantené un **orden consistente** en el CSS.

4.5.Enfoque de Desarrollo Mobile First

El enfoque de desarrollo *mobile first* implica diseñar y desarrollar un sitio web inicialmente para dispositivos móviles y luego adaptarlo progresivamente para pantallas más grandes como tabletas y computadoras de escritorio. Este enfoque garantiza que el contenido y la funcionalidad principal estén accesibles y optimizados para los usuarios móviles, que representan una gran parte del tráfico web actual.

Pasos para Implementar un Diseño Mobile First

HTML

```
<div class="container">
  <header class="header">Header</header>
  <main class="main">Main</main>
  <aside class="sidebar">Sidebar</aside>
  <footer class="footer">Footer</footer>
</div>
```

CSS — Mobile First → Tablet → Desktop

1) Base: Mobile (una columna)

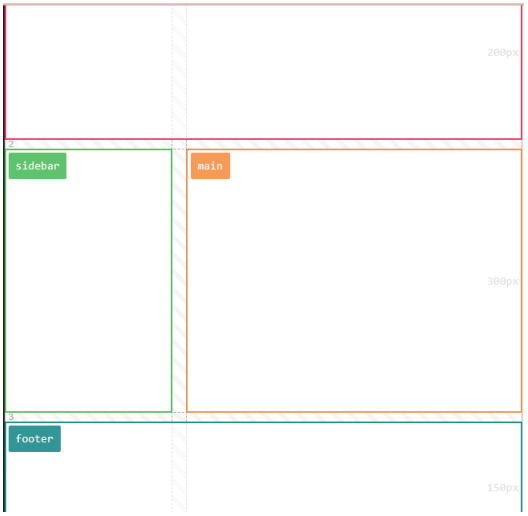
Orden cómodo de lectura vertical: `header → main → sidebar → footer`.

```
.container{
  display: grid;
  gap: 10px;
  grid-template-columns: 1fr;
  grid-template-rows: auto;
  grid-template-areas:
    "header"
    "main"
    "sidebar"
    "footer";
}

.header  { grid-area: header; }
.main    { grid-area: main; }
.sidebar { grid-area: sidebar; }
.footer  { grid-area: footer; }

/* Ayuda visual (opcional) */
.header  { border:1px solid #e63946; }
.main    { border:1px solid #457b9d; }
```

```
.sidebar { border:1px solid #2a9d8f; }
.footer  { border:1px solid #fb8500; }
```

2) Tablet (≥ 768px): dos columnas

`header` y `footer` a lo ancho; `sidebar` a la izquierda, `main` a la derecha.




```
@media (min-width: 768px){
 .container {
    grid-template-columns: 1fr 2fr;
    grid-template-rows: auto 1fr auto;
    grid-template-areas:
      "header header"
      "sidebar main"
      "footer footer";
  }

}
```

3) Desktop (≥ 1024px): tres columnas (layout original)

Recuperamos la tercera columna con el “punto” vacío a la derecha.

```
@media (min-width: 1024px){
  .container {
    grid-template-columns: 1fr 2fr 1fr;
    grid-template-rows: auto minmax(200px, 1fr) auto;
    grid-template-areas:
      "header  header  header"
      "sidebar main    ."
      "footer  footer  footer";
  }

}
```

Con este esquema, partís de **mobile** y vas **escalando** hasta llegar al **layout complejo** de escritorio, reutilizando exactamente las **mismas áreas** que trabajaron en el ejemplo anterior.

👉 Notá que ahora las filas son flexibles con `auto` y `minmax()`, no fijas en px.

Conclusión: min-width vs max-width en diseño responsive

Al trabajar con **media queries** es importante recordar la diferencia entre `min-width` y `max-width`, ya que esta elección define la estrategia de tu diseño responsive:

- **max-width**: se usa cuando el diseño inicial está pensado para pantallas grandes (desktop first). A partir de ahí, vas "reduciendo" y aplicando cambios a medida que la pantalla se hace más chica.
  Ejemplo: `@media (max-width: 600px)` → estilos para móviles.

- **min-width**: se usa en un enfoque mobile first. Comenzás diseñando para pantallas pequeñas y, a medida que aumenta el tamaño, agregás reglas que enriquecen el diseño.
  Ejemplo: `@media (min-width: 768px)` → estilos para tablets en adelante.

💡**Recordá**: no hay una única manera correcta, pero sí es clave ser consistente con la estrategia elegida. En este caso, si venís trabajando con `min-width`, mantenelo para asegurar coherencia en todo el proyecto y evitar confusiones.

4.6. Actividad práctica

⚙️ Actividad Práctica — Responsive con Grid + Flexbox

En esta actividad vas a aplicar los conceptos de CSS Grid, Flexbox y Box Modeling para construir un layout responsive real en tu proyecto.

🎯 Objetivo

Hacer responsive el index y una página más de tu proyecto.
El layout debe adaptarse correctamente a distintos tamaños de pantalla utilizando CSS Grid para la estructura principal, Flexbox para organizar componentes internos, y media queries para los puntos de quiebre.

📌 Alcance (elegí una opción)

- **Opción A · Una sección:**
  Aplicá la consigna a una sección clave (por ejemplo: *hero*, *features*, galería o contacto).

- **Opción B · Varias secciones:**
  Repetí el proceso en 2 o 3 secciones (por ejemplo: *hero* + grilla de tarjetas +

footer), asegurando coherencia visual y técnica en todo el layout.

🧱 Requisitos Técnicos

- **Utilizá CSS Grid para organizar el layout principal de las secciones (por ejemplo: columnas, filas, áreas).**
- **Utilizá Flexbox dentro de esas secciones para alinear y distribuir componentes internos (por ejemplo: elementos del navbar, tarjetas en fila, botones, etc.).**
- **Aplicá media queries siguiendo una estrategia mobile first con `min-width` para definir los puntos de quiebre.**
- **Ajustá márgenes, paddings y bordes usando las propiedades de Box Modeling (`margin`, `padding`, `border`, etc.) para lograr un espaciado limpio y consistente.**
- **Usá la propiedad moderna `gap` para controlar el espaciado entre filas y columnas en las grillas. Si querés, podés mencionar `row-gap` y `column-gap` como propiedades históricas, pero no deben ser parte de los ejemplos principales.**

📱 Responsive — Puntos Clave

- **Mobile (base): empezá diseñando para pantallas pequeñas con una sola columna.**
- **Tablet (≥ 768px): reorganizá las áreas usando Grid para aprovechar el espacio horizontal.**
- **Desktop (≥ 1024px): escalá el layout a una grilla más compleja (por ejemplo, tres columnas o layout con sidebar).**

Ejemplo de estructura típica Mobile → Tablet → Desktop:

```
.container {

  display: grid;

  gap: 10px;
```

```
  grid-template-columns: 1fr;
  grid-template-areas:
    "header"
    "main"
    "footer";
}

@media (min-width: 768px) {
  .container {
    grid-template-columns: 1fr 2fr;
    grid-template-areas:
      "header header"
      "sidebar main"
      "footer footer";
  }
}

@media (min-width: 1024px) {
  .container {
    grid-template-columns: 1fr 2fr 1fr;
    grid-template-areas:
      "header header header"
      "sidebar main ."
      "footer footer footer";
```

```
  }
}
```

📝 Aclaraciones Importantes

- Ambas páginas (index + la elegida) deben ser totalmente responsive, incluyendo:
    - Grillas principales.
    - Componentes internos (nav, cards, aside, footer, etc.).
- Mantené **una estrategia única** en todo el proyecto:
    - Si usás mobile first con min-width, sostenelo en todas las hojas.
    - No mezcles enfoques (max-width en algunas y min-width en otras), ya que eso genera inconsistencias.
- Probá manualmente el comportamiento **estrechando y ampliando la ventana** para asegurarte de que el layout no “se rompa” en las transiciones entre breakpoints.

✨ Sugerencia Extra

Podés cerrar la actividad con una pequeña **galería responsive** de 3 columnas que pase a 2 en tablet y 1 en mobile, para reforzar los conceptos:

```
.gallery {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
```

```
  gap: 10px;
}

@media (max-width: 768px) {
  .gallery { grid-template-columns: repeat(2, 1fr); }
}

@media (max-width: 480px) {
  .gallery { grid-template-columns: 1fr; }
}
```

⚙️Actividad +IA (opcional)

Luego de terminar tu trabajo responsive con **Grid, Flexbox, box-modeling y media queries**, utilizá una herramienta de IA para **testear, validar y optimizar tu diseño**.

Podés apoyarte en **ChatGPT, Galileo AI o Uizard**, compartiendo tu código y pidiéndole:

- Revisar si tu estructura es consistente en mobile, tablet y desktop.
- Señalar posibles problemas de accesibilidad o semántica.
- Sugerir mejoras en la organización de las grillas y flexbox.
- Proponer alternativas para reducir media queries innecesarias.

💡 **Ejemplo de prompt**:

> “Revisá este código CSS y HTML. Mi objetivo es que sea responsive con Grid y Flexbox. Indicame si las media queries están bien planteadas o si puedo unificarlas. También recomendame mejoras para que se adapte mejor a distintos dispositivos.”

De esta forma, usás la IA como un **validador y asistente técnico**, no como reemplazo de tu trabajo. Primero construís tu solución, y luego pedís sugerencias de mejora.