GRID

1. Introducción

El sistema flex está orientado a estructuras de una sola dimensión, por lo que puede ser laborioso crear estructuras más complejas. Necesitamos algo más potente para estructuras web de varias dimensiones rápidamente. Con el paso del tiempo, muchos frameworks CSS y librerías comenzaron a utilizar un sistema basado en un grid donde se definía una cuadrícula a la que era posible darle tamaño, posición o colocación, cambiando el nombre de sus clases.

Grid nace de esa necesidad, obteniendo las ventajas de ese sistema grid, añadiéndole numerosas mejoras y características que permiten crear rápidamente cuadrículas flexibles y potentes de forma prácticamente instantánea con una nueva familia de propiedades CSS.

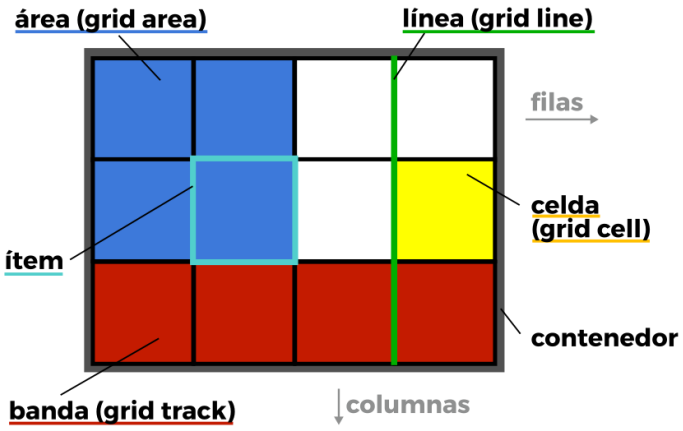
Antes de comenzar con grid es conveniente conocer el sistema flex, ya que grid toma la filosofía (y muchas de las bases y conceptos) que se utilizan en flex.

Por último, puedes encontrar la especificación de W3C para el sistema grid en:

https://www.w3.org/TR/css-grid-1/

2. Conceptos previos

Para crear diseños basados en grid necesitaremos tener en cuenta una serie de conceptos que utilizaremos a partir de ahora y que definiremos a continuación:

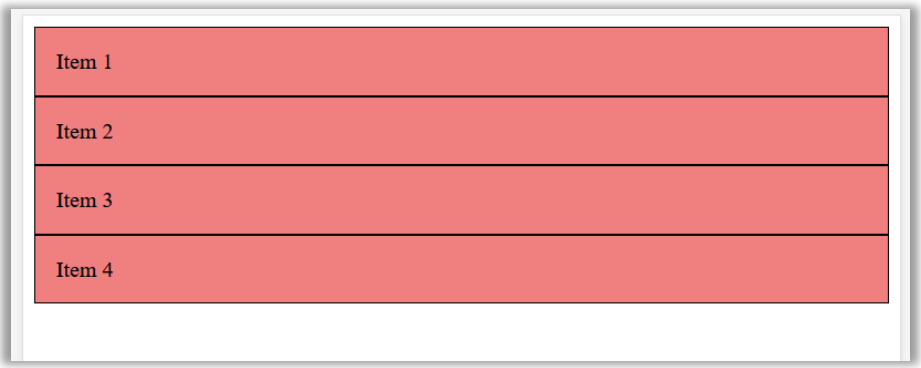


En la imagen, tenemos:

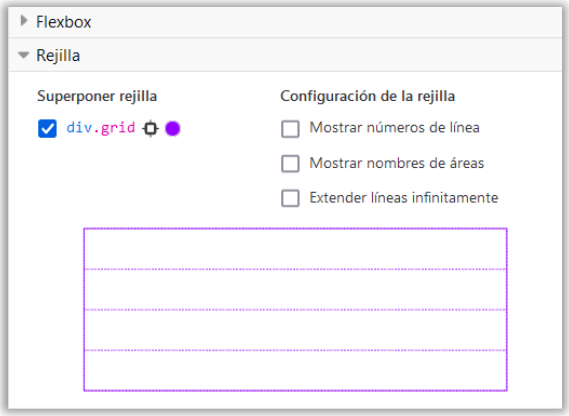
- **Contenedor:** El elemento padre contenedor que definirá la cuadrícula.
- **Ítem:** Cada uno de los hijos que contiene la cuadrícula.
- **Celda (*grid cell*):** Cada una de las celdas de la cuadrícula.
- **Area (*grid area*):** Región o conjunto de celdas de la cuadrícula.
- **Banda (*grid track*):** Banda horizontal o vertical de celdas de la cuadrícula.
- **Línea (grid line):** Separador horizontal o vertical de las celdas de la cuadrícula.

Imaginemos el siguiente escenario, un contenedor grid y 3 ítems en su interior:

```
<style>
  .grid {
    display: grid;
  }

  .item {
    background-color: lightcoral;
    border: 1px solid black;
    padding: 20px;
  }
</style>

<div class="grid">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
</div>
```



Visualmente no parece haber hay diferencia entre aplicar `display: grid` al contenedor o no hacerlo. Sin embargo, al aplicárselo e inspeccionar el elemento con el inspector de Firefox podemos ver que se activa una curiosa herramienta, que utilizaremos con regularidad:



Como pasaba con flex, el valor de `display` puede ser `grid` o `inline-grid`, dependiendo de cómo queramos que se comporte el contenedor, si como un elemento de línea o como un elemento de bloque.

3. Propiedades para definir filas y columnas

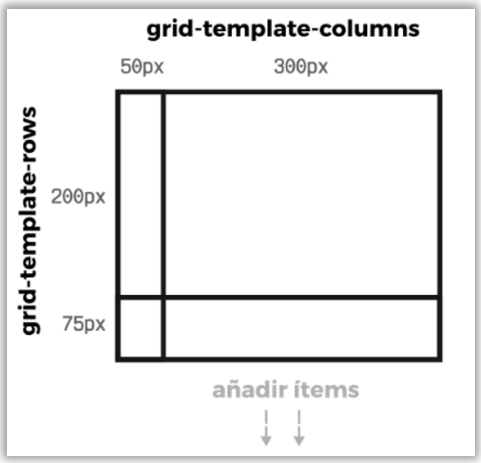
Las propiedades `grid-template-rows` y `grid-template-columns` permiten establecer el número y tamaño de cada fila y columna.

Propiedad	Significado
`grid-template-rows`	Establece el número y tamaño de filas.
`grid-template-columns`	Establece el número y tamaño de columnas.

Conociendo estas dos propiedades, veamos el siguiente código CSS:

```css
.grid {
  display: grid;
  grid-template-columns: 50px 300px;
  grid-template-rows: 200px 75px;
}
```

Con `display: grid` definimos que queremos crear un grid y con las propiedades `grid-template-rows` y `grid-template-columns` definimos los tamaños de las columnas y las filas del mismo. Esto significa que, a priori, tendríamos una cuadricula de 2x2, 4 celdas en total. Puedes utilizar las *devtools* del navegador para comprobar cómo varía el tamaño de las filas y columnas modificando los valores de estas propiedades.



Corre de nuestra cuenta vigilar que el número de elementos hijos en el grid es el que debería. Dependiendo del número de elementos hijos que tenga definido el contenedor grid en su HTML, tendremos una cuadrícula de 2x2 elementos (4 ítems), 2x3 elementos (6 ítems), 2x4 elementos (8 ítems) y así sucesivamente. Si el número de ítems es impar (por ejemplo, 5 ítems), la última celda de la cuadrícula se quedaría vacía.

A medida que fuéramos incluyendo más ítems en el grid, podríamos aumentar también el número de parámetros de la propiedad `grid-template-columns` y/o la propiedad `grid-template-rows`.

Si tenemos menos ítems del número total de celdas definidas con estas dos propiedades, simplemente las celdas vacías quedarían sin rellenar. Veremos más adelante qué ocurre si tenemos más ítems de lo que se ha definido con estas dos propiedades.

3.1. `grid-template-rows` y `grid-template-columns`

Como ya hemos comentado, estas dos propiedades permiten establecer el número y tamaño de filas y columnas que tendrá nuestro grid.

Para ambas propiedades los valores posibles son iguales. Tanto en la tabla como en los ejemplos siguientes usaremos la propiedad `grid-template-columns` por no repetir, pero con `grid-template-rows` sería igual.

Valor	Significado
`none`	La fila o columna sólo se creará si es necesario y su tamaño se determinará por el valor de grid-auto-rows o grid-auto-columns.
`auto`	El tamaño de la columna se determina por el tamaño del contenedor y por el tamaño de los ítems de la columna.
`max-content`	Establece el tamaño de cada columna para que dependa del elemento más grande de la columna.
`min-content`	Establece el tamaño de cada columna para que dependa del elemento más pequeño de la columna.
`length (px, em, %...)`	Establece el tamaño de las columnas utilizando un valor de longitud (px, em, %...).

Tanto en esta tabla como en las siguientes, el valor subrayado indica el valor por defecto de la propiedad.

Dado que el valor por defecto es `none`, si no declaramos una de estas dos propiedades, las filas o columnas se crearán sólo si es necesario. Prueba a eliminar la declaración de cada una de estas dos propiedades para observar el comportamiento del contenedor grid y los ítems hijos.

3.1.1. `auto`

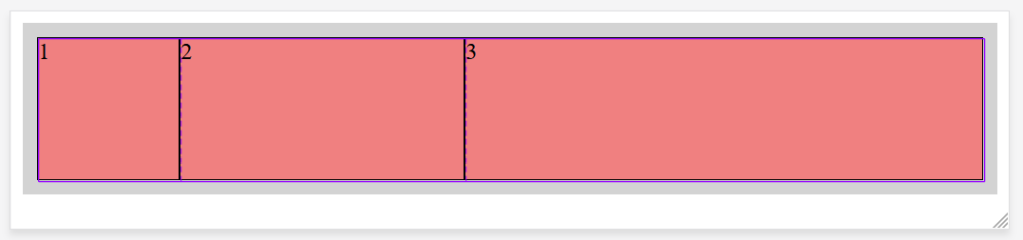
El tamaño de la columna se determina por el tamaño del contenedor y por el tamaño de los ítems de la columna.

Esto significa que se toma el espacio necesario para el contenido, aunque se podría reducir en caso de que haya otras columnas interfiriendo y no quepan. También se podría adaptar en caso de que sobre espacio. Si sobra espacio se reparte por igual entre las filas o columnas, pero a partir del tamaño calculado en función del contenido que ya tengan.

Con este valor hay que tener muy en cuenta si estamos trabajando con un contenedor de bloque o de línea, es decir, si el contenedor es `display: grid` o `display: inline-grid`. La razón es que `auto` se adapta tanto a su contenido como a su contenedor.

Fíjate en el siguiente ejemplo en el que tenemos `auto` en la tercera columna:

```
<style>
  .container {
    display: grid;
    padding: 10px;
    background-color: lightgrey;
    grid-template-rows: 100px;
    grid-template-columns: 100px 200px auto;
  }

  .item {
    background-color: lightcoral;
    border: 1px solid black;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```



El valor **auto** está haciendo que la anchura de la tercera columna se adapte al contenedor. Es el contenedor quien se está estirando hasta alcanzar toda la anchura disponible y, con ello, está *tirando* de la anchura de la tercera columna.

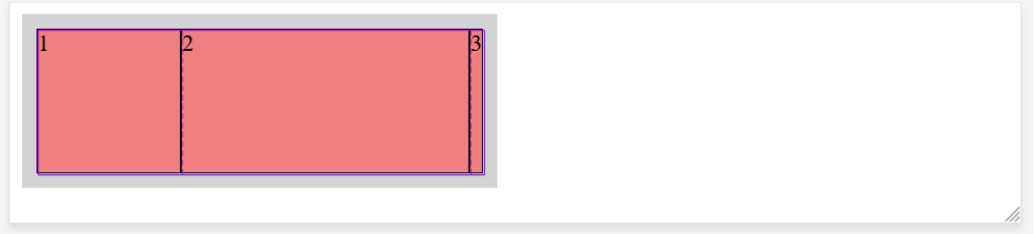
Sin embargo, si cambiamos a:

```
display: inline-grid;
```

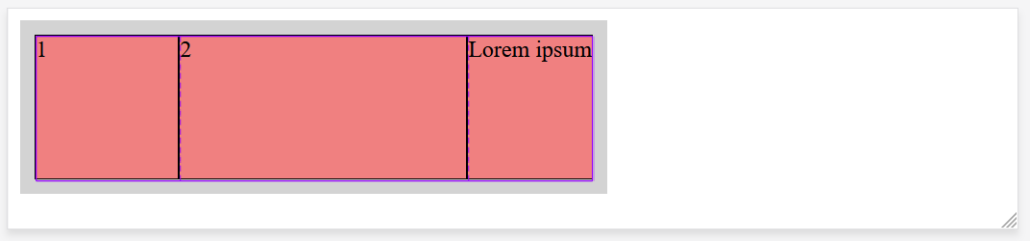
El contenedor dejará de estirarse y será tan ancho como hayamos definido en las columnas:

```
grid-template-columns: 100px 200px auto;
```

Es decir, la primera ocupará 100px, la segunda 200px y la tercera se adaptará a su contenedor y su contenido. Como ahora el contenedor no se estira, la columna ocupará lo que ocupa su contenido:



Si aumentamos su contenido aumentamos también la anchura de la columna:

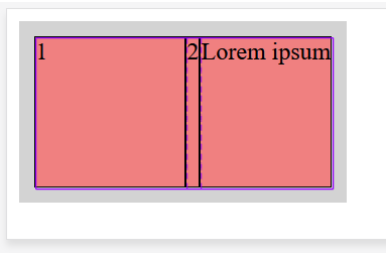


Pero, ¿qué ocurre cuando tenemos dos columnas `auto`?

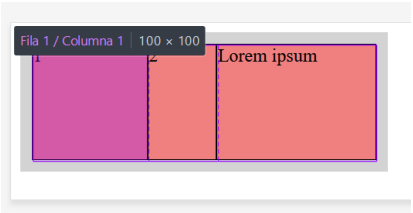
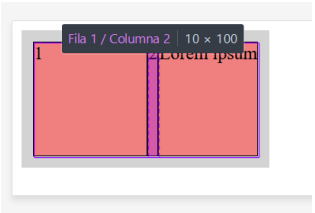
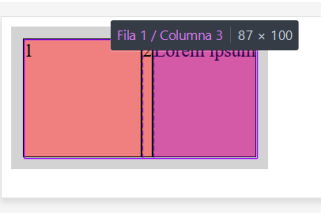
Cuando el navegador va a renderizar un grid, primero calcula el espacio que necesita darle a cada celda que tenga el tamaño en `auto` para alojar el contenido que tengan. Después, reparte el espacio disponible dividiéndolo en las fracciones (`fr`) que queden (si están indicadas). Si no están indicadas, como es este caso, reparte el espacio restante a partes iguales entre los elementos con valor `auto`.

Siguiendo con el ejemplo de antes, donde teníamos `display: inline-grid` en el contenedor, ahora ponemos dos columnas `auto`:

```
grid-template-columns: 100px auto auto;
```



Fijémonos en el tamaño total que nos indica el inspector de Firefox y en la anchura de las columnas:

Columna 1: 100px	Columna 2: 10px	Columna 3: 87px
		

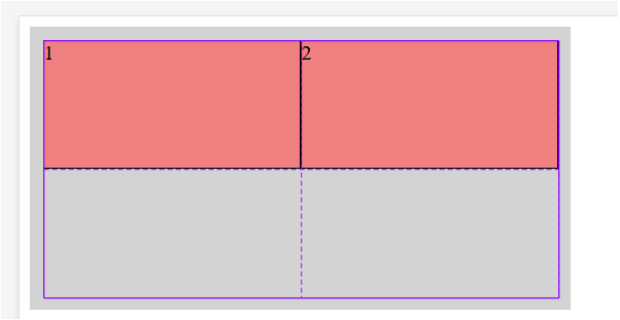
Lo que nos da una anchura total de **197px**, la suma de las 3 columnas.

Si le damos al contenedor grid 297px de anchura, esos 100px más se repartirán a partes iguales entre las dos columnas con el valor `auto`, pasando la segunda de 10 a 60px y la tercera de 87 a 137px. Puedes probarlo con el inspector del navegador o modificando el ejemplo.

Otro aspecto importante es que, **con `auto`, si no hay ítems para cubrir la fila o columna, no se reserva espacio para ella**. Prueba a eliminar un ítem en el ejemplo anterior para ver lo que ocurre.

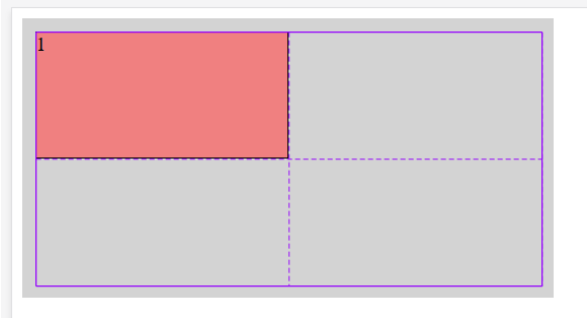
Fíjate en el siguiente ejemplo:

```
<style>
  .container {
    display: inline-grid;
    padding: 10px;
    background-color: lightgrey;
    grid-template-rows: 100px 100px;
    grid-template-columns: 200px 200px auto;
  }

  .item {
    background-color: lightcoral;
    border: 1px solid black;
  }
</style>

<body>
  <div class="container">
    <div class="item">1</div>
    <div class="item">2</div>
  </div>
</body>
```



Aunque estamos definiendo un grid de 3 columnas, la tercera no se crea ni se reserva espacio para ella porque solo tenemos 2 ítems. Esto ocurre porque tenemos declarado `display: inline-grid` en el contenedor, que hace que éste ocupe sólo el ancho necesario. Si lo cambiamos a `display: grid`, el ancho sobrante se lo llevará esta columna con valor `auto`.

Sin embargo, si solo metemos un ítem en el grid, ¿se reservará espacio para la segunda columna? La respuesta es sí, se reservarán los 100px de espacio dado que hemos definido que la según da columna ocupará 100px:



3.1.2. Unidad fraccional (fr)

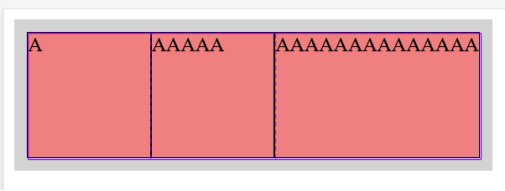
La unidad `fr`, descrita en la especificación, se utiliza para distribuir el espacio disponible entre las filas o columnas en un contenedor grid. Los diferentes valores de `fr` asignados a filas o columnas especifican cómo se repartirá el espacio disponible.

Fíjate en el siguiente ejemplo, donde usamos 3 columnas con unidades `fr`:

```
<style>
  .container {
    display: grid;
    width: 600px;
    padding: 10px;
    background-color: lightgrey;
    grid-template-rows: 100px;
    grid-template-columns: 1fr 1fr 1fr;
  }

  .item {
    background-color: lightcoral;
    border: 1px solid black;
  }
</style>

<div class="container">
  <div class="item">A</div>
  <div class="item">AAAAA</div>
  <div class="item">AAAAAAAAAAAAAA</div>
</div>
```

Cada columna ocupa 200px porque hemos dado una anchura de 600px en total al contenedor y el contenido de las 3 columnas cabe perfectamente en esos 200px.

Sin embargo, si reducimos esos 600px vemos que las columnas se irán haciendo más estrechas todas al mismo tiempo, pero cuando la tercera, que tiene más contenido, alcanza el límite de contenido deja de estrecharse para evitar el desbordamiento.

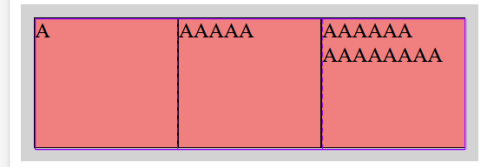


El tamaño donde esta tercera columna deja de estrecharse es `min-content`.

¿Qué ocurre si damos un espacio al contenido de la tercera columna? Algo así:

```
<div class="item">AAAAAA AAAAAAAA</div>
```

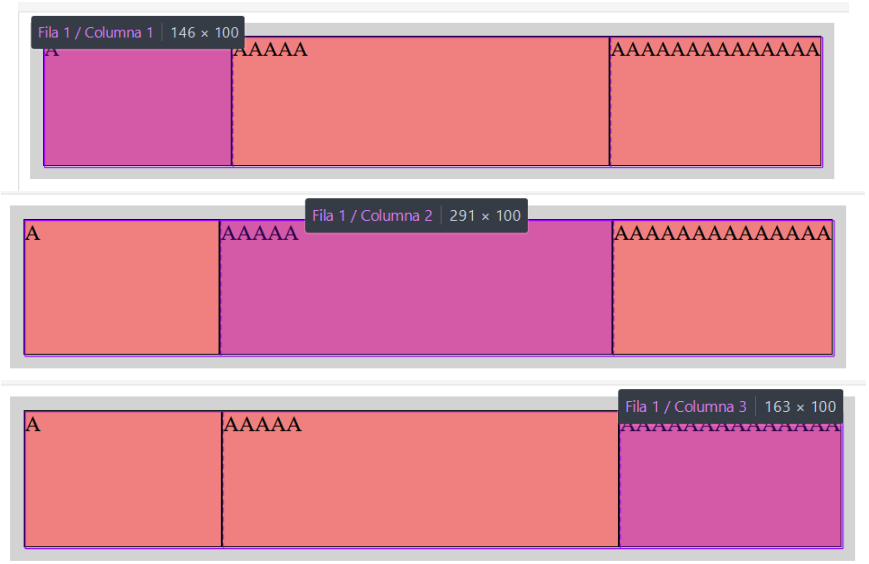
Observa el comportamiento:



Probemos ahora con esta situación:

```
grid-template-columns: 1fr 2fr 1fr;
```

Podríamos pensar que ahora, con 600px de ancho, la primera y la tercera columna ocuparían ambas 150px y la segunda ocuparía 300px. Pero no, eso ocurriría si hubiera espacio disponible para mostrax r el contenido de todas las columnas, puedes probarlo dando más anchura al contenedor. En realidad, la tercera columna ocupa más que la primera porque lo necesita para evitar el desbordamiento de su contenido.



3.1.3. Diferencia entre `fr` y `auto`

Observa el siguiente ejemplo:

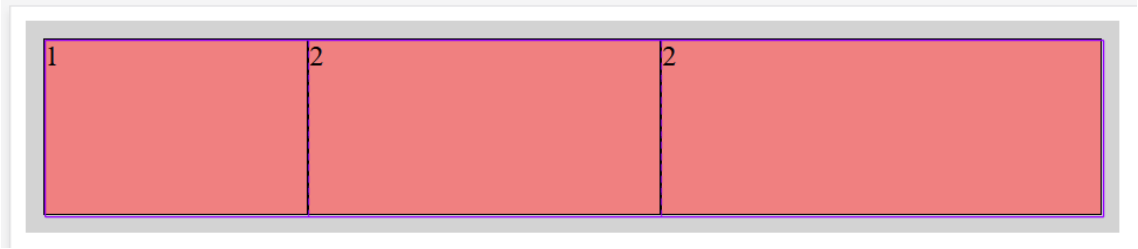
```
<style>
  .container {
    display: grid;
    width: 600px;
    padding: 10px;
    background-color: lightgrey;
    grid-template-rows: 100px;
    grid-template-columns: 1fr 1fr 1fr;
  }

  .item {
    background-color: lightcoral;
    border: 1px solid black;
  }
</style>

<div class="container">
  <div class="item" style="min-width: 50px;">1</div>
  <div class="item" style="min-width: 100px;">2</div>
  <div class="item" style="min-width: 150px;">3</div>
</div>
```

Se muestra cada columna con 200px de ancho porque `fr` calcula cuánto necesita para cada columna (50+100+150=200px) y **reparte el espacio restante hasta que todas alcancen el mismo ancho**.

Ahora probemos con:

```
grid-template-columns: auto auto auto;
```

Ahora las columnas ocupan 150px, 200px y 250px respectivamente. El motivo es que `auto` **reparte el espacio restante a partes iguales contando con el contenido**. Como está definido el espacio mínimo para cada columna con `min-width`, el espacio disponible es de 300px (600-50-100-150). Esos 300px dan para 100px a cada columna, por eso ocupan 150, 200 y 250px cada una de ellas.



Por tanto, la diferencia principal entre `auto` y `1fr` radica en cómo distribuyen el espacio disponible dentro de un grid.

- `auto`: Primero se calcula el espacio mínimo ocupará que cada fila o columna y después se distribuye el espacio restante entre ellas a partes iguales.
- `fr`: El espacio disponible se reparte proporcionalmente entre las filas o columnas en función del valor de `fr` independientemente del contenido.

En resumen, `auto` **se ajusta al tamaño del contenido, mientras que** `fr` **distribuye el espacio disponible entre las columnas o filas de manera equitativa**.

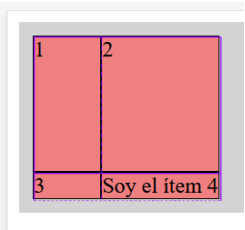
3.1.4. `min-content` y `max-content`

La especificación de W3C define estos dos valores de la siguiente manera (el valor `fit-content` no es válido para estas dos propiedades):

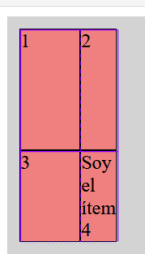
- `min-content` es el tamaño más pequeño que puede tomar un elemento sin desbordar su contenido.
- `max-content` es el tamaño "ideal" de un elemento en un eje dado cuando tiene disponible un espacio infinito.

Observa este ejemplo:

```
<style>
  .container {
    display: grid;
    padding: 10px;
    background-color: lightgrey;
    grid-template-rows: 100px;
    grid-template-columns: 50px max-content;
  }
  .item {
    background-color: lightcoral;
    border: 1px solid black;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">Soy el ítem 4</div>
</div>
```



La segunda columna se ensancha tanto como lo hace su celda más ancha. Si ahora cambiamos `max-content` por `min-content`, la anchura de la columna será la mínima posible que permita mostrar el contenido de cada celda de la columna.



En este caso, es la palabra "ítem" quien está estableciendo esa anchura mínima. Prueba escribiendo otra palabra más larga para comprobar el comportamiento.

3.1.5. El problema de los porcentajes

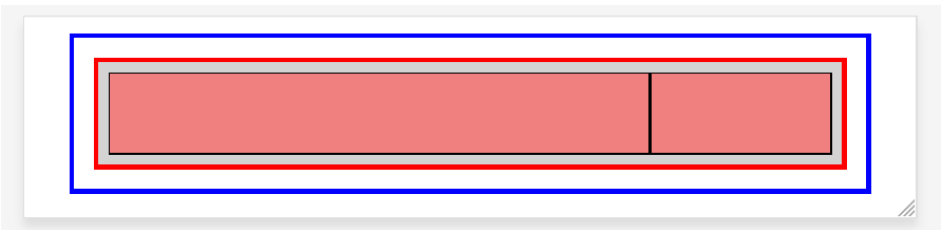
Hay una situación en la que utilizar porcentajes para dar anchura a las columnas de un grid no funciona como desearíamos. Considera la siguiente situación:

```
<style>
  * {
    box-sizing: border-box;
  }

  .container {
    border: 3px solid blue;
    width: 400px;
    margin: 0 auto;
  }

  .mi-grid {
    display: grid;
    grid-template-columns: 75% 25%;
    margin: 10px;
    border: 3px solid red;
    padding: 5px;
    background-color: lightgray;
  }

  .item {
    border: 1px solid black;
    background: lightcoral;
    padding: 20px;
  }
</style>

<div class="container">
  <div class="mi-grid">
    <div class="item"></div>
    <div class="item"></div>
  </div>
</div>
```

Fíjate en la propiedad:

```
grid-template-columns: 75% 25%;
```

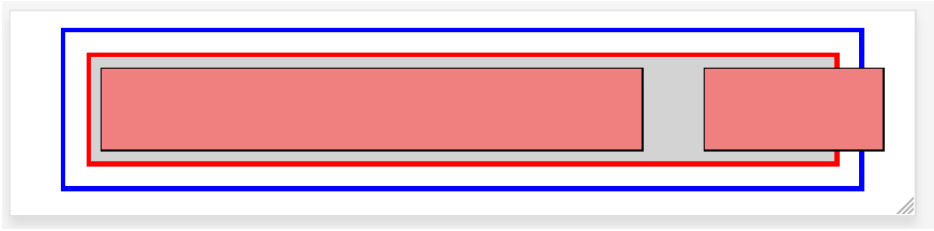
El navegador nos muestra claramente nuestro contenedor con borde azul, el grid con borde rojo y cada uno de los 2 ítems, ocupando el primero el 75% y 25% el segundo:



Pero, ¿qué ocurre si queremos introducir un gap entre los dos ítems? Digamos:

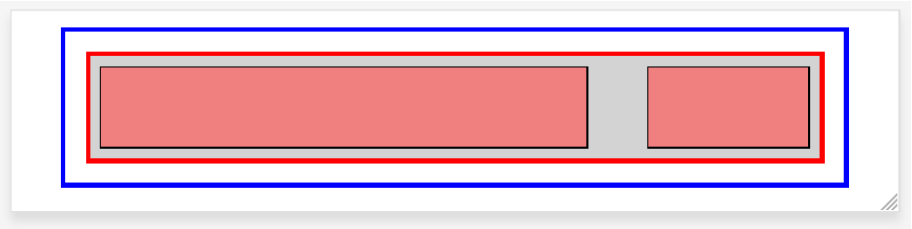
```
.mi-grid {
  ...
  gap: 30px;
  ...
}
```

Pues que tendremos desbordamiento:



Esto ocurre porque **la propiedad `gap` no entra en el cálculo que el navegador hace para dar anchura a los ítems**. Para mantener que el primer ítem sea 3 veces más ancho que el segundo podemos usar `fr` en lugar de porcentajes:

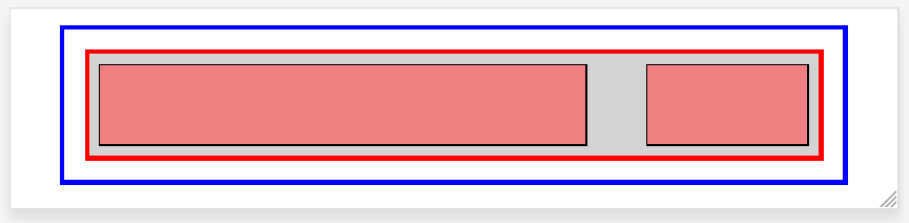
```
grid-template-columns: 3fr 1fr;
```



Esta es una diferencia importante con respecto al comportamiento de flex. Cambiemos el código para ver el mismo ejemplo con flex:

```
.mi-grid {
  display: flex;
  ...
}

.mi-grid > .item:nth-child(1) {
  flex-basis: 75%;
}

.mi-grid > .item:nth-child(2) {
  flex-basis: 25%;
}
```

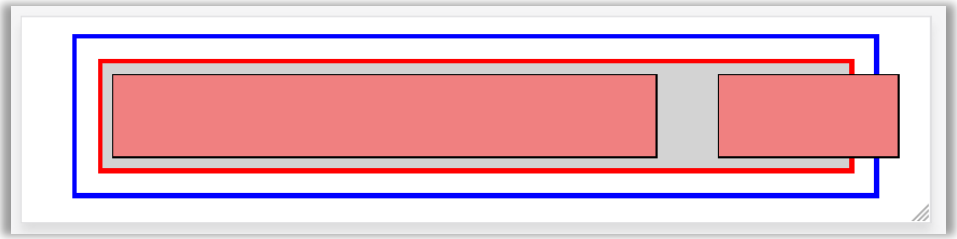
Obtendremos exactamente lo mismo que obtuvimos con grid usando `fr`:



La razón de que aquí sí funcionen los porcentajes no es porque estemos usando `flex-basis`, puedes cambiarlo por `width` y ver que sigue funcionando bien. La razón es que **en `flex` existe una propiedad**, `flex-shrink`, activada por defecto, que **obliga a los ítems a encogerse si no caben en su contenedor**. **No existe una propiedad análoga en** `grid`, de ahí la diferencia de comportamiento.

Podemos probar a poner esta propiedad a 0 y volveremos a ver desbordamiento:

```
.item {
  flex-shrink: 0;
  ...
}
```



3.1. La función `minmax()`

La función `minmax()` se puede utilizar como valor para definir rangos flexibles de filas o columnas. Funciona de la siguiente forma:

```
grid-template-columns: minmax(tamaño-min, tamaño-max);
```

Prueba con este ejemplo:

```
.grid {
  display: grid;
  grid-template-columns: repeat(2, minmax(400px, 600px));
  grid-template-rows: repeat(2, 1fr);
}
```

Si redimensionas la ventana del navegador verás que el tamaño de las columnas se amolda a la anchura disponible siempre que su tamaño de columna esté entre 400px y 600px.

3.2. La función `repeat()`

En algunos casos, con `grid-template-columns` y `grid-template-rows` podemos necesitar indicar las mismas cantidades un número alto de veces, resultando repetitivo y molesto escribirlas varias veces. Se puede utilizar la función `repeat()` para indicar repetición de valores, señalando el número de veces que se repiten y el tamaño en cuestión. La expresión a utilizar sería la siguiente:

```
repeat(número de veces, tamaño)
```

Por ejemplo:

```
grid-template-columns: 100px repeat(4, 50px) 200px;
grid-template-rows: repeat(2, 1fr 2fr);
```

Es equivalente a:

```
grid-template-columns: 100px 50px 50px 50px 50px 200px;
grid-template-rows: 1fr 2fr 1fr 2fr;
```

Asumiendo que tuviéramos un contenedor grid con 24 ítems hijos en el HTML, el ejemplo anterior crearía una cuadrícula con 6 columnas y 4 filas. Recuerda que en el caso de tener más ítems hijos, el patrón se seguiría repitiendo.

3.2.1. Las opciones `auto-fill` y `auto-fit`

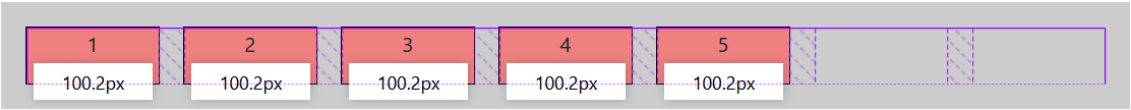
Cuando usamos la función `repeat()` hay dos palabras claves que podemos utilizar: `auto-fill` o `auto-fit`. Con ellas indicamos al navegador que queremos que **rellene o ajuste** el contenedor grid con múltiples elementos hijos dependiendo del tamaño del contenedor.

Observa el siguiente ejemplo disponible en codepen:

Fíjate en la línea:

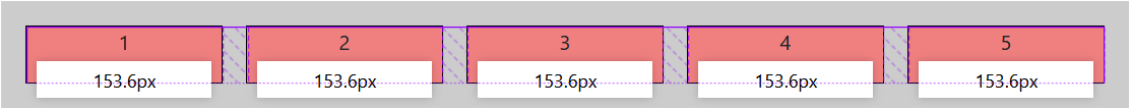
```
grid-template-columns: repeat(auto-fill, minmax(100px, 1fr));
```

Con `auto-fill` le decimos al navegador que cree tantas columnas como sea posible, siempre y cuando su tamaño mínimo sea de 100 pixeles y su tamaño máximo sea de una fracción (`1fr`) de la anchura disponible del grid.



Esto crea tantas columnas como quepan en cada fila, existan los ítems en el HTML o no. Prueba a redimensionar el contenedor, verás como cuando los ítems que sí existen van a bajar de 100px se pasan a la siguiente línea.

Con `auto-fit`, en cambio, el navegador ajusta el tamaño de las columnas hasta rellenar todo el espacio disponible, sin crear columnas “invisibles”.



Resumiendo:

- `auto-fill` rellena cada fila con tantas columnas como quepan. Las columnas pueden estar vacías, pero seguirán ocupando un espacio en la fila.
- `auto-fit` ajusta las columnas existentes expandiéndolas para que ocupen el espacio disponible.

Un aspecto **importante** a tener en cuenta es que cuando se usa `repeat()` no podemos usar dentro de `minmax()` valores como `auto`, `min-content` o `max-content`, tal y como se indica en la especificación: *Automatic repetitions (auto-fill or auto-fit) cannot be combined with intrinsic or flexible sizes.*

3.3. La función `fit-content(longitud)`

Esta función actúa como `max-content` al principio. Sin embargo, una vez que la fila/columna alcanza el tamaño pasado a la función el contenido comienza a ajustarse. Por ejemplo, `fit-content(100px)` creará una fila/columna de menos de 100px si el tamaño `max-content` es inferior a 100px, pero nunca mayor de 100px.

Puedes ver un ejemplo y una explicación de esta función en codepen.

3.4. Huecos (*gaps*): `row-gap` y `column-gap`

Por defecto, la cuadrícula tiene todas sus celdas pegadas a sus celdas contiguas. Aunque sería posible separarlas mediante `margin`, existe una forma más apropiada: los huecos o *gutters*.

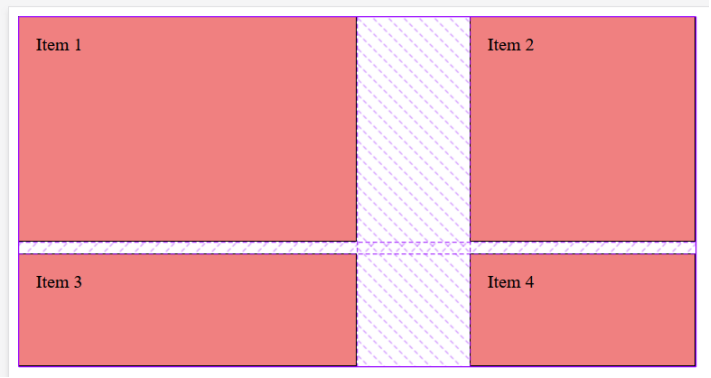
Para especificar estos espacios entre celdas podemos utilizar las propiedades `column-gap` y/o `row-gap`, que ya usamos en `flex`.

Propiedad	Valores posibles	Significado
`row-gap`	Número (px, %...), `normal` (por defecto)	Espacio entre filas.
`column-gap`	Número (px, %...), `normal` (por defecto)	Espacio entre columnas.

Observa el siguiente ejemplo de un grid de 2x2 donde establecemos un espacio entre filas de 10px y entre columnas de 100px:

```
.grid {
  display: grid;
  grid-template: 200px 100px / 300px 200px;
  column-gap: 100px;
  row-gap: 10px;
}
```

Con `column-gap` establecemos un hueco de 100px entre celdas de distintas columnas, mientras que con `row-gap` establecemos un hueco de 10px entre celdas de distintas filas. Nos quedaría algo similar a esto:



Es posible que encuentres estas propiedades por su nombre anterior, pero son las mismas: `grid-row-gap` y `grid-column-gap`.

3.4.1. Atajo: `gap`

La propiedad `gap` es un atajo para las propiedades `column-gap` y `row-gap` (también para `flex`) que permite indicar el valor de estas propiedades de una sola vez.

Con 1 parámetro establecemos el valor de las dos propiedades a la vez. Así:

```
gap: 40px;
```

Es equivalente a:

```
row-gap: 40px;
column-gap: 40px;
```

Con 2 parámetros sería:

```
gap: <row-gap> <column-gap>
```

Por ejemplo:

```
gap: 20px 80px;
```

Es equivalente a:

```
row-gap: 20px;
column-gap: 80px;
```

¡Importante! Al principio, las propiedades `column-gap`, `row-gap` y `gap` eran conocidas como `grid-column-gap`, `grid-row-gap` y `grid-gap`, por lo que aún puede que encuentres información obsoleta que las mencione. Hoy en día deberías utilizar las tres primeras en su lugar.

4. Propiedades de alineación

4.1. Alineaciones globales

Existen una serie de propiedades que se pueden utilizar para colocar y ajustar nuestro grid dentro del contenedor padre o ajustar los ítems dentro de él. Algunas de estas propiedades probablemente ya las conocerás de flex, sin embargo, en grid pueden tener un comportamiento diferente.

Propiedad	Descripción
`justify-items`	Alinea **los ítems** en **horizontal** dentro de cada celda. Especificación
`align-items`	Alinea **los ítems** en **vertical** dentro de cada celda. Especificación
`justify-content`	Alinea **el contenido** (la cuadricula) en **horizontal** en el contenedor padre. Especificación
`align-content`	Alinea **el contenido** (la cuadricula) en **vertical** en el contenedor padre. Especificación

Los posibles valores de cada propiedad son los siguientes:

Propiedad	Posibles valores
`justify-items`	`start` | `end` | `center` | `stretch` | `right` | `left`
`align-items`	`start` | `end` | `center` | `stretch`
`justify-content`	`start` | `end` | `center` | `stretch` | `space-around` | `space-between` | `space-evenly`
`align-content`	`start` | `end` | `center` | `stretch` | `space-around` | `space-between` | `space-evenly`

Un aspecto **importante** es que `justify-content` y `align-content` alinean el grid dentro de su contenedor, mientras que `justify-items` y `align-items` alinean los ítems dentro del grid.

Puedes ver un ejemplo interactivo del funcionamiento de estas 4 propiedades en estos dos *codepens*. La diferencia entre ellos es que en el segundo se utilizan dos propiedades para colocar los ítems en el grid, `grid-template-areas` y `grid-area`, que veremos un poco más adelante.

- GRID. Propiedades de alineación (I).
- GRID. Propiedades de alineación (II).

4.2. Alineaciones específicas

Las cuatro propiedades vistas en el apartado anterior afectan a todos los ítems y al grid. En el caso de que queramos que uno de los ítems hijos tenga una distribución diferente al resto podemos aplicar en el elemento hijo la propiedad `justify-self` (en horizontal) o `align-self` (en vertical) sobrescribiendo su distribución su general.

Propiedad	Descripción
`justify-self`	Alinea a un ítem concreto en el **eje horizontal**.
`align-self`	Alinea a un ítem concreto en el **eje vertical**.

Estas propiedades funcionan exactamente igual que sus análogas `justify-items` o `align-items` y tienen los mismos valores, pero en lugar de indicarse en el elemento contenedor se hace sobre un ítem hijo y repercute sólo en dicho elemento.

4.3. Atajos: `place-items`, `place-content` y `place-self`

Si vamos a crear estructuras grid donde utilicemos los pares de propiedades `justify-items` y `align-items` por un lado, `justify-content` y `align-content` por otro, e incluso `justify-self` y `align-self` por otro, podemos utilizar las siguientes propiedades de atajo:

Propiedad	Valores posibles	Significado
`place-items`	`<align-items> <justify-items>`	Atajo para `*-items`
`place-content`	`<align-content> <justify-content>`	Atajo para `*-content`
`place-self`	`<align-self> <justify-self>`	Atajo para `*-self`

5. Orden de los elementos: `order`

La propiedad `order` funciona exactamente igual que en flex. Mediante esta propiedad podemos modificar y establecer el orden de aparición de elementos mediante números que actuarán como *pesos* de los elementos:

Propiedad	Valores posibles	Significado
`order`	`Número`. 0 por defecto	Establecer el orden de aparición del ítem.

Echa un vistazo al siguiente ejemplo, disponible en codepen:

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
  }

  .container div {
    border: 1px solid black;
    padding: 15px;
    text-align: center;
    background: gold;
    color: #000;
  }

  .container :nth-of-type(1) { order: 4; }
  .container :nth-of-type(2) { order: 3; }
  .container :nth-of-type(3) { order: 2; }
  .container :nth-of-type(4) { order: 1; }
</style>

<div class="container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
  <div>4</div>
</div>
```

El orden de aparición de los ítems ha sido alterado por la propiedad `order`.

6. Grid por áreas

Hasta ahora hemos creado cuadrículas grid donde definimos sus filas y sus columnas a través de las propiedades `grid-template-rows` y `grid-template-columns`. Sin embargo, esta no es la única forma de definir cuadrículas. Si necesitamos un poco más de flexibilidad a la hora de definir un grid, podemos utilizar una funcionalidad denominada *grid por áreas*, que **permite definir con texto la ubicación y forma que van a tener las celdas de un grid**.

Esta manera de definir un grid no excluye utilizar filas y columnas. Ambas maneras pueden trabajar conjuntamente o por separado, según interese.

6.1. `grid-template-areas` y `grid-area`

Utilizaremos la propiedad `grid-template-areas` en nuestro contenedor padre para especificar el orden de las áreas en la cuadrícula. Posteriormente, en cada ítem hijo, utilizamos la propiedad `grid-area` para indicar el nombre del área del que se trata y que el navegador pueda identificarlas.

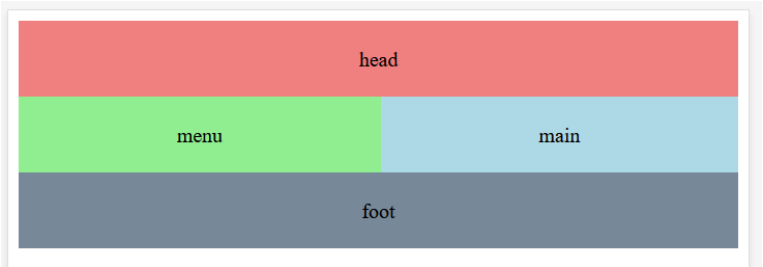
Propiedad	Valores posibles	Significado
`grid-template-areas`	`none | texto`	Indica la disposición de las áreas en el grid. Cada texto entre comillas simboliza una fila.
`grid-area`	`area`	Indica el nombre del área. Se usa sobre ítems hijos del grid. No lleva comillas.

Veamos un ejemplo, también disponible en codepen.

```
<div class="container">
  <div class="item item-1">head</div>
  <div class="item item-2">menu</div>
  <div class="item item-3">main</div>
  <div class="item item-4">foot</div>
</div>

<style>
  .container {
    display: grid;
    grid-template-areas:
      "head head"
      "menu main"
      "foot foot";
  }

  .item {
    padding: 20px;
    text-align: center;
  }
```

```
  .item-1 {
    grid-area: head;
    background: lightcoral;
  }

  .item-2 {
    grid-area: menu;
    background: lightgreen;
  }

  .item-3 {
    grid-area: main;
    background: lightblue;
  }

  .item-4 {
    grid-area: foot;
    background: lightslategray;
  }
</style>
```

Lo que da como resultado:



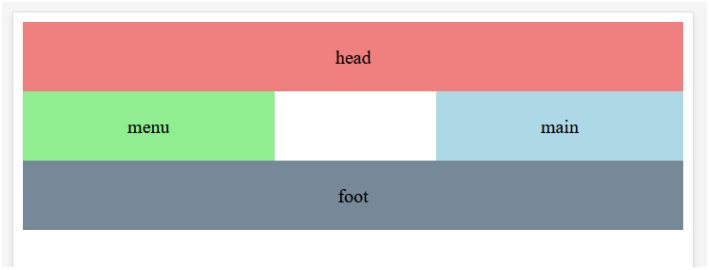
- El ítem 1 sería nuestra cabecera (*head*), que ocuparía la primera fila (toda la parte superior).
- El ítem 2 sería nuestro menú lateral (*menu*), que ocuparía el área izquierda del grid (debajo de la cabecera).
- El ítem 3 sería nuestro contenido (*main*), que ocuparía el área derecha del grid (debajo de la cabecera).
- El ítem 4 sería nuestro pie de cuadrícula (*foot*), que ocuparía la última fila (área inferior del grid).

Si queremos **representar un área, pero sin darle ningún nombre en especial** podemos utilizar el **carácter punto** (‘.’). Siguiendo con el ejemplo anterior,

```
.container {
  display: grid;
  grid-template-areas:
    "head head head"
    "menu . main"
    "foot foot foot";
}
```

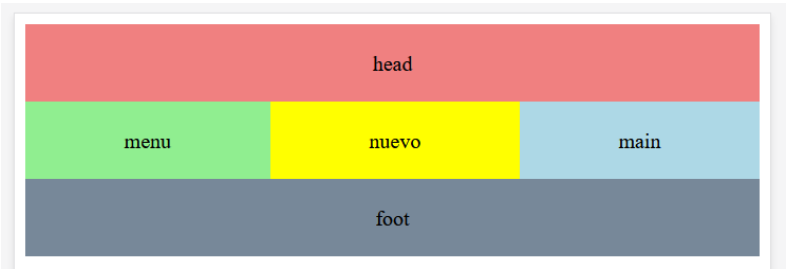
Fíjate que al escribir un punto para introducir un área entre `menú` y `main` esa fila pasa a tener 3 columnas, no dos. Por tanto, debemos ampliar también el resto de filas.

Hemos transformado el grid en 3 filas y 3 columnas. Se ve de la siguiente manera:



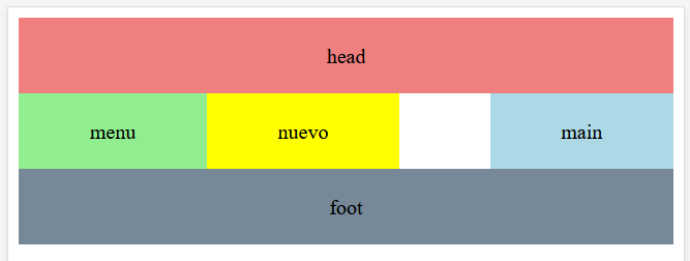
Vemos que se ha creado un hueco en el medio, justo donde ubicamos el punto. Si ahora añadimos otro ítem sin indicar área se situará en ese espacio vacío:

```
<div class="item" style="background-color: yellow">nuevo</div>
```



Si hay más de un hueco libre, el nuevo ítem ocupará el primero disponible:

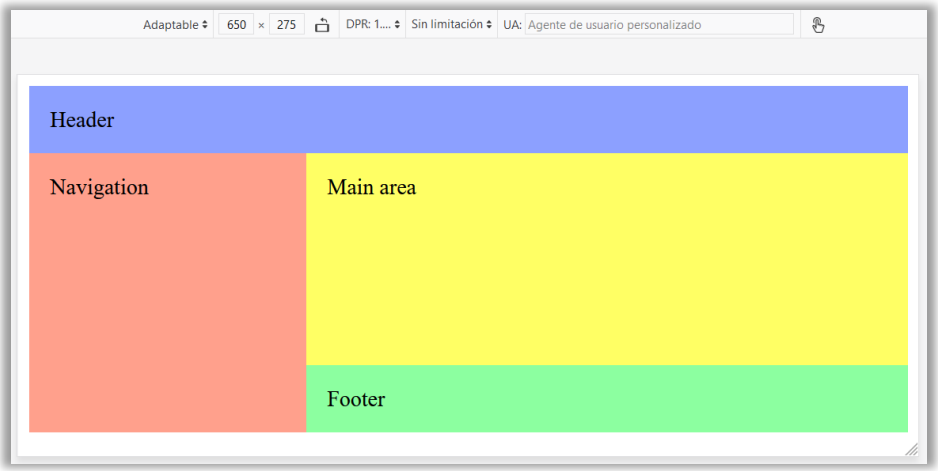
```
.container {
  display: grid;
  grid-template-areas:
    "head head head head"
    "menu . . main"
    "foot foot foot foot";
}
```



Por supuesto, podemos complementar `grid-template-areas` con `grid-template-rows` y `grid-template-columns`.

Por ejemplo, dado el siguiente HTML y CSS, ¿qué valores tendríamos que dar a esas 3 propiedades para obtener un resultado como el de la imagen? El código está disponible en codepen.

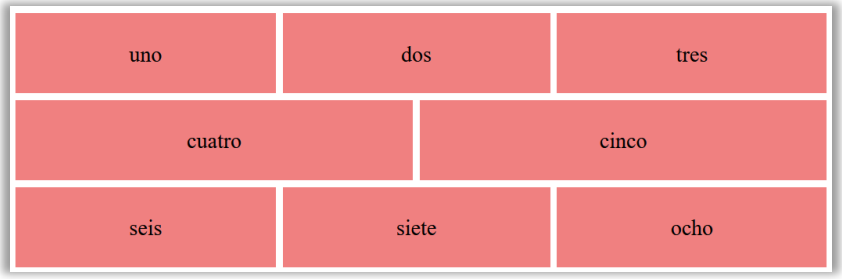
```
<section id="page">
  <header>Header</header>
  <nav>Navigation</nav>
  <main>Main area</main>
  <footer>Footer</footer>
</section>

<style>
  #page {
    display: grid;
    height: 250px;
    grid-template-areas: ???;
    grid-template-rows: ???;
    grid-template-columns: ???;
  }

  #page>* {
    padding: 15px;
  }

  #page>header {
    grid-area: head;
    background-color: #8ca0ff;
  }

  #page>nav {
    grid-area: nav;
    background-color: #ffa08c;
  }

  #page>main {
    grid-area: main;
    background-color: #ffff64;
  }

  #page>footer {
    grid-area: footer;
    background-color: #8cffa0;
  }
</style>
```



La solución es muy simple, pero muy ilustrativa también de cómo utilizar estas 3 propiedades al mismo tiempo:

```
#page {
  display: grid;
  height: 250px;
  grid-template-areas:
    "head head"
    "nav  main"
    "nav  footer";
  grid-template-columns: 200px 1fr;
  grid-template-rows: auto 1fr auto;
}
```

Además, con las áreas podemos crear ***grids* asimétricos**, por ejemplo (código disponible en codepen):



El código sería:

```
<div class="container">
  <div class="item">uno</div>
  <div class="item">dos</div>
  <div class="item">tres</div>
  <div class="item">cuatro</div>
  <div class="item">cinco</div>
  <div class="item">seis</div>
  <div class="item">siete</div>
  <div class="item">ocho</div>
```

```
</div>
<style>
  .container {
    display: grid;
    gap: 5px;
    grid-template-areas:
      "uno uno dos dos tres tres"
      "cuatro cuatro cuatro cinco cinco cinco"
      "seis seis siete siete ocho ocho";
  }

  .item {
    padding: 20px;
    text-align: center;
    background: lightcoral;
  }

  .container .item:nth-child(1) {
    grid-area: uno;
  }

  .container .item:nth-child(2) {
    grid-area: dos;
  }

  .container .item:nth-child(3) {
    grid-area: tres;
  }

  .container .item:nth-child(4) {
    grid-area: cuatro;
  }

  .container .item:nth-child(5) {
    grid-area: cinco;
  }

  .container .item:nth-child(6) {
    grid-area: seis;
  }

  .container .item:nth-child(7) {
    grid-area: siete;
  }

  .container .item:nth-child(8) {
    grid-area: ocho;
  }
</style>
```

Imagina que queremos que nuestro layout sea en 2 columnas para móviles (600px máximo de anchura). ¿Qué habría que hacer?

Simplemente habría que añadir una media query que redefiniera nuestro layout:

```
@media (max-width: 600px) {
  .container {
    grid-template-areas:
      "uno dos"
      "tres cuatro"
      "cinco seis"
      "siete ocho";
  }
}
```

Aunque para que fuera realmente perfecto habría que hacerlo *mobile first*, como en el siguiente código (disponible también en codepen).

```
.container {
  display: grid;
  gap: 5px;
  grid-template-areas:
    "uno dos"
    "tres cuatro"
    "cinco seis"
    "siete ocho";
}

...

@media (min-width: 600px) {
  .container {
    grid-template-areas:
      "uno uno dos dos tres tres"
      "cuatro cuatro cuatro cinco cinco cinco"
      "seis seis siete siete ocho ocho";
  }
}
```

Puedes ver un ejemplo interactivo de la propiedad `grid-area` en este siguiente codepen. Esta propiedad sirve también como atajo para otras 4 propiedades que veremos más adelante.

6.2. Atajo: `grid-template`

La propiedad `grid-template` sirve de atajo para definir `grid-template-rows`, `grid-template-columns` y `grid-template-areas`.

Se puede expresar de varias maneras:

1. Para definir `grid-template-rows` y `grid-template-columns`:

```
grid-template: <grid-template-rows> / <grid-template-columns>
```

Por ejemplo:

```
grid-template: 200px 1fr / 50px 1fr;
```

2. Para definir `grid-template-areas`. Por ejemplo:

```
grid-template:
  "a a a"
  "b b b";
```

3. Para definir las 3 propiedades a la vez. Para poner un ejemplo de esta situación utilizaremos el ejemplo del apartado anterior, donde teníamos:

```
#page {
  display: grid;
  height: 250px;
  grid-template-areas:
    "head head"
    "nav  main"
    "nav  foot";
  grid-template-rows: auto 1fr auto;
  grid-template-columns: 200px 1fr;
}
```

Ahora podemos unificar esas 3 propiedades de la siguiente manera:

```
#page {
  display: grid;
  height: 250px;
  grid-template:
    "head head" auto
    "nav main" 1fr
    "nav foot" auto
    / 200px 1fr;
}
```

Donde las palabras `auto` y `1fr` al lado de la definición de las áreas indican el tamaño de las filas y la última línea, `/ 200px 1fr` indica el tamaño de las columnas.

Podemos prescindir de las definiciones de los tamaños de filas, columnas o ambas, por lo que esto sería perfectamente posible:

```
grid-template:
  "head head"
  "nav main"
  "nav foot"
  / 200px 1fr;
```

Exactamente igual de posible que esto otro:

```
grid-template:
  "head head" auto
  "nav main" 1fr
  "nav foot" auto;
```

Es importante tener en cuenta que `grid-template` es una propiedad abreviada, lo que significa que puede sobrescribir el valor de cualquiera de las tres propiedades individuales si se especifica en la misma declaración. Por lo tanto, es recomendable utilizarla con cuidado para evitar sobrescribir valores que desees mantener.

7. Celdas irregulares

Mediante propiedades como `grid-template-rows`, `grid-template-columns` y/o `grid-template-areas` podemos definir cómo será una cuadrícula desde su elemento contenedor padre, creando un grid de forma flexible, sencilla y práctica.

Sin embargo, las cuadrículas que podemos definir tienen ciertos límites, sobre todo cuando queremos que una celda de la cuadrícula tenga una distribución concreta que haga la cuadrícula irregular. En estos casos entran en juego las siguientes **propiedades**, que **se utilizan en los elementos hijos de la cuadrícula**.

Propiedad	Significado
`grid-column-start`	Indica en que columna empezará el ítem de la cuadrícula.
`grid-column-end`	Indica en que columna terminará el ítem de la cuadrícula.
`grid-row-start`	Indica en que fila empezará el ítem de la cuadrícula
`grid-row-end`	Indica en que fila terminará el ítem de la cuadrícula

Tenemos 2 pares de propiedades, un par con el prefijo `grid-column-*` y otro par con el prefijo `grid-row-*`. Luego, dentro de ambas, tenemos un sufijo `-start` para indicar donde empieza y otra con un sufijo `-end` para indicar donde termina.

Con estas propiedades estableceremos casos particulares donde a un elemento hijo le asignaremos la parte de la cuadrícula donde debe empezar y terminar, creando un caso particular sobre el resto.

7.1. Valores posibles

Valor	Significado
`auto`	No se indica ningún comportamiento particular. Se queda igual. Valor por defecto.
`[número]`	Indica la línea específica, es decir, la separación de columnas o filas.
`span [número]`	Indica hasta cuántas líneas debe llegar.
`[nombre-linea] [nombre]`	Ídem al anterior, pero con líneas nombradas.
`span [nombre-linea] [nombre]`	Ídem al anterior, indicando nombre de línea hasta donde llegar.
`[nombre-linea] [nombre] [número] número`	Ídem al anterior, pero busca el nombre que aparece por *[numero]* vez.

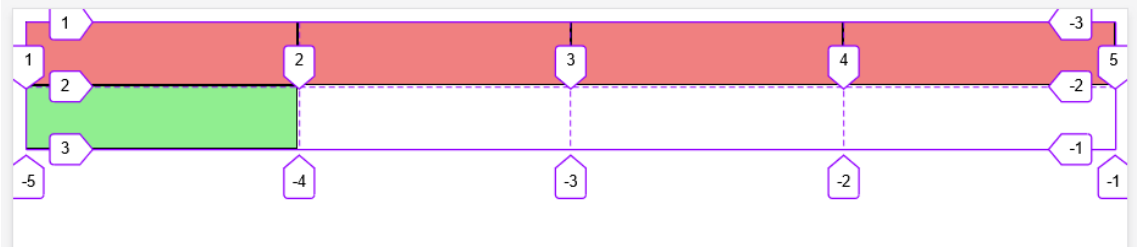
De estas 6 posibles combinaciones de valores, de momento vamos a centrarnos en los 3 primeros.

- `auto`. El valor `auto` (por defecto) no cambia en nada nuestra celda.
- `[número]`. Poniendo un número hacemos referencia a la línea que divide las celdas del grid. Por ejemplo, si tenemos una fila con 4 columnas, ten en cuenta que tendríamos 5 líneas.
- `span [número]`. Si indicamos la palabra `span` antes del número, hacemos referencia al número de líneas que deberemos alargar la celda.

Imagina la siguiente situación:

```
<div class="container">
  <div class="item"></div>
  <div class="item"></div>
  <div class="item"></div>
  <div class="item"></div>
  <div class="item mi-item"></div>
</div>

<style>
  .container {
    display: grid;
    grid-template-rows: repeat(2, auto);
    grid-template-columns: repeat(4, auto);
  }

  .item {
    padding: 20px;
    text-align: center;
    background: lightcoral;
    border: 1px solid black;
  }

  .mi-item {
    background: lightgreen;
  }
</style>
```

El ítem número 5, con clase mi-ítem, se sitúa en la quinta posición del grid, a continuación de sus 4 hermanos.



Pero podemos modificar esto si añadimos:

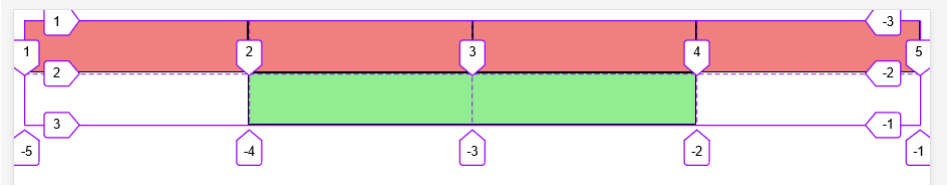
```
.mi-item {
  grid-column-start: 2;
  grid-column-end: 4;
}
```

O bien:

```
.mi-item {
  grid-column-start: 2;
  grid-column-end: span 2;
}
```

Ambos son equivalentes. En el primero el ítem comienza en la línea 2 y termina en la línea 4, por lo que abarca dos celdas. En el segundo caso, el ítem comienza en la columna 2 y se alarga 2 celdas, ocupando la 2 y la 3, por lo que abarca las mismas dos celdas que en el caso anterior.

Con el inspector de Firefox se aprecia claramente dónde está y cómo se ubica el ítem en ambos casos:



7.2. Ejemplo de *grid-column-**

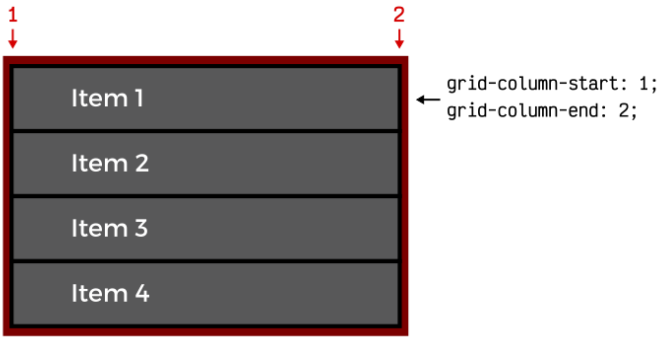
Vamos a plantear el siguiente ejemplo (codepen), donde vamos a definir un grid muy sencillo de 4 elementos. Le aplicamos también algo de estilo visual:

```
<div class="container">
  <div class="item item-1">Item 1</div>
  <div class="item item-2">Item 2</div>
  <div class="item item-3">Item 3</div>
  <div class="item item-4">Item 4</div>
</div>

<style>
  .container {
    display: inline-grid;
    border: 5px solid red;
  }

  .item {
    background: grey;
    color: white;
    padding: 1em;
    border: 5px solid black;
    min-width: 200px;
  }
</style>
```

De esta forma, tenemos una cuadrícula de 4 elementos colocada en vertical. Vamos a colocar unos estilos a la celda 1, modificando el CSS de la clase `item-1`. En realidad, visualmente no va a existir ningún cambio, pero veremos mejor el código y nos anticiparemos a los cambios que vamos a hacer a continuación:

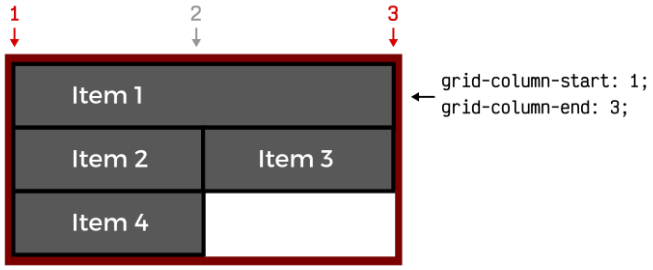
```
.item-1 {
  grid-column-start: 1;
  grid-column-end: 2; /* Mismo efecto si colocamos 1 */
}
```

Si colocamos este fragmento de CSS en el ejemplo anterior, no cambia absolutamente nada. Esto ocurre porque estamos indicando que el item-1 ocupe la ubicación en columna desde la línea 1 hasta la línea 2, que es la posición que ya tiene:



Vamos a modificar la propiedad `grid-column-end` para que apunte a 3 en lugar de a 2. Ahora la celda ocupará la primera y segunda celda, modificando la estructura de la cuadrícula grid, que pasará de una cuadrícula 1x4 a ser una cuadrícula irregular 2x3:

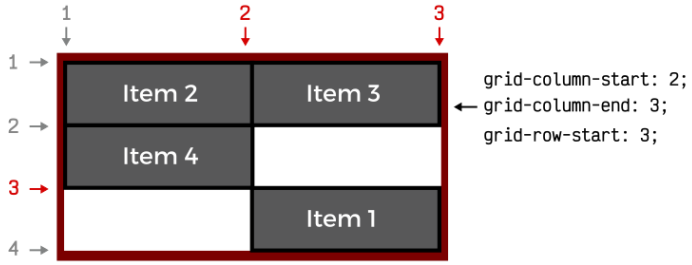
```
grid-column-end: 3;
```



Ahora vamos a indicar con la propiedad `grid-column-start` que comenzaremos en la segunda línea y con `grid-column-end` que acabaremos en la tercera. De la misma forma, utilizaremos la propiedad `grid-row-start` para indicar donde comenzaremos respecto a las filas, y le especificaremos la fila 3:

```
.item-1 {
  grid-column-start: 2;
  grid-column-end: 3;
  grid-row-start: 3;
}
```

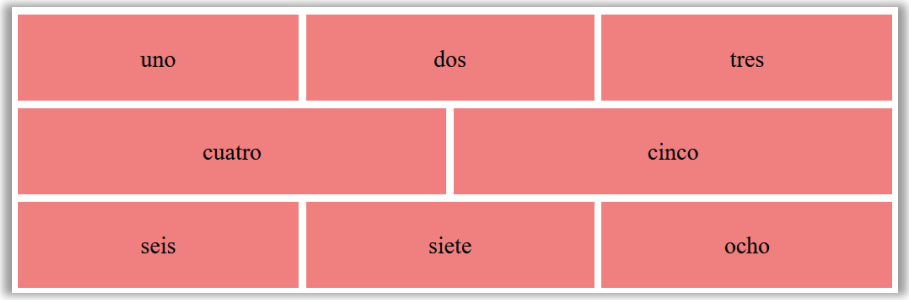
Observa que la celda pasa a moverse a la segunda columna (línea 2 y 3), y a moverse a la tercera fila (línea 3). De esta forma podemos crear fácilmente cuadrículas con celdas irregulares de forma muy flexible y potente:



Recuerda que también es posible utilizar la palabra clave `span` seguida de un número de línea, que indica la cantidad de líneas que abarcará. Por ejemplo, el último ejemplo es equivalente al siguiente:

```
.item-1 {
  background: red;
  grid-column-start: 2;
  grid-column-end: span 1;
  grid-row-start: 3;
}
```

Otro ejemplo, podemos rehacer el ejemplo que hicimos con grid por áreas:



Tienes el código en el siguiente codepen.

7.3. Atajos: `grid-row` y `grid-column`

Con las propiedades `grid-row` y `grid-column` podemos escribir de forma abreviada las 4 propiedades anteriores:

Propiedad	Descripción
`grid-row`	Atajo para `grid-row-start` y `grid-row-end`
`grid-column`	Atajo para `grid-column-start` y `grid-column-end`

Así, el siguiente código:

```
grid-column-start: 2;
grid-column-end: span 1;
grid-row-start: 3;
```

Es equivalente a:

```
grid-column: 2 / span 1;
grid-row: 3;
```

Observa que con las propiedades `grid-column` y `grid-row` separamos con una barra '/' cada uno de los valores de cada propiedad individual. Si no necesitamos utilizar ambas, simplemente ponemos el valor del primero sin utilizar '/'.

7.4. Atajo: `grid-area`

Por si no nos resulta cómodo aún trabajar con las propiedades de atajo `grid-column` y `grid-row`, podemos utilizar la propiedad `grid-area` que vimos en el apartado de grid por áreas. Esta propiedad permite resumir las cuatro propiedades `grid-column-start`, `grid-column-end`, `grid-row-start` y `grid-row-end` en una sola.

Propiedad	Descripción
`grid-area`	Atajo para `grid-row` y `grid-column`

Donde el orden es el siguiente:

```
grid-area: <row-start> / <column-start> / <row-end> / <column-end>
```

El ejemplo del apartado anterior sería el siguiente

```
grid-area: auto / 2 / span 4 / span 1;
```

Puedes ver un ejemplo interactivo en el siguiente codepen.

8. Líneas con nombre

Hasta ahora hemos hablado de celdas o casillas de una cuadrícula. Sin embargo, también podemos hablar de líneas, e incluso ponerles nombres y luego hacer referencia a ellas en ciertas propiedades.

Las líneas en un grid son aquellas divisiones o líneas separadoras de cada celda que existen tanto en horizontal como en vertical. Cuando hablamos de `linenames` (o *nombres de línea*) hacemos referencia a las líneas separadoras de nuestra cuadrícula, a las cuales se les ha dado un nombre.

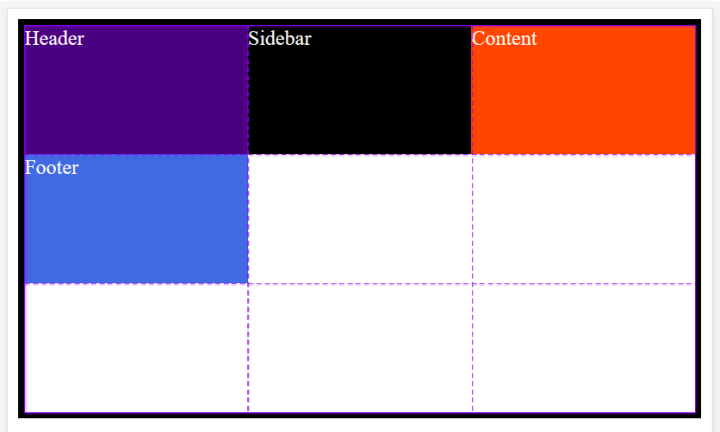
Estas líneas se pueden definir extendiendo la sintaxis de las propiedades `grid-template-columns` y `grid-template-rows`, que vimos al principio:

Propiedad	Valores posibles
`grid-template-columns`	`[linea1] col1 [linea2] col2 ... [últimalinea]`
`grid-template-rows`	`[linea1] fila1 [linea2] fila2 ... [últimalinea]`

Observa el siguiente ejemplo, también disponible en codepen:

```
<div class="container">
  <div class="header">Header</div>
  <div class="sidebar">Sidebar</div>
  <div class="content">Content</div>
  <div class="footer">Footer</div>
</div>

<style>
  .container {
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
    grid-template-rows: 1fr 1fr 1fr;
    min-height: 300px;
    color: white;
    border: 5px solid black;
  }

  .header { background: indigo; }
  .sidebar { background: black; }
  .content { background: orangered; }
  .footer { background: royalblue; }
</style>
```

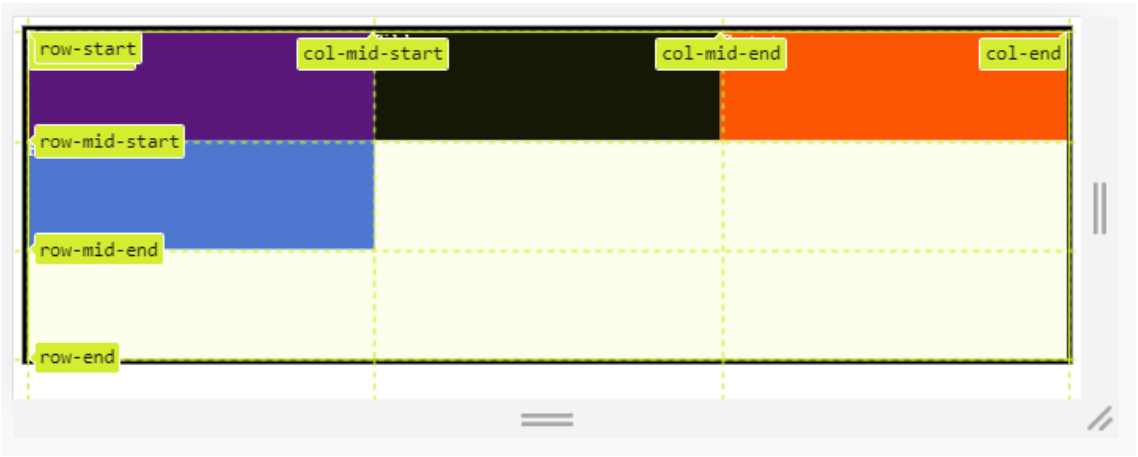
Hemos creado un grid de 3x3 y las hemos distribuido de forma equitativa con 3 valores `1fr` tanto en columnas como en filas. Además, le hemos dado un alto mínimo de 300px para tener un espacio mínimo y un color de texto blanco. Ten en cuenta también que en el HTML sólo tenemos 4 elementos hijos por lo que no rellenaremos completamente el grid. Quedaría algo así:



Ya tenemos una estructura definida, pero ahora vamos a añadir unos nombres a las líneas divisorias de cada celda. Para ello, vamos a añadir los nombres de cada línea entre corchetes en las propiedades `grid-template-columns` y `grid-template-rows`. Así:

```
.container {
  display: grid;
  grid-template-columns: [col-start] 1fr [col-mid-start] 1fr [col-mid-end] 1fr
[col-end];
  grid-template-rows: [row-start] 1fr [row-mid-start] 1fr [row-mid-end] 1fr
[row-end];
  min-height: 300px;
  color: white;
  border: 5px solid black;
}
```

Observa que en horizontal (columnas) tenemos 4 líneas separadoras, ya que tenemos 3 columnas. Al igual que en vertical (filas), que también tenemos 4 líneas separadoras.



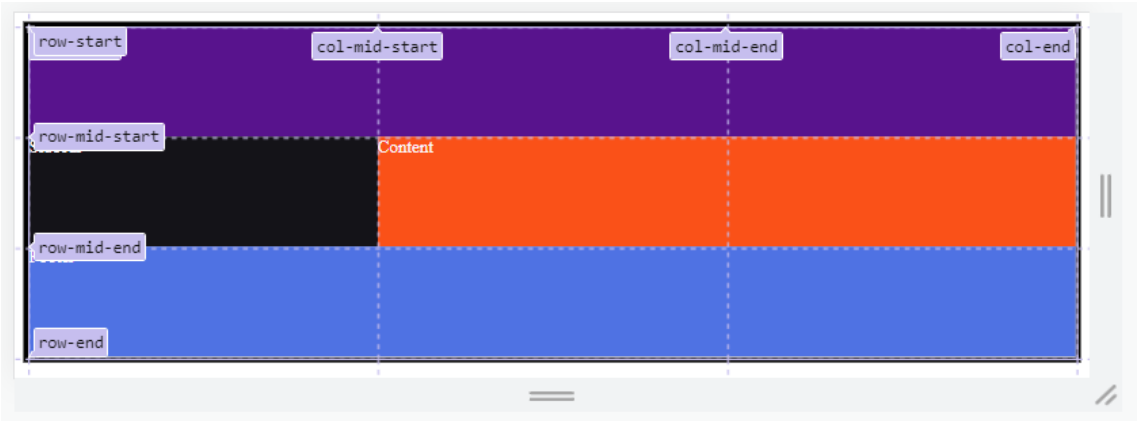
El inspector de Firefox (versión 134.0) no permite mostrar los nombres de las líneas, pero el de Chrome sí (v 131.0), que es de donde está sacada esta imagen.

Teniendo ya líneas con nombres, sólo nos quedaría delimitar qué zonas del grid queremos que ocupe cada uno de nuestros elementos `<div>`. Para ello, vamos a utilizar las propiedades `grid-column-start`, `grid-column-end` y `grid-row-start`, `grid-row-end` que aprendimos en el apartado anterior.

Añadimos el siguiente código a los elementos hijos:

```css
.header {
  background: indigo;
  /* Ocupa toda la fila inicial */
  grid-column-start: col-start;
  grid-column-end: col-end;
}

.sidebar {
  background: black;
  /* Realmente no hace falta, porque ya está colocada en este lugar */
  grid-row-start: row-mid-start;
  grid-row-end: row-mid-end;
}

.content {
  background: orangered;
  /* Ocupa las dos celdas de la derecha de la fila central */
  grid-column-start: col-mid-start;
  grid-column-end: col-end;
  grid-row-start: row-mid-start;
  grid-row-end: row-mid-end;
}

.footer {
  background: royalblue;
  /* Ocupa toda la fila final */
  grid-column-start: col-start;
  grid-column-end: col-end;
  grid-row-start: row-mid-end;
  grid-row-end: row-end;
}
```

Por lo que, con estos cambios, nuestra estructura grid quedaría así:



Por supuesto, podríamos haber llamado a nuestras líneas `col-1`, `col-2`, etc., pero la especificación recomienda añadir el sufijo `-start` y `-end` a los nombres de las líneas.

8.1. Usando los atajos `grid-column` y `grid-row`

Vamos a modificar el ejemplo anterior utilizando las propiedades de `grid-column` y `grid-row`, que nos evitará tener que escribir tanto código. Recuerda que la sintaxis de `grid-column` es `grid-column-start` / `grid-column-end` y `grid-row` es `<grid-row-start> / <grid-row-end>`.

```
.header {
  background: indigo;
  grid-column: col-start / col-end;
}

.sidebar {
  background: black;
  grid-row: row-mid-start / row-mid-end;
  /* Redundante */
}

.content {
  background: orangered;
  grid-column: col-mid-start / col-end;
  grid-row: row-mid-start / row-mid-end;
  /* Redundante */
}

.footer {
  background: royalblue;
  grid-column: col-start / col-end;
  grid-row: row-mid-end / row-end;
  /* Redundante */
}
```

Hemos colocado las propiedades `grid-row` para que se entienda cómo se utilizaría, sin embargo, en este ejemplo concreto, las propiedades `grid-row` son redundantes y no hacen falta, ya que la celda está naturalmente posicionada en esa fila y no hace falta alterarla.

8.2. Usando el atajo `grid-area`

Como vimos anteriormente, la propiedad `grid-area` sirve de atajo para las propiedades `grid-column` y `grid-row`, por lo que se podría utilizar para resumir aún más el resultado anterior.

Vamos a eliminar los valores redundantes y a convertir el resto a la propiedad `grid-area`. Recuerda que el formato de `grid-area` era:

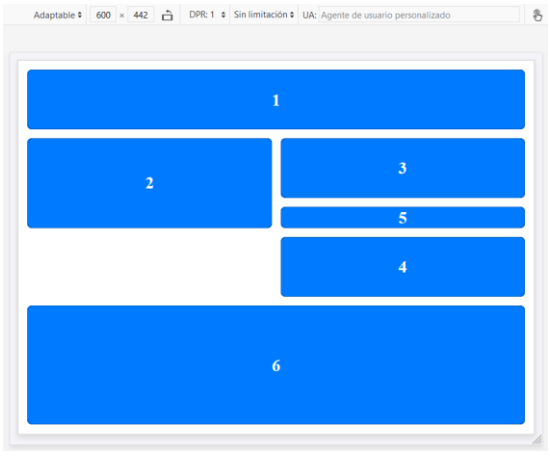
```
<grid-row-start> / <grid-column-start> / <grid-row-end> / <grid-column-end>
```

Los valores redundantes podemos indicarlos mediante la palabra `auto`.

```
.header {
  background: indigo;
  grid-area: auto / col-start / auto / col-end;
  /* grid-column: col-start / col-end; */
}

.sidebar {
  background: black;
  grid-area: row-mid-start / auto / row-mid-end / auto;
  /* grid-row: row-mid-start / row-mid-end; */
}

.content {
  background: orangered;
  grid-area: row-mid-start / col-mid-start / row-mid-end / col-end;
  /* grid-column: col-mid-start / col-end;
  grid-row: row-mid-start / row-mid-end; */
}

.footer {
  background: royalblue;
  grid-area: row-mid-end / col-start / row-end / col-end;
  /* grid-column: col-start / col-end;
  grid-row: row-mid-end / row-end; */
}
```

Hagamos un pequeño ejercicio. Intenta diseñar un layout como el siguiente utilizando líneas con nombre. El grid tiene una altura de 400px, una anchura máxima de 800px y el breakpoint está en 600px. Tienes la solución en codepen.

<600px



>600px

9. Tamaños de filas y columnas indefinidas

9.1. Propiedades `grid-auto-rows` y `grid-auto-columns`

Estas dos propiedades permiten definir el tamaño de las filas o columnas que no estén definidas aún.

Ya conocemos las propiedades `grid-template-rows` y `grid-template-columns`. Con ellas establecemos el número y tamaño de filas y columnas de manera explícita. Sin embargo, puede haber situaciones en las que no sepamos exactamente el número de filas o columnas que necesitaremos porque no sepamos cuántos ítems tendremos que colocar. Ahí entran en juego `grid-auto-rows` y `grid-auto-columns`.

Los posibles valores de estas propiedades son:

Propiedad	Posibles valores	Descripción

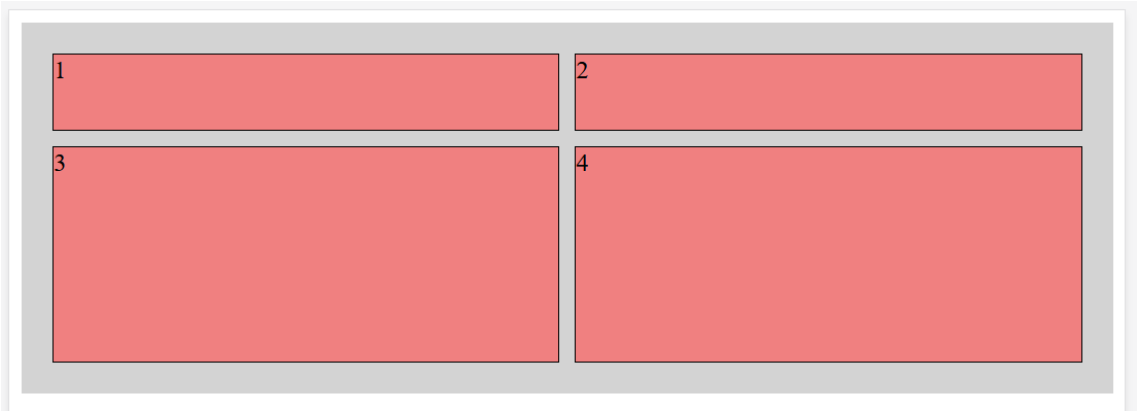
`grid-auto-rows`	`auto`
	`fit-content()`
	`minmax()`
	`max-content`
	`min-content`
	`%`
	`[length]`
		Establece el tamaño de filas nuevas.

`grid-auto-columns`		Establece el tamaño de columnas nuevas.

Imagina el siguiente escenario, donde tenemos definido un grid de 1 fila y 2 columnas, pero tenemos que colocar 4 ítems.

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
</div>

<style>
  .container {
    background-color: lightgray;
    display: grid;
    grid-template-rows: 50px;
    grid-template-columns: 1fr 1fr;
    height: 200px;
    gap: 10px;
    padding: 20px;
  }
  .item {
    background-color: lightcoral;
    border: 1px solid black;
  }
</style>
```

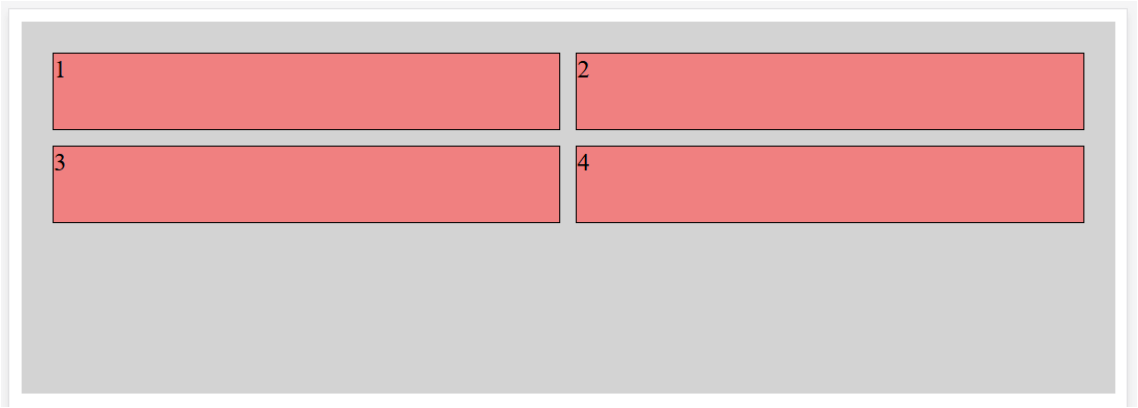
En esta situación observamos que la primera fila, la única que está definida, ocupa los 50px indicados. La segunda ocupa el resto de la altura del grid porque el valor por defecto para las nuevas filas es `auto`.



Ahora indicaremos que las nuevas filas tengan también 50px de alto:

```
grid-auto-rows: 50px;
```

Ahora la segunda fila es de esos 50px que hemos indicado. El funcionamiento es exactamente el mismo con columnas.



Podemos incluso eliminar la línea:

```
grid-template-rows: 50px;
```

Y dejar sólo:

```
grid-auto-rows: 50px;
```

Dado que ahora todas las filas van a ocupar 50px.

10. Rellenando huecos: `grid-auto-flow`

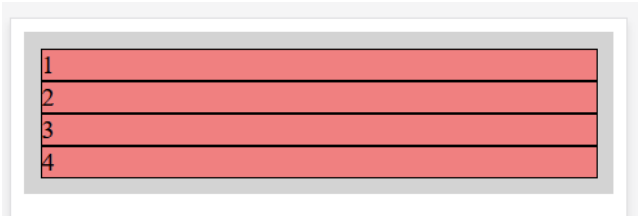
La propiedad `grid-auto-flow` determina cómo se colocan los ítems en un grid cuando no se ha especificado explícitamente dónde debe colocarse cada uno.

Valor	Significado
`row`	Sitúa los ítems en filas consecutivas. Por defecto.
`column`	Sitúa los ítems en columnas consecutivas.
`dense`	Sitúa los ítems de forma que se intenta rellenar los huecos que aparecen antes en el grid. Si existen ítems más pequeños más tarde en el código HTML, esto puede hacer que los algunos ítems aparezcan desordenados al rellenar los huecos que dejan los ítems más grandes.
`row dense`	Sitúa los ítems para rellenar primero las filas.
`column dense`	Sitúa los ítems para rellenar primero las columnas.

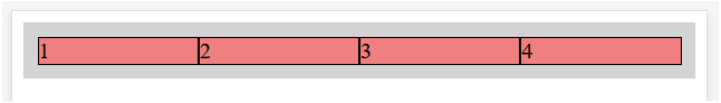
La diferencia entre los valores `dense` y `row dense` es muy sutil:

- El valor `dense` hace que los elementos se empaqueten de manera óptima para llenar los espacios vacíos en el grid en lugar de simplemente agregarlos en orden.
- El valor `row dense` es una combinación de `row` y `dense`. Es similar a `dense`, pero específicamente los elementos se empaquetarán en filas. Si no se especifica `row` en `grid-auto-flow: row dense`, los elementos se empaquetarán tanto en filas como en columnas.

El valor por defecto de `grid-auto-flow` es `row`, lo que indica que los elementos se ubicarán en filas en el orden que están escritos en nuestro código. Si tuviéramos 4 elementos se organizarían del siguiente modo:



Si cambiamos a `grid-auto-flow: column`:



Puedes ver un ejemplo interactivo de esta propiedad en el este codepen.

11. Atajo. La propiedad `grid`

La propiedad `grid` es un ata para establecer las propiedades `grid-template-rows`, `grid-template-columns`, `grid-template-areas`, `grid-auto-rows`, `grid-auto-columns` y `grid-auto-flow` en una sola declaración. Si alguna de estas propiedades no aparece en la declaración, esta tomará el valor por defecto. Su uso no es aconsejable, por su complejidad, hasta no tener un dominio claro de las propiedades implicadas.

Puedes ver el uso de esta propiedad en w3schools o en developer.mozilla.org.

12. Anexo I. Tabla resumen

En la siguiente tabla puedes encontrar todas las propiedades relacionadas con grid con su descripción y un link a la página de la propiedad en w3schools.

Propiedad	Descripción
column-gap	Especifica el espacio entre las columnas.
gap	Atajo para las propiedades `row-gap` y `column-gap`.
grid	Atajo para `grid-template-rows`, `grid-template-columns`, `grid-template-areas`, `grid-auto-rows`, `grid-auto-columns` y `grid-auto-flow`.
grid-area	Especifica un nombre para el elemento de la cuadrícula o es un atajo para las propiedades `grid-row-start`, `grid-column-start`, `grid-row-end` y `grid-column-end`.
grid-auto-columns	Especifica un tamaño de columna por defecto.
grid-auto-flow	Especifica cómo se insertan los elementos colocados automáticamente en la cuadrícula.
grid-auto-rows	Especifica un tamaño de fila por defecto.
grid-column	Atajo para las propiedades `grid-column-start` y `grid-column-end`.
grid-column-end	Especifica dónde terminar el elemento de la cuadrícula.
grid-column-gap	Especifica el tamaño del espacio entre las columnas.
grid-column-start	Especifica dónde empezar el elemento de la cuadrícula.
grid-gap	Atajo para las propiedades `grid-row-gap` y `grid-column-gap`.
grid-row	Atajo para las propiedades `grid-row-start` y `grid-row-end`.
grid-row-end	Especifica dónde terminar el elemento de la cuadrícula.
grid-row-gap	Especifica el tamaño del espacio entre las filas.
grid-row-start	Especifica dónde empezar el elemento de la cuadrícula.
grid-template	Atajo para las propiedades `grid-template-rows`, `grid-template-columns` y `grid-areas`.
grid-template-areas	Especifica cómo mostrar las columnas y las filas, usando elementos de cuadrícula con nombre.
grid-template-columns	Especifica el tamaño y el número de columnas de un diseño de cuadrícula.
grid-template-rows	Especifica el tamaño de las filas en un diseño de cuadrícula.
row-gap	Especifica el espacio entre las filas de la cuadrícula.

13. Webgrafía

- https://www.w3.org/TR/css-grid-1/
- https://lenguajecss.com/css/maquetacion-y-colocacion/grid-css/
- Generador de layouts grid: https://cssgrid-generator.netlify.app/