A Complete Guide to CSS Grid Layout

by Chris House March 4, 2016

Introduction

CSS Grid Layout (aka "Grid"), is a two-dimensional grid-based layout system that aims to do nothing less than completely change the way we design grid-based user interfaces. CSS has always been used to lay out our web pages, but it's never done a very good job of it. First we used tables, then floats, positioning and inline-block, but all of these methods were essentially hacks and left out a lot of important functionality (vertical centering, for instance). Flexbox helped out, but it's intended for simpler one-dimensional layouts, not complex two-dimensional ones (Flexbox and Grid actually work very well together). Grid is the very first CSS module created specifically to solve the layout problems we've all been hacking our way around for as long as we've been making websites.

There are two primary things that inspired me to create this guide. The first is Rachel Andrew's awesome book Get Ready for CSS Grid Layout. It's a thorough, clear introduction to Grid and is the basis of this entire article. I *highly* encourage you to buy it and read it. My other big inspiration is Chris Coyier's A Complete Guide to Flexbox, which has been my go-to resource for everything flexbox. It's helped a ton of people, evident by the fact that it's the top result when you Google "flexbox." You'll notice many similarities between his post and mine, because why not steal from the best?

My intention with this guide is to present the Grid concepts as they exist in the very latest version of the specification. So I won't be covering the out of date IE syntax, and I'll do my best to update this guide regularly as the spec matures.

- Basics and Browser Support

Getting started with Grid is easy. You just define a container element as a grid with `display: grid`, set the column and row sizes with `grid-template-columns` and `grid-template-rows`, and then place its child elements into the grid with `grid-column` and `grid-row`. Similarly to flexbox, the source order of the grid items doesn't matter. Your CSS can place them in any order, which makes it super easy to rearrange your grid with media queries. Imagine defining the layout of your entire page, and then completely rearranging it to accommodate a different screen width all with only a couple lines of CSS. Grid is one of the most powerful CSS modules ever introduced.

An important thing to understand about Grid is that it's not ready to be used in production yet. It's currently a W3C Working Draft and isn't supported correctly in any browsers yet by default. Internet Explorer 10 and 11 support it, but it's an old implementation with an outdated syntax. In order to experiment with Grid today, your best bet is to use Chrome, Opera or Firefox with special flags enabled. In Chrome, navigate to chrome://flags and enable "experimental web platform features". That method also works in Opera (opera://flags). In Firefox, enable the layout.css.grid.enabled flag.

Here's a browser support table which I'll keep up-to-date:

Internet Explorer/Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Chrome for Android
10+ (Old Syntax)	40+ (Behind Flag)	29+ (Behind Flag)	Not Supported	28+ (Behind Flag)	Not Supported	Not Supported

Aside from Microsoft, browser manufacturers appear to be holding off on letting Grid loose in the wild until the spec is fully cooked. This is a good thing, as it means we won't have to worry about learning multiple syntaxes.

It's only a matter of time before you can use Grid in production. But the time to learn it is now.

Important Terminology

Before diving into the concepts of Grid it's important to understand the terminology. Since the terms involved here are all kinda conceptually similar, it's easy to confuse them with one another if you don't first memorize their meanings defined by the Grid specification. But don't worry, there aren't many of them.

Grid Container

The element on which `display: grid` is applied. It's the direct parent of all the grid items. In this example `container` is the grid container.

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

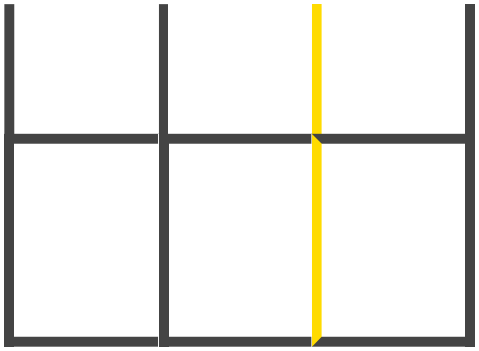
Grid Item

The children (e.g. *direct* descendants) of the grid container. Here the `item` elements are grid items, but `sub-item` isn't.

```HTML
<div class="container">
  <div class="item"></div>
  <div class="item">
        <p class="sub-item"></p>
  </div>
  <div class="item"></div>
</div>
```

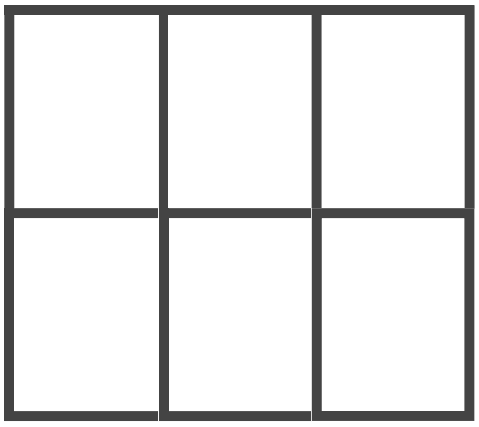
Grid Line

The dividing lines that make up the structure of the grid. They can be either vertical ("column grid lines") or horizontal ("row grid lines") and reside on either side of a row or column. Here the yellow line is an example of a column grid line.



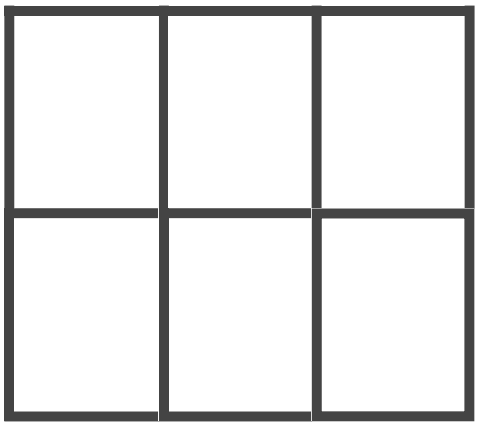
Grid Track

The space between two adjacent grid lines. You can think of them like the columns or rows of the grid. Here's the grid track between the second and third row grid lines.



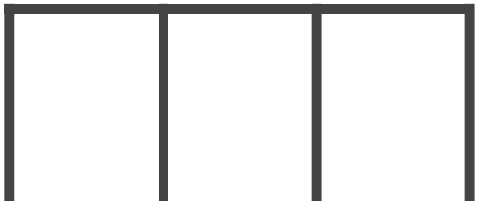
Grid Cell

The space between two adjacent row and two adjacent column grid lines. It's a single "unit" of the grid. Here's the grid cell between row grid lines 1 and 2, and column grid lines 2 and 3.

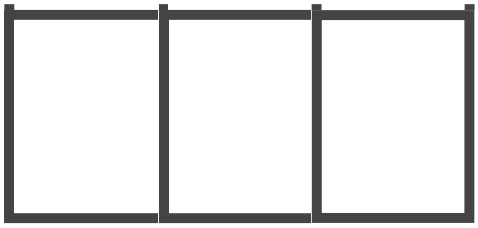


Grid Area

The total space surrounded by four grid lines. A grid area may be



comprised of any number of grid cells. Here's the grid area between row grid lines 1 and 3, and column grid lines 1 and 3.



Grid Properties Table of Contents

Properties for the Grid Container

display

Defines the element as a grid container and establishes a new *grid formatting context* for its contents.

Values:

grid - generates a block-level grid
inline-grid - generates an inline-level grid
subgrid - if your grid container is itself a grid item (i.e. nested grids), you can use this property to indicate that you want the sizes of its rows/columns to be taken from its parent rather than specifying its own.

```
CSS

.container{
  display: grid | inline-grid | subgrid;
}
```

Note: `column`, `float`, `clear`, and `vertical-align` have no effect on a grid container.

[top]

grid-template-columns
grid-template-rows

Defines the columns and rows of the grid with a space-separated list of values. The values represent the track size, and the space between them represents the grid line.

Values:

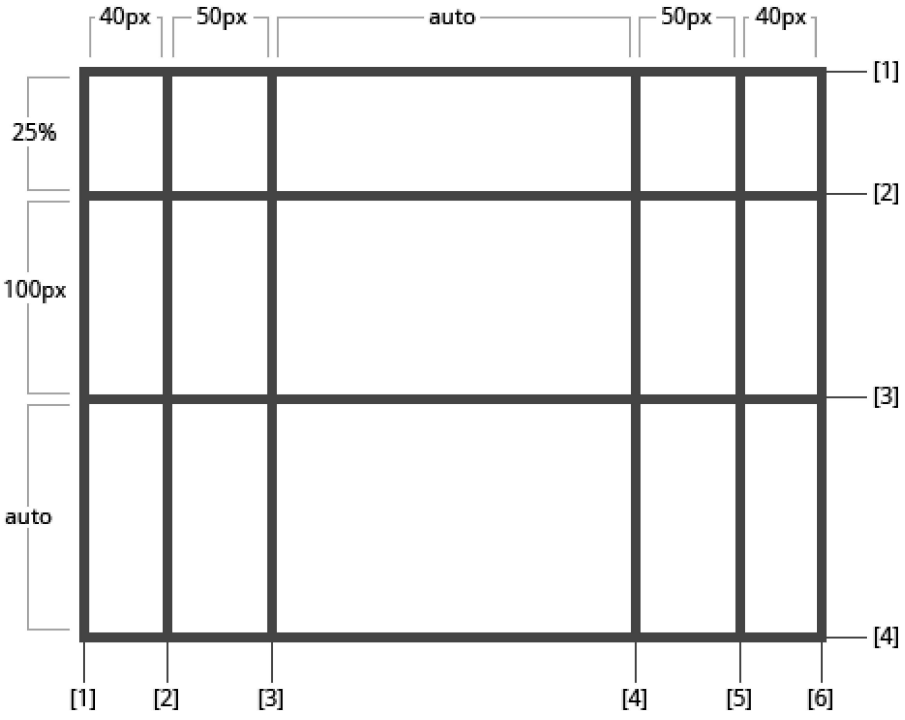
<track-size> - can be a length, a percentage, or a fraction of the free space in the grid (using the `fr` unit)
<line-name> - an arbitrary name of your choosing

```
CSS

.container{
  grid-template-columns: <track-size> ... | <line-name> <track-size> ...;
  grid-template-rows: <track-size> ... | <line-name> <track-size> ...;
}
```

Examples:

When you leave an empty space between the track values, the grid lines are automatically assigned numerical names:

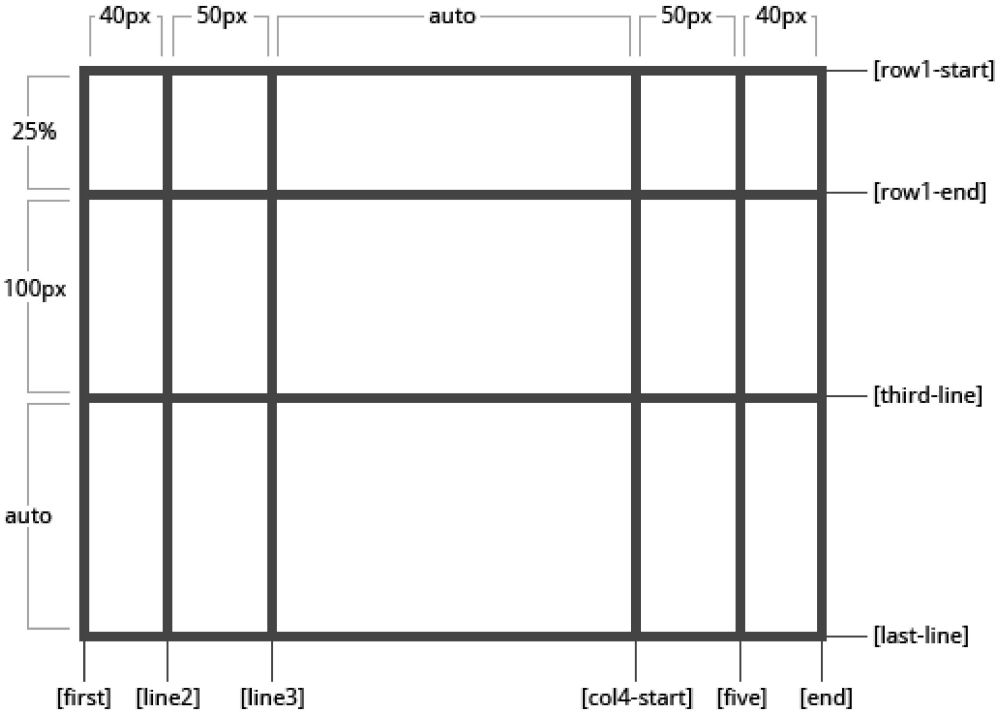
```
CSS

.container{
  grid-template-columns: 40px 50px auto 50px 40px;
  grid-template-rows: 25% 100px auto;
}
```



But you can choose to explicitly name the lines. Note the bracket syntax for the line names:

```
CSS

.container{
  grid-template-columns: [first] 40px [line2] 50px [line3] auto [col4-start
```

```
  grid-template-rows: [row1-start] 25% [row1-end] 100px [third-line] auto [
}
```



Note that a line can have more than one name. For example, here the second line will have two names: row1-end and row2-start:

```
CSS

.container{
  grid-template-rows: [row1-start] 25% [row1-end row2-start] 25% [row2-end]
}
```

If your definition contains repeating parts, you can use the `repeat()` notation to streamline things:

```
CSS

.container{
  grid-template-columns: repeat(3, 20px [col-start]) 5%;
}
```

Which is equivalent to this:

```css
.container{
  grid-template-columns: 20px [col-start] 20px [col-start] 20px [col-start]
}
```

The `fr` unit allows you to set the size of a track as a fraction of the free space of the grid container. For example, this will set each item to one third the width of the grid container:

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

The free space is calculated *after* any non-flexible items. In this example the total amount of free space available to the `fr` units doesn't include the 50px:

```css
.container{
  grid-template-columns: 1fr 50px 1fr 1fr;
}
```

[top]

grid-template-areas

Defines a grid template by referencing the names of the grid areas which are specified with the `grid-area` property. Repeating the name of a grid area causes the content to span those cells. A period signifies an empty cell. The syntax itself provides a visualization of the structure of the grid.

Values:

<grid-area-name> - the name of a grid area specified with `grid-area`

. - a period signifies an empty grid cell

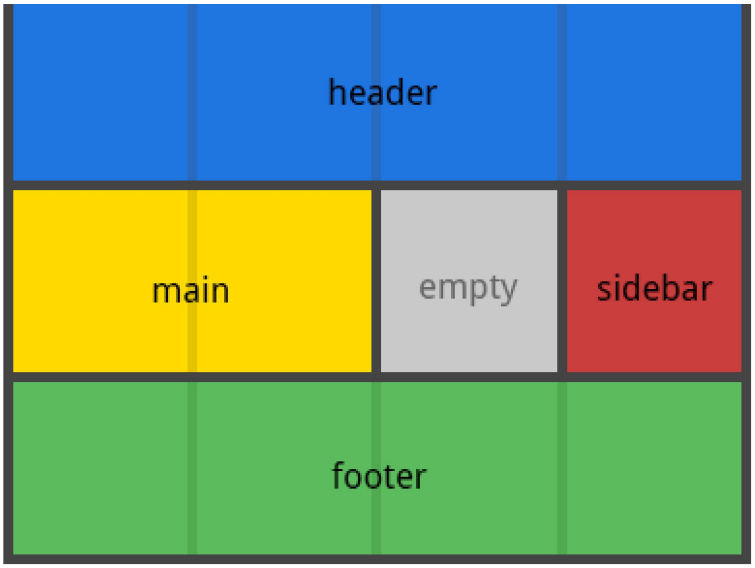
none - no grid areas are defined

```
CSS

.container{
  grid-template-areas: "<grid-area-name> | . | none | ..."
                       "..."
}
```

Example:

```
CSS

.item-a{
  grid-area: header;
}
.item-b{
  grid-area: main;
}
.item-c{
  grid-area: sidebar;
}
.item-d{
  grid-area: footer;
}

.container{
  grid-template-columns: 50px 50px 50px 50px;
  grid-template-rows: auto;
  grid-template-areas: "header header header header"
                       "main main . sidebar"
                       "footer footer footer footer"
}
```

That'll create a grid that's four columns wide by three rows tall. The entire top row will be comprised of the **header** area. The middle row will be comprised of two **main** areas, one empty cell, and one **sidebar** area. The last row is all **footer**.



Each row in your declaration needs to have the same number of cells.

You can use any number of adjacent periods to declare a single empty cell. As long as the periods have no spaces between them they represent a single cell.

Notice that you're not naming lines with this syntax, just areas. When you use this syntax the lines on either end of the areas are actually getting named automatically. If the name of your grid area is ***foo***, the name of the area's starting row line and starting column line will be ***foo*-start**, and the name of its last row line and last column line will be ***foo*-end**. This means that some lines might have multiple names, such as the far left line in the above example, which will have three names: header-start, main-start, and footer-start.

[top]

`grid-column-gap`
`grid-row-gap`

Specifies the size of the grid lines. You can think of it like setting the width of the gutters between the columns/rows.

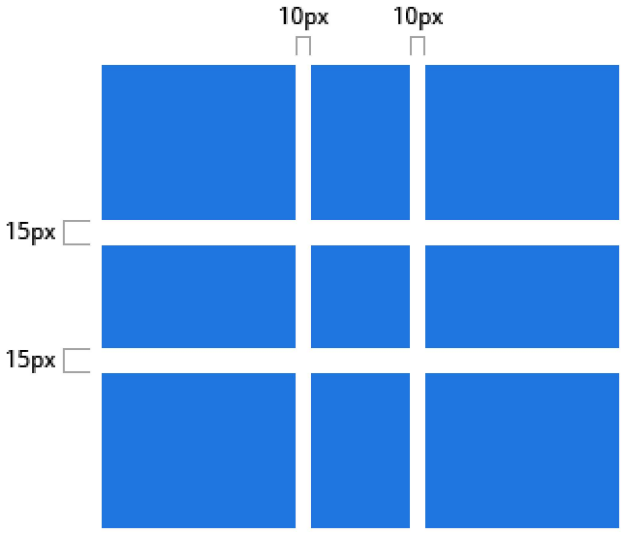
Values:

<line-size> - a length value

```CSS
.container{
  grid-column-gap: <line-size>;
  grid-row-gap: <line-size>;
}
```

Example:

```CSS
.container{
  grid-template-columns: 100px 50px 100px;
  grid-template-rows: 80px auto 80px;
  grid-column-gap: 10px;
  grid-row-gap: 15px;
}
```



The gutters are only created *between* the columns/rows, not on the outer edges.

[top]

grid-gap

A shorthand for `grid-column-gap` + `grid-row-gap`.

Values:

<grid-column-gap> <grid-row-gap> - length values

```
CSS

.container{
  grid-gap: <grid-column-gap> <grid-row-gap>;
}
```

Example:

```
CSS

.container{
  grid-template-columns: 100px 50px 100px;
  grid-template-rows: 80px auto 80px;
  grid-gap: 10px 15px;
}
```

If no `grid-row-gap` is specified, it's set to the same value as `grid-column-gap`

[top]

`justify-items`

Aligns the content inside a grid item along the *column* axis (as opposed to `align-items` which aligns along the *row* axis). This value applies to all grid items inside the container.

Values:

start - aligns the content to the left end of the grid area

end - aligns the content to the right end of the grid area

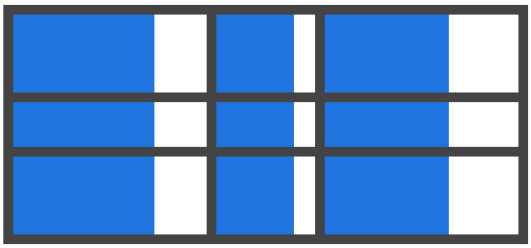
center - aligns the content in the center of the grid area

stretch - fills the whole width of the grid area (this is the default)

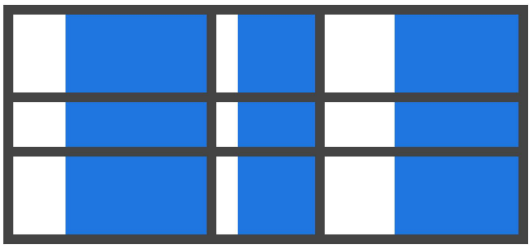
```css
.container{
  justify-items: start | end | center | stretch;
}
```

Examples:

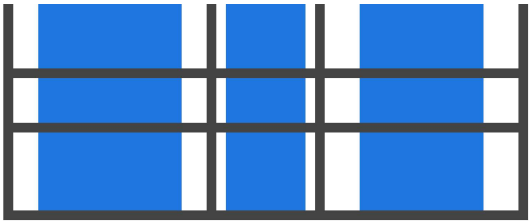
```css
.container{
  justify-items: start;
}
```



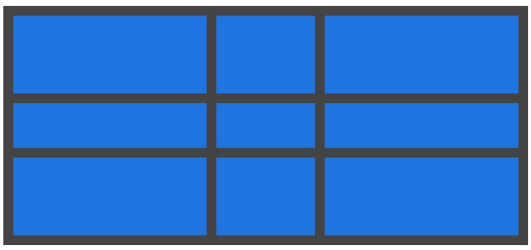
```css
.container{
  justify-items: end;
}
```



```css
.container{
  justify-items: center;

}
```



```
CSS

.container{
  justify-items: stretch;
}
```



This behavior can also be set on individual grid items via the `justify-self` property.

[top]

`align-items`

Aligns the content inside a grid item along the *row* axis (as opposed to `justify-items` which aligns along the *column* axis). This value applies to all grid items inside the container.

Values:

start - aligns the content to the top of the grid area
end - aligns the content to the bottom of the grid area
center - aligns the content in the center of the grid area

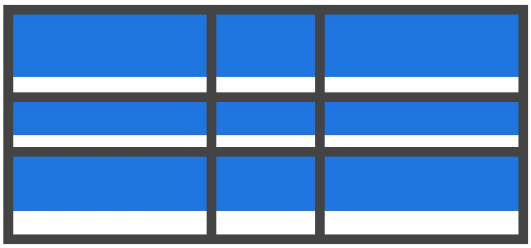
stretch - fills the whole height of the grid area (this is the default)

```
CSS
```

```
CSS
.container{
  align-items: start | end | center | stretch;
}
```

Examples:

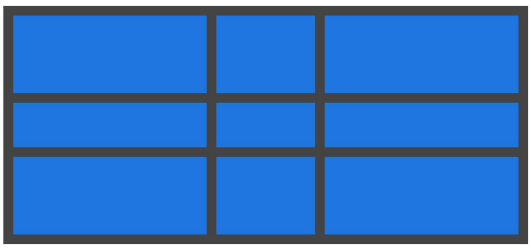
```
CSS
.container{
  align-items: start;
}
```



```
CSS
.container{
  align-items: end;
}
```

```
CSS
.container{
  align-items: center;
}
```

```
CSS

.container{
  align-items: stretch;
}
```



This behavior can also be set on individual grid items via the `align-self` property.

[top]

`justify-content`

Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like `px`. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the *column* axis (as opposed to `align-content` which aligns the grid along the *row* axis).

Values:

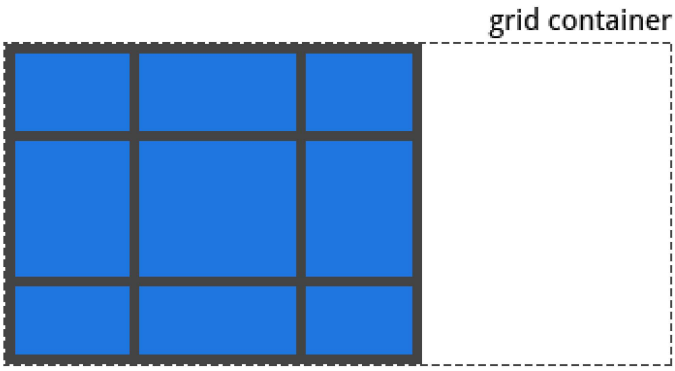
- **start** - aligns the grid to the left end of the grid container
- **end** - aligns the grid to the right end of the grid container
- **center** - aligns the grid in the center of the grid container
- **stretch** - resizes the grid items to allow the grid to fill the full width of the grid container

- **space-around** - places an even amount of space between each grid item, with half-sized spaces on the far ends
- **space-between** - places an even amount of space between each grid item, with no space at the far ends
- **space-evenly** - places an even amount of space between each grid item, including the far ends

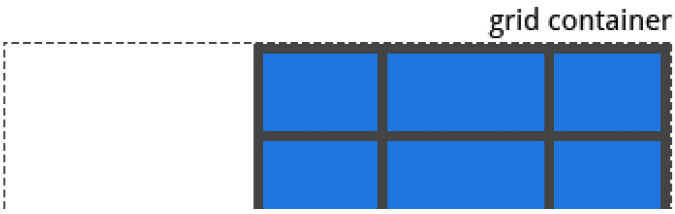
```
CSS

.container{
  justify-content: start | end | center | stretch | space-around | space-be
}
```

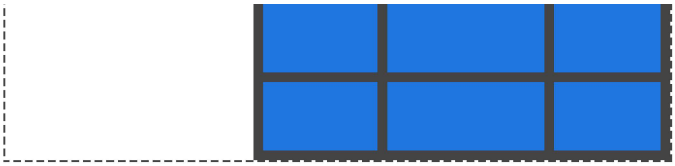
Examples:

```
CSS

.container{
  justify-content: start;
}
```

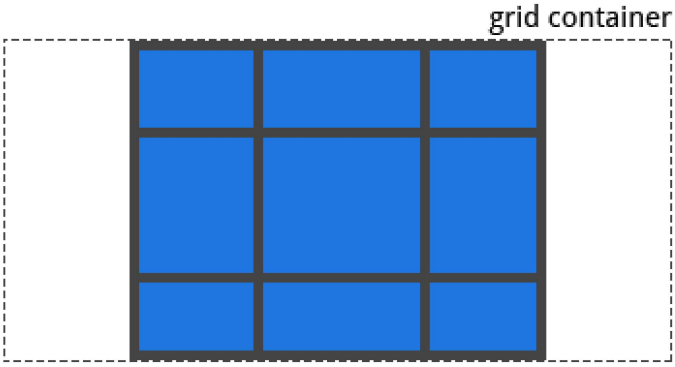



```
CSS

.container{
  justify-content: end;
}
```

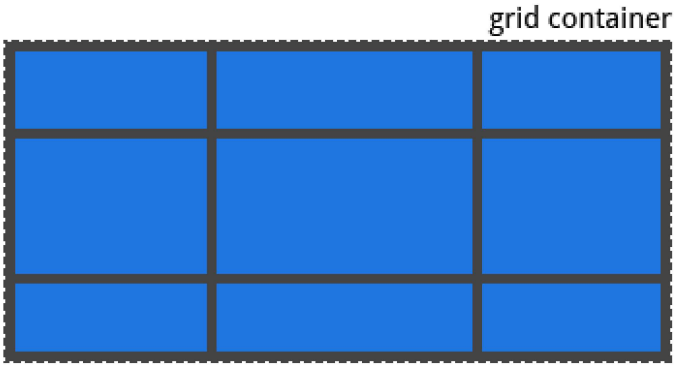




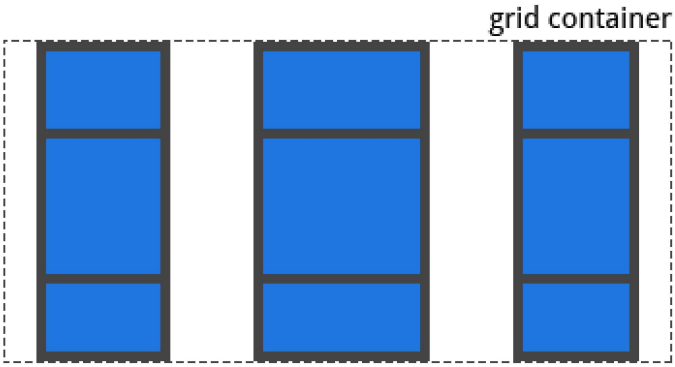
```CSS
.container{
  justify-content: center;
}
```



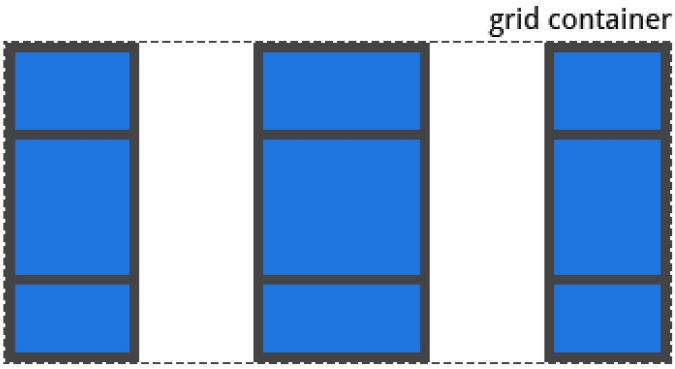

```CSS
.container{
  justify-content: stretch;
}
```



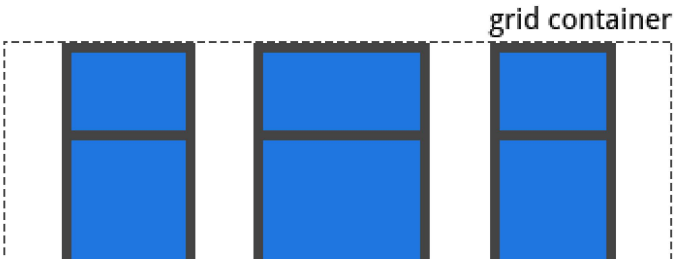

```CSS
.container{
  justify-content: space-around;
}
```




```CSS
.container{
  justify-content: space-between;
}
```




```CSS
.container{
  justify-content: space-evenly;
}
```



[top]

`align-content`

Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like `px`. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the *row* axis (as opposed to `justify-content` which aligns the grid along the *column* axis).

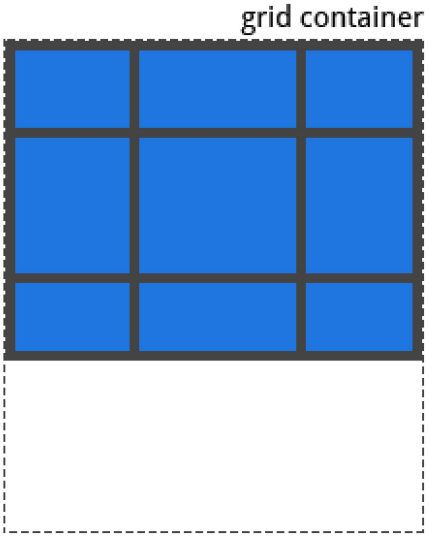
Values:

- **start** - aligns the grid to the top of the grid container
- **end** - aligns the grid to the bottom of the grid container
- **center** - aligns the grid in the center of the grid container
- **stretch** - resizes the grid items to allow the grid to fill the full height of the grid container
- **space-around** - places an even amount of space between each grid item, with half-sized spaces on the far ends
- **space-between** - places an even amount of space between each grid item, with no space at the far ends
- **space-evenly** - places an even amount of space between each grid item, including the far ends

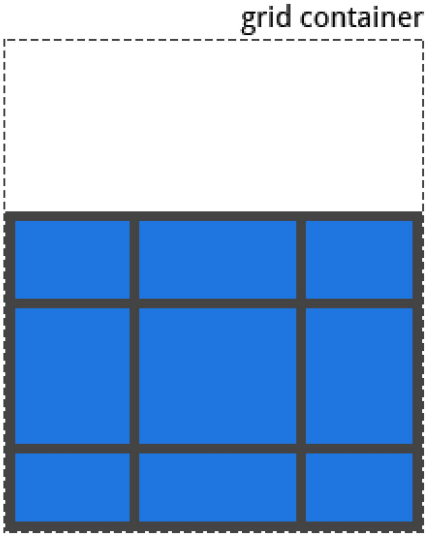
```
CSS

.container{
  align-content: start | end | center | stretch | space-around | space-betw
}
```

Examples:

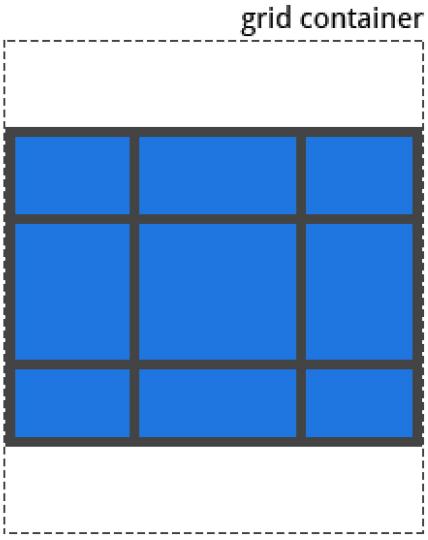
```css
.container{
  align-content: start;
}
```



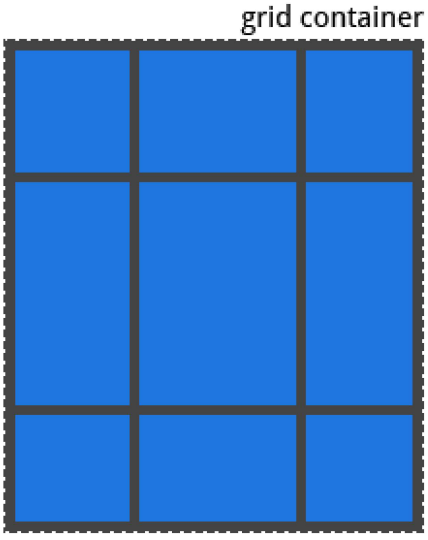

```css
.container{
  align-content: end;
}
```



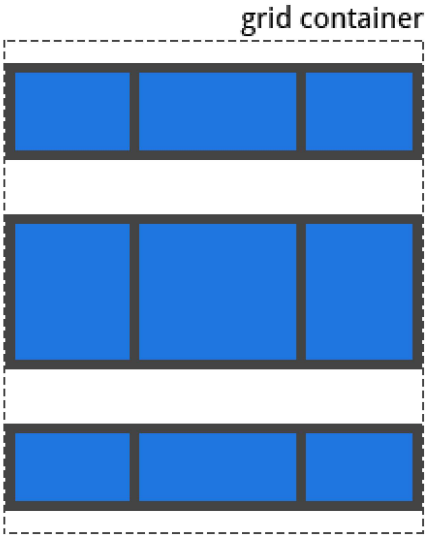
```css
.container{
  align-content: center;
}
```



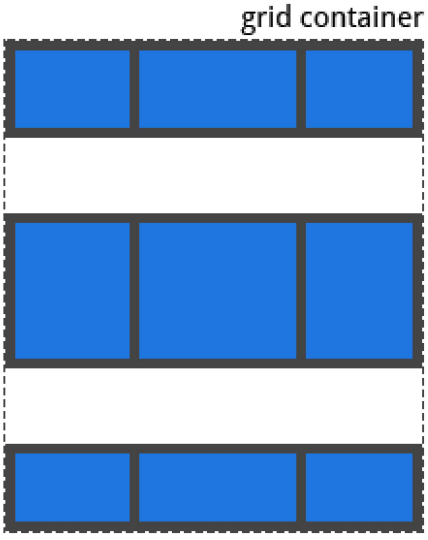

```css
.container{
  align-content: stretch;
}
```



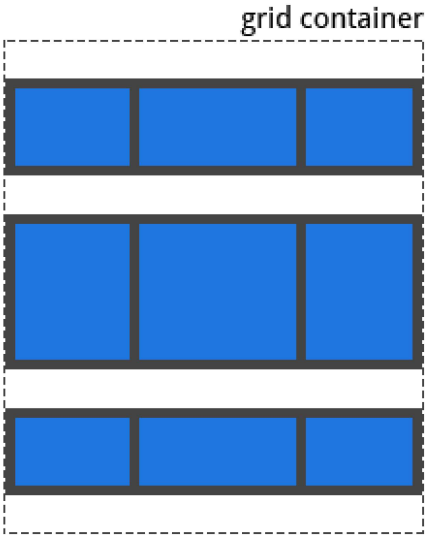
```css
.container{
  align-content: space-around;
}
```




```css
.container{
  align-content: space-between;
}
```



```css
.container{
  align-content: space-evenly;
}
```



[top]

grid-auto-columns
grid-auto-rows

Specifies the size of any auto-generated grid tracks (aka *implicit grid tracks*). Implicit grid tracks get created when you explicitly position rows or columns (via `grid-template-rows`/`grid-template-columns`) that are out of range of the defined grid.

Values:

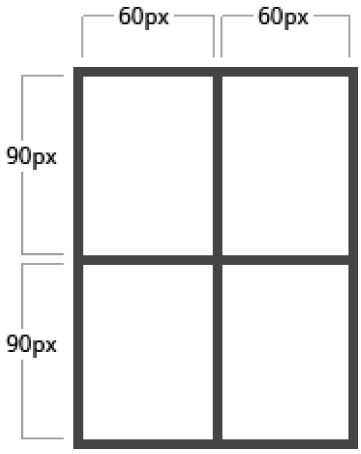
<track-size> - can be a length, a percentage, or a fraction of the free space in the grid (using the `fr` unit)

```css
.container{
  grid-auto-columns: <track-size> ...;
```

```
  grid-auto-rows: <track-size> ...;
}
```

To illustrate how implicit grid tracks get created, think about this:

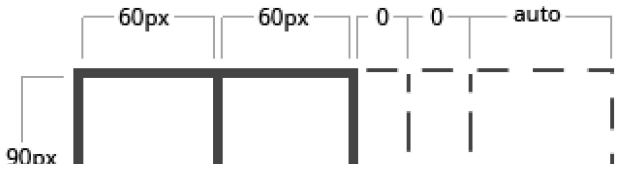
```CSS
.container{
  grid-template-columns: 60px 60px;
  grid-template-rows: 90px 90px
}
```

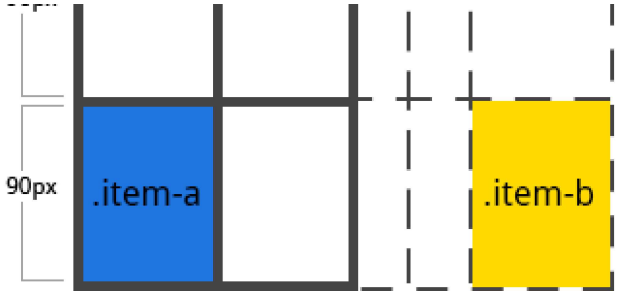


This creates a 2 x 2 grid.

But now imagine you use `grid-column` and `grid-row` to position your grid items like this:

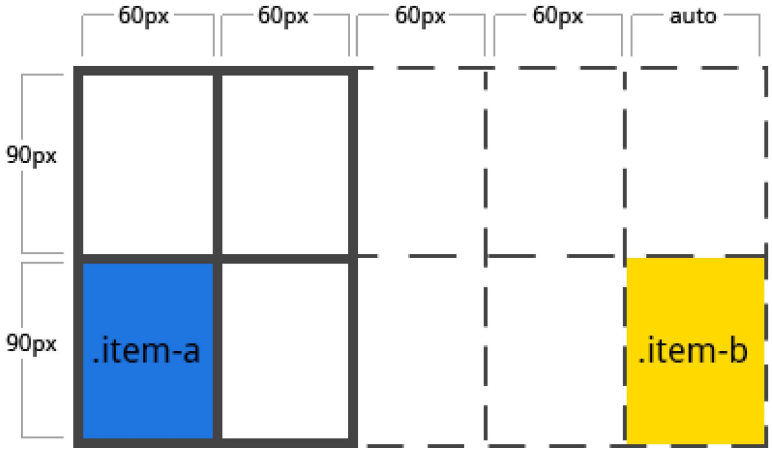
```CSS
.item-a{
  grid-column: 1 / 2;
  grid-row: 2 / 3;
}
.item-b{
  grid-column: 5 / 6;
  grid-row: 2 / 3;
}
```





We told .item-b to start on column line 5 and end at column line 6, *but we never defined a column line 5 or 6*. Because we referenced lines that don't exist, implicit tracks with widths of 0 are created to fill in the gaps. We can use `grid-auto-columns` and `grid-auto-rows` to specify the widths of these implicit tracks:

```
CSS

.container{
  grid-auto-columns: 60px;
}
```



[top]

grid-auto-flow

If you have grid items that you don't explicitly place on the grid, the *auto-placement algorithm* kicks in to automatically place the items. This property controls how the auto-placement algorithm works.

Values:

row - tells the auto-placement algorithm to fill in each row in turn, adding new rows as necessary
column - tells the auto-placement algorithm to fill in each column in turn, adding new columns as necessary
dense - tells the auto-placement algorithm to attempt to fill in holes earlier in the grid if smaller items come up later

```CSS
.container{
  grid-auto-flow: row | column | row dense | column dense
}
```

Note that **dense** might cause your items to appear out of order.

Examples:

Consider this HTML:

```HTML
<section class="container">
    <div class="item-a">item-a</div>
    <div class="item-b">item-b</div>
    <div class="item-c">item-c</div>
    <div class="item-d">item-d</div>
    <div class="item-e">item-e</div>
</section>
```

You define a grid with five columns and two rows, and set `grid-auto-flow` to `row` (which is also the default):

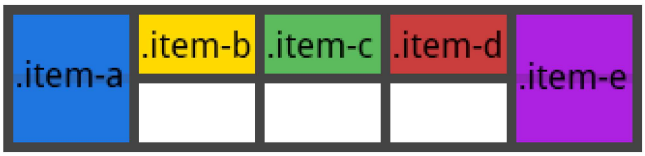
```CSS
.container{
    display: grid;
    grid-template-columns: 60px 60px 60px 60px 60px;
```

```
    grid-template-rows: 30px 30px;
    grid-auto-flow: row;
}
```

When placing the items on the grid, you only specify spots for two of them:

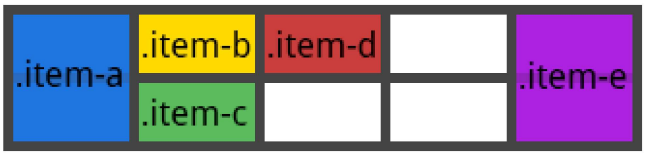
```
CSS

.item-a{
    grid-column: 1;
    grid-row: 1 / 3;
}
.item-e{
    grid-column: 5;
    grid-row: 1 / 3;
}
```

Because we set `grid-auto-flow` to `row`, our grid will look like this. Notice how the three items we didn't place (**item-b**, **item-c** and **item-d**) flow across the available rows:



If we instead set `grid-auto-flow` to `column`, **item-b**, **item-c** and **item-d** flow down the columns:

```
CSS

.container{
    display: grid;
    grid-template-columns: 60px 60px 60px 60px 60px;
    grid-template-rows: 30px 30px;
    grid-auto-flow: column;
}
```



[top]

grid

A shorthand for setting all of the following properties in a single declaration: `grid-template-rows`, `grid-template-columns`, `grid-template-areas`, `grid-auto-rows`, `grid-auto-columns`, and `grid-auto-flow`. It also sets `grid-column-gap` and `grid-row-gap` to their initial values, even though they can't be explicitly set by this property.

Values:

none - sets all sub-properties to their initial values
<grid-template-rows> / <grid-template-columns> - sets `grid-template-rows` and `grid-template-columns` to the specified values, respectively, and all other sub-properties to their initial values
<grid-auto-flow> [<grid-auto-rows> [/ <grid-auto-columns>]] - accepts all the same values as `grid-auto-flow`, `grid-auto-rows` and `grid-auto-columns`, respectively. If `grid-auto-columns` is omitted, it is set to the value specified for `grid-auto-rows`. If both are omitted, they are set to their initial values

```CSS
.container{
    grid: none | <grid-template-rows> / <grid-template-columns> | <grid-aut
}
```

Examples:

The following two code blocks are equivalent:

```CSS
.container{
    grid: 200px auto / 1fr auto 1fr;
}
```

```
}
```

CSS

```
.container{
    grid-template-rows: 200px auto;
    grid-template-columns: 1fr auto 1fr;
    grid-template-areas: none;
}
```

And the following two code blocks are equivalent:

CSS

```
.container{
    grid: column 1fr / auto;
}
```

CSS

```
.container{
    grid-auto-flow: column;
    grid-auto-rows: 1fr;
    grid-auto-columns: auto;
}
```

It also accepts a more complex but quite handy syntax for setting everything at once. You specify `grid-template-areas`, `grid-auto-rows` and `grid-auto-columns`, and all the other sub-properties are set to their initial values. What you're doing is specifying the line names and track sizes inline with their respective grid areas. This is easiest to describe with an example:

CSS

```
.container{
  grid: [row1-start] "header header header" 1fr [row1-end]
        [row2-start] "footer footer footer" 25px [row2-end]
        / auto 50px auto;
}
```

That's equivalent to this:

CSS

```
.container{
  grid-template-areas: "header header header"
                       "footer footer footer";
  grid-template-rows: [row1-start] 1fr [row1-end row2-start] 25px [row2-end
  grid-template-columns: auto 50px auto;
}
```

[top]

Properties for the Grid Items

grid-column-start

grid-column-end

grid-row-start

grid-row-end

Determines a grid item's location within the grid by referring to specific grid lines. `grid-column-start`/`grid-row-start` is the line where the item begins, and `grid-column-end`/`grid-row-end` is the line where the item ends.

Values:

<line> - can be a number to refer to a numbered grid line, or a name to refer to a named grid line

span <number> - the item will span across the provided number of grid tracks

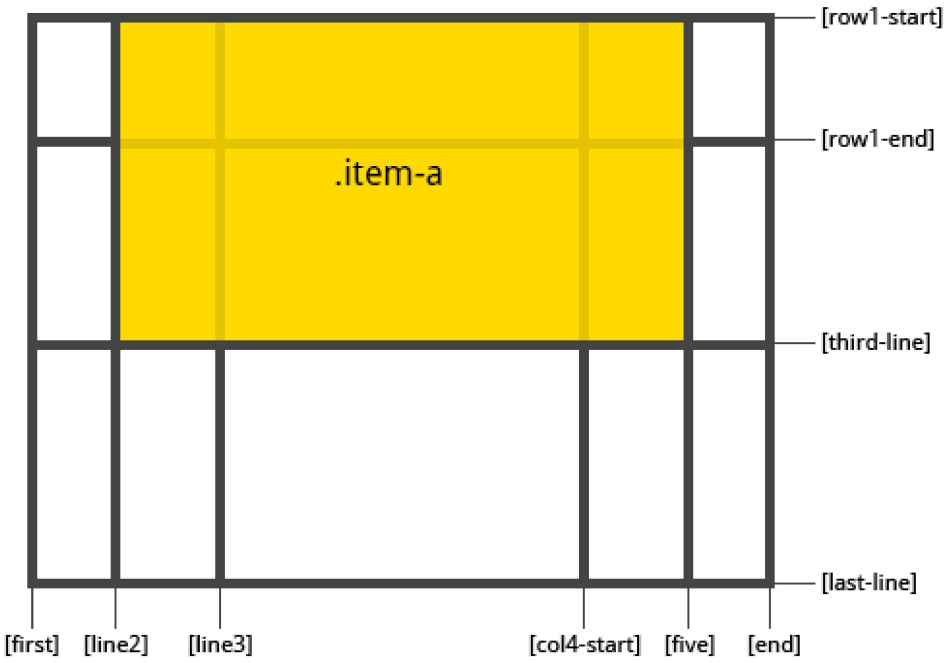
span <name> - the item will span across until it hits the next line with the provided name

auto - indicates auto-placement, an automatic span, or a default span of one

```
CSS

.item{
  grid-column-start: <number> | <name> | span <number> | span <name> | auto
  grid-column-end: <number> | <name> | span <number> | span <name> | auto
  grid-row-start: <number> | <name> | span <number> | span <name> | auto
  grid-row-end: <number> | <name> | span <number> | span <name> | auto
}
```

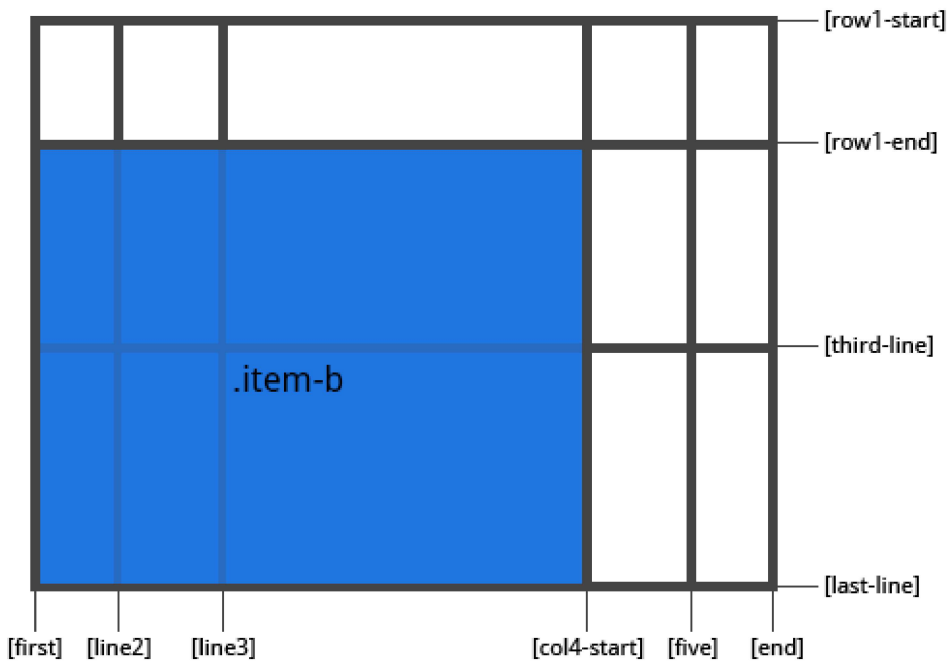
Examples:

```
CSS

.item-a{
  grid-column-start: 2;
  grid-column-end: five;
  grid-row-start: row1-start
  grid-row-end: 3
}
```




```
CSS

.item-b{
  grid-column-start: 1;
  grid-column-end: span col4-start;
```

```
  grid-row-start: 2
  grid-row-end: span 2
}
```



If no `grid-column-end` / `grid-row-end` is declared, the item will span 1 track by default.

Items can overlap each other. You can use `z-index` to control their stacking order.

[top]

`grid-column`
`grid-row`

Shorthand for `grid-column-start` + `grid-column-end`, and `grid-row-start` + `grid-row-end`, respectively.

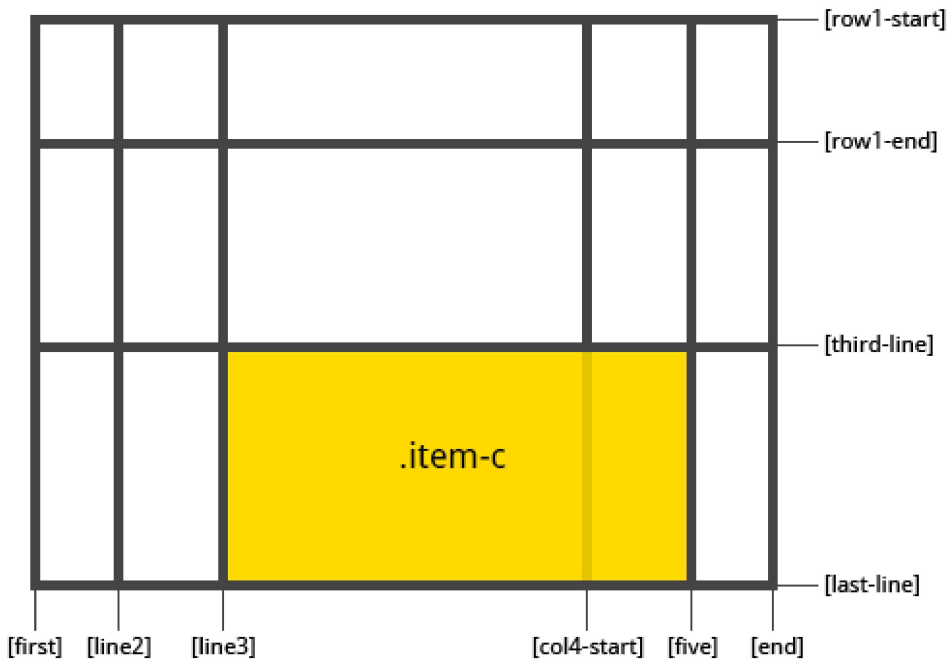
Values:

<start-line> / <end-line> - each one accepts all the same values as the longhand version, including span

```
CSS

.item{
  grid-column: <start-line> / <end-line> | <start-line> / span <value>;
  grid-row: <start-line> / <end-line> | <start-line> / span <value>;
}
```

Example:

```
CSS

.item-c{
  grid-column: 3 / span 2;
  grid-row: third-line / 4;
}
```



If no end line value is declared, the item will span 1 track by default.

[top]

grid-area

Gives an item a name so that it can be referenced by a template created with the `grid-template-areas` property. Alternatively, this property can be used as an even shorter shorthand for `grid-row-start` + `grid-column-start` + `grid-row-end` + `grid-column-end`.

Values:

<name> - a name of your choosing
<row-start> / <column-start> / <row-end> / <column-end> - can be numbers or named lines

```CSS
.item{
  grid-area: <name> | <row-start> / <column-start> / <row-end> / <column-en
}
```

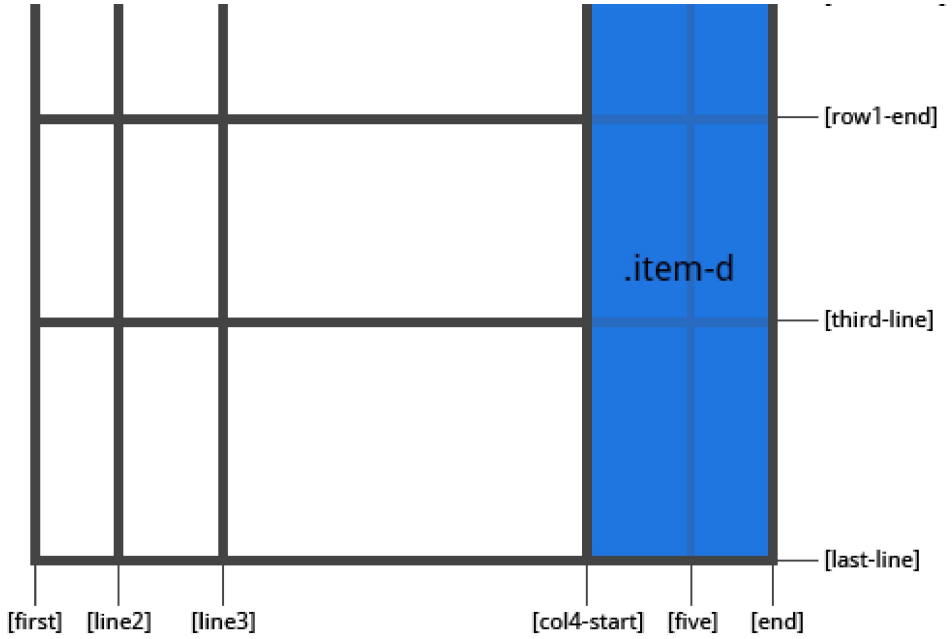
Examples:

As a way to assign a name to the item:

```CSS
.item-d{
  grid-area: header
}
```

As the short-shorthand for `grid-row-start` + `grid-column-start` + `grid-row-end` + `grid-column-end`:

```CSS
.item-d{
  grid-area: 1 / col4-start / last-line / 6
}
```



[top]

`justify-self`

Aligns the content inside a grid item along the *column* axis (as opposed to `align-self` which aligns along the *row* axis). This value applies to the content inside a single grid item.

Values:

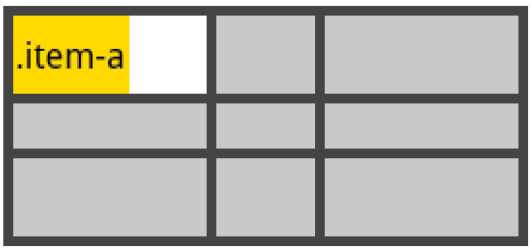
start - aligns the content to the left end of the grid area
end - aligns the content to the right end of the grid area
center - aligns the content in the center of the grid area
stretch - fills the whole width of the grid area (this is the default)

```CSS
.item{
  justify-self: start | end | center | stretch;
}
```

Examples:

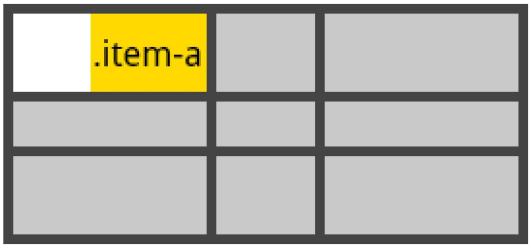
```CSS
```

```
.item-a{
  justify-self: start;
}
```



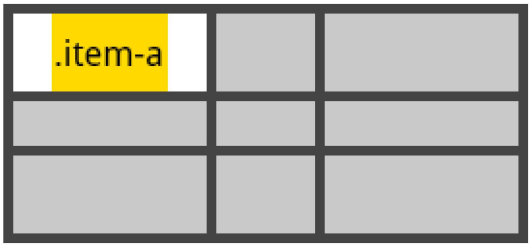
CSS

```
.item-a{
  justify-self: end;
}
```



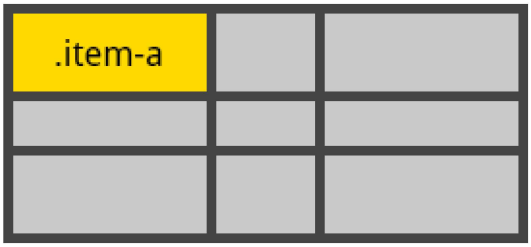
CSS

```
.item-a{
  justify-self: center;
}
```



CSS

```
.item-a{
  justify-self: stretch;
}
```



To set alignment for *all* the items in a grid, this behavior can also be set on the grid container via the `justify-items` property.

[top]

`align-self`

Aligns the content inside a grid item along the *row* axis (as opposed to `justify-self` which aligns along the *column* axis). This value applies to the content inside a single grid item.

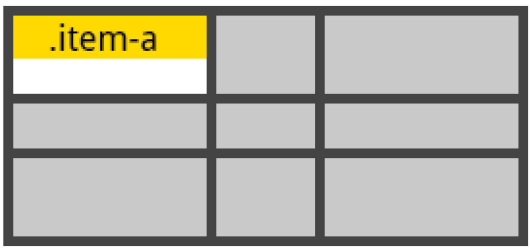
Values:

start - aligns the content to the top of the grid area
end - aligns the content to the bottom of the grid area
center - aligns the content in the center of the grid area
stretch - fills the whole height of the grid area (this is the default)

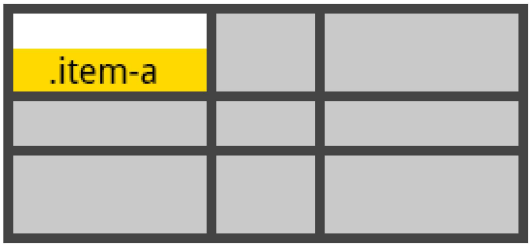
```CSS
.item{
  align-self: start | end | center | stretch;
}
```

Examples:

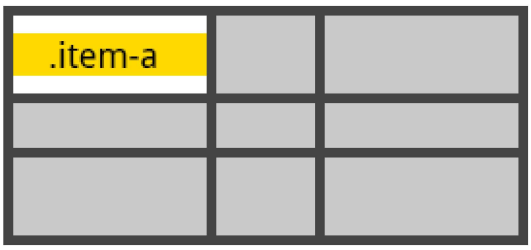
```css
.item-a{
  align-self: start;
}
```



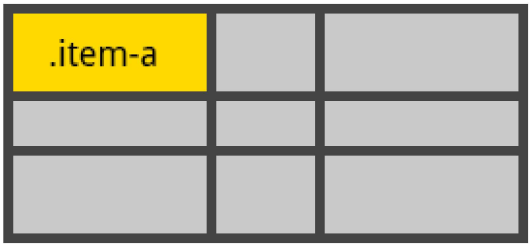

```css
.item-a{
  align-self: end;
}
```




```css
.item-a{
  align-self: center;
}
```



```css
.item-a{
  align-self: stretch;
}
```



To align *all* the items in a grid, this behavior can also be set on the grid container via the `align-items` property.

[top]

Facundo Corradini • a year ago

This got to be one of the best resources out there for CSS grids, but it lacks one of the most important features: the AUTO-FILL function, which is used to automatically add columns to the layout according to the container width.
This is extremely useful, and combined with the minmax() function, it provides a perfect flexbox alternative, even "fixing" the behaviour of the last row on a justify-content:space-between / space-around flex container.

.grid{
/*sets the container as a grid with variable number of columns*/
display:grid;
grid-template-columns: repeat(auto-fill, minmax(150px,1fr));
}

Here's a working example at my CodePen: https://codepen.io/facundoc...

And of course, a link to the specification: https://drafts.csswg.org/cs...

cheers!

Reply • Share ›

Brad Dalton • a year ago

Safari, iOS Safari and Chrome for Android now supported

Please update http://caniuse.com/#feat=cs...

Any idea when I.E will support?

Reply • Share ›

guoliim • 2 years ago

nice guide

Reply • Share ›

Mojtaba Seyedi • 2 years ago

Great post, Thanks a lot
In alignment properties there is a mistake about column-axis and row-axis. for example justify-content aligns grid items along row-axis not column-axis.

Reply • Share ›

Miles Rausch • 2 years ago

Came across this guide in "This Week's HTML5, CSS and Browser Tech News #283." Now might be a good time to go through and update given all the grid news this month.

Reply • Share ›

Stephen Ainsworth • 2 years ago

Great tutorial. One of the better ones I've gone through. I think you're grid gap is the wrong way around, it should be:

grid-gap: <grid-row-gap> <grid-column-gap>;

Reply • Share ›

Polarnj • 2 years ago

Honesty...is grid layout ever going to be supported? Weve been hearing how great t will be for like 4 years.. :/

• Reply • Share ›

Marks ➚ Polarnj • 2 years ago

CSS Grid Layout Module Level 1 is now (as of Feb 9th 2017) W3C Candidate Recommendation :-)

1 • Reply • Share ›

Oliver Williams • 2 years ago

What about auto fill and min-max?

• Reply • Share ›

UI DESK • 3 years ago

awesome! complete reference on grid layout!

• Reply • Share ›

ALSO ON **CHRIS.HOUSE**

Grunt configuration for React + Browserify + Babelify

5 comments • 3 years ago

Avatar **Bijomon Varghese** — Thanks a ton Chris, your browserify configuration saved my life. Changing my configuration with yours reduced

Why CSS Custom Properties (a.k.a. CSS Variables) are Awesome

1 comment • 3 years ago

Avatar **Peter L** — CSS custom properties are a great asset, but they should have existed much earlier. IE doesn't support them, but is still

Thinking About the :has() CSS Selector

5 comments • 3 years ago

Building a Home Page with Grid

2 comments • 3 years ago

me@chris.house

chrishouse

chrishouse83

Chris House is a web designer and front-end developer from Kansas City.