CSS Grid

CSS Grid Layout, is a two-dimensional grid-based layout system that, compared to any web layout system of the past, completely changes the way we design user interfaces.

To get started you have to 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`.

Syntax:

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

- **`grid`** – generates a block-level grid
- **`inline-grid`** – generates an inline-level grid

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 (more on this unit over at DigitalOcean)
- `<line-name>` – an arbitrary name of your choosing

```
.container {
  grid-template-columns: ...  ...;
  /* e.g.
     1fr 1fr
     minmax(10px, 1fr) 3fr
     repeat(5, 1fr)
     50px auto 100px 1fr
  */
  grid-template-rows: ... ...;
  /* e.g.
     min-content 1fr min-content
     100px 1fr max-content
  */
}
```

Grid lines are automatically assigned positive numbers from these assignments .

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

```
.container {
  grid-template-columns: [first] 40px [line2] 50px [line3] auto [col4-start] 50px [five] 40px [end];
  grid-template-rows: [row1-start] 25% [row1-end] 100px [third-line] auto [last-line];
}
```

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

```
.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:

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

Which is equivalent to this:

```
.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:

```
.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:

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

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

```
.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:

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

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

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

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

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

Example:

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

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

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

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

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

grid-template

A shorthand for setting `grid-template-rows`, `grid-template-columns`, and `grid-template-areas` in a single declaration.

Values:

none – sets all three properties to their initial values

`<grid-template-rows> / <grid-template-columns>` – sets grid-template-columns and grid-template-rows to the specified values, respectively, and sets grid-template-areas to none

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

It also accepts a more complex but quite handy syntax for specifying all three. Here's an example:

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

That's equivalent to this:

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

column-gap & row-gap & 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

```
.container {
  /* standard */
  column-gap: <line-size>;
  row-gap: <line-size>;

  /* old */
  grid-column-gap: <line-size>;
  grid-row-gap: <line-size>;
}
```

`gap` is a shorthand for `row-gap` and `column-gap`

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

justify-items

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

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

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

align-items

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

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

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

place-items

`place-items` sets both the `align-items` and `justify-items` properties in a single declaration.

Values:

- **`<align-items>` / `<justify-items>`** – The first value sets `align-items`, the second value `justify-items`. If the second value is omitted, the first value is assigned to both properties.

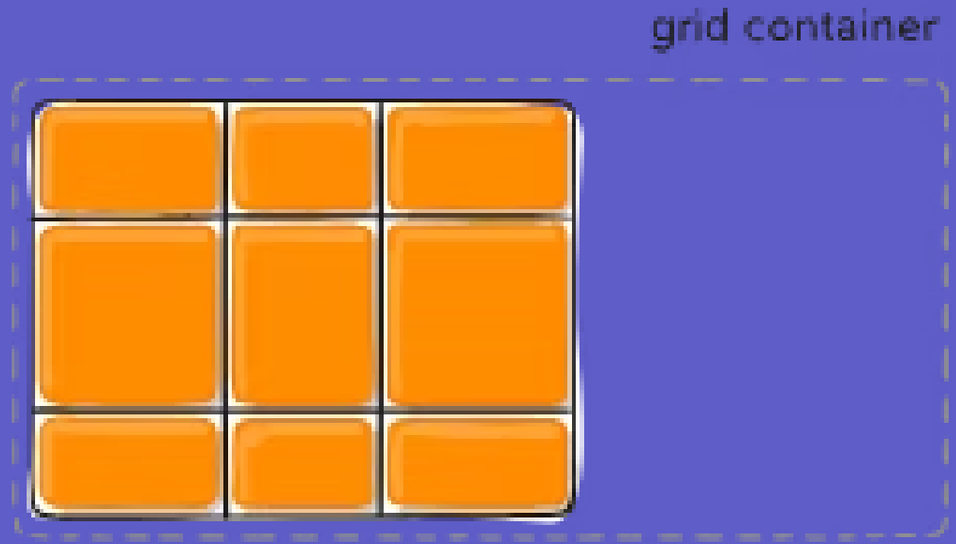
This can be very useful for super quick multi-directional centering:

```
.center {
 display: grid;  place-items: center;
}
```

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 *inline (row)* axis (as opposed to `align-content` which aligns the grid along the *block (column)* axis).

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



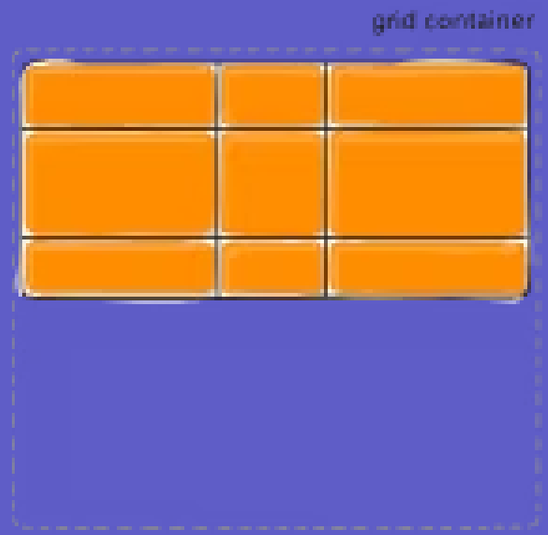
Examples:

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

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 *block (column)* axis (as opposed to `justify-content` which aligns the grid along the *inline (row)* axis).

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



Examples:

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

place-content

`place-content` sets both the `align-content` and `justify-content` properties in a single declaration.

Values:

- **`<align-content>` / `<justify-content>`** – The first value sets `align-content`, the second value `justify-content`. If the second value is omitted, the first value is assigned to both properties.

All major browsers except Edge support the `place-content` shorthand property.

grid-auto-columns & grid-auto-rows

Specifies the size of any auto-generated grid tracks (aka *implicit grid tracks*). Implicit tracks get created when there are more grid items than cells in the grid or when a grid item is placed outside of the explicit grid.

Values:

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

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 (default)
- `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

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

Media query

CSS Media query uses the `@media` rule to include a block of CSS properties only if a certain condition is true.

The most common media queries in the world are those that target particular viewport ranges and apply custom styles, which birthed the whole idea of responsive design.

```
/* When the browser is at least 600px and above */
@media screen and (min-width: 600px) {
  .element {
    /* Apply some styles */
  }
}
```

There are a few ways we can use media queries directly in HTML.

```
  <!-- Served to all users -->
    <link rel="stylesheet" href="all.css" media="all" />
    <!-- Served to screens that are at least 20em wide -->
    <link rel="stylesheet" href="small.css" media="(min-width: 20em)" />
    <!-- Served to screens that are at least 64em wide -->
    <link rel="stylesheet" href="medium.css" media="(min-width: 64em)" />
    <!-- Served to screens that are at least 90em wide -->
    <link rel="stylesheet" href="large.css" media="(min-width: 90em)" />
    <!-- Served to screens that are at least 120em wide -->
    <link rel="stylesheet" href="extra-large.css" media="(min-width: 120em)" />
    <!-- Served to print media, like printers -->
    <link rel="stylesheet" href="print.css" media="print" />
```

Another example related using media query in html:

```
<picture>
  <!-- Use this image if the screen is at least 800px wide -->
  <source srcset="cat-landscape.png" media="(min-width: 800px)">
  <!-- Use this image if the screen is at least 600px wide -->
  <source srcset="cat-cropped.png" media="(min-width: 600px)">

  <!-- Use this image if nothing matches -->
  <img src="cat.png" alt="A calico cat with dark aviator sunglasses.">
</picture>
```

Typical Device Breakpoints:

```
// X-Small devices (portrait phones, less than 576px)
// No media query for `xs` since this is the default in Bootstrap

// Small devices (landscape phones, 576px and up)
@media (min-width: 576px) { ... }

// Medium devices (tablets, 768px and up)
@media (min-width: 768px) { ... }

// Large devices (desktops, 992px and up)
@media (min-width: 992px) { ... }

// X-Large devices (large desktops, 1200px and up)
@media (min-width: 1200px) { ... }

// XX-Large devices (larger desktops, 1400px and up)
@media (min-width: 1400px) { ... }
```