CSS Grid

CSS Grid is the new kid in the CSS town,and while not yet fully supported by the browsers , it's going to be the future system of layouts.

CSS Grid is a fundamentally new approach to building layouts using CSS.

CSS Grid is not a competitor to Flexbox .They interoperate and collaborate on complex layouts, because CSS Grid works on 2 dimensions (rows AND columns) while Flexbox works on a single dimension.

Building layouts for the web has traditionally been a complicated topic.The 2 powerful and supported tools at your disposal:

- **CSS Flexbox**
- **CSS Grid**

These are the 2 tools to build out the web layouts of the future.

The Basics

The CSS Grid layout is activated on a container element by setting display:grid.

As with flexbox,you can define some properties on the container, and some properties on each individual item in the grid.

These properties combined will determine final look of the grid.

The most basic containers properties are grid-template-columns and grid-template-rows.

Grid-template-columns and grid-template-rows

Those properties define the number of columns and rows in the grid,and they also set the width of each column/row

The following snippet defines a grid with 4 column each 200px wide, and 2 rows with a 300px height each.

.container {

display:grid;

grid-template-columns:200px 200px 200px 200px;

grid-template-rows:300px 300px;

}

Adding space between the cells

Unless specified, there is no space between the cells.

You can add spacing by using those properties:

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

Example:

.container {

display:grid;

grid-template-columns:100px 200px;

grid-template-rows:100px 50px;

grid-column-gap:20px;

grid-row-gap:25px;

}

Spawning items on multiple columns and rows

Every cell item has the options to occupy more than just one box in the row,and expand horizontally or vertically to get more space,while respecting grid proportions in the container

Those are the properties we will use for that:

- grid-column-start
- grid-column-end
- grid-row-start
- grid-row-end

example:

.container{

grid-template-columns:100px 200px;

grid-template-rows:100px 50px;

}
.item1{
grid-column-start:2;
grid-column-end:4;
}

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

.container {

display:grid | inline-grid;

}

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:

“ | . |none | ...”

“...”;

}

Example:

.item-a{

grid-area:header;

}

.item-b{

grid-area:sidebar;

}

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.

Values:

- **start** – aligns items to be flush with the start edge of their cell
- **end** – aligns items to be flush with the end edge of their cell
- **center** – aligns items in the center of their cell
- **stretch** – fills the whole width of the cell (this is the default)

Example:

.container{

justify-items:start;

}

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.

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

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).

Values:

- **start** – aligns the grid to be flush with the start edge of the grid container
- **end** – aligns the grid to be flush with the end edge 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

  Example:

  .container {

  Justify-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.

Example:

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

Wrapping up

These are the basics of CSS Grid.There are still some things which I didn't include as the document will be of too many pages.

Flexbox

Flexbox is one of the two modern layouts systems along with CSS Grid.

Compared to grid ,flexbox is a one-dimensional layout model.It will control the layout based on a row or on a column but not together at the same time.

The main goal of flexbox is to allow items to fill the whole space offered by the container, depending on some rules you set

Properties
(flex container)

display

This defines a flex container; inline or block depending on the given value. It enables a flex context for all its direct children.

```
.container {
  display: flex; /* or inline-flex */
}
```

Note that CSS columns have no effect on a flex container.

flex-direction

This establishes the main-axis, thus defining the direction flex items are placed in the flex container. Flexbox is (aside from optional wrapping) a single-direction layout concept. Think of flex items as primarily laying out either in horizontal rows or vertical columns.

```
.container {
  flex-direction: row | row-reverse | column | column-reverse;
}
```

- row (default): left to right in ltr; right to left in rtl
- row-reverse: right to left in ltr; left to right in rtl
- column: same as row but top to bottom
- column-reverse: same as row-reverse but bottom to top

flex-wrap

By default, flex items will all try to fit onto one line. You can change that and allow the items to wrap as needed with this property.

```
.container {
  flex-wrap: nowrap | wrap | wrap-reverse;
}
```

- nowrap (default): all flex items will be on one line
- wrap: flex items will wrap onto multiple lines, from top to bottom.
- wrap-reverse: flex items will wrap onto multiple lines from bottom to top.

justify-content

This defines the alignment along the main axis. It helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have

reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.

```
.container {
  justify-content: flex-start | flex-end | center | space-between |
space-around | space-evenly | start | end | left | right ... + safe
| unsafe;
}
```

- flex-start (default): items are packed toward the start of the flex-direction.
- flex-end: items are packed toward the end of the flex-direction.
- start: items are packed toward the start of the writing-mode direction.
- end: items are packed toward the end of the writing-mode direction.
- left: items are packed toward left edge of the container, unless that doesn't make sense with the flex-direction, then it behaves like start.
- right: items are packed toward right edge of the container, unless that doesn't make sense with the flex-direction, then it behaves like start.
- center: items are centered along the line
- space-between: items are evenly distributed in the line; first item is on the start line, last item on the end line
- space-around: items are evenly distributed in the line with equal space around them. Note that visually the spaces aren't equal, since all the items have equal space on both sides. The first item will have one unit of space against the container edge, but two units of space between the next item because that next item has its own spacing that applies.
- space-evenly: items are distributed so that the spacing between any two items (and the space to the edges) is equal.

align-items

This defines the default behavior for how flex items are laid out along the **cross axis** on the current line. Think of it as the justify-content version for the cross-axis (perpendicular to the main-axis).

```
.container {
  align-items: stretch | flex-start | flex-end | center | baseline | first baseline |
last baseline | start | end | self-start | self-end + ... safe | unsafe;
}
```

- stretch (default): stretch to fill the container (still respect min-width/max-width)
- flex-start / start / self-start: items are placed at the start of the cross axis. The difference between these is subtle, and is about respecting the flex-direction rules or the writing-mode rules.
- flex-end / end / self-end: items are placed at the end of the cross axis. The difference again is subtle and is about respecting flex-direction rules vs. writing-mode rules.
- center: items are centered in the cross-axis
- baseline: items are aligned such as their baselines align

align-content

This aligns a flex container's lines within when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis.

```
.container {
  align-content: flex-start | flex-end | center | space-between | space-around | space-evenly | stretch | start | end | baseline | first baseline | last baseline + ... safe | unsafe;
}
```

- flex-start / start: items packed to the start of the container. The (more supported) flex-start honors the flex-direction while start honors the writing-mode direction.
- flex-end / end: items packed to the end of the container. The (more support) flex-end honors the flex-direction while end honors the writing-mode direction.
- center: items centered in the container
- space-between: items evenly distributed; the first line is at the start of the container while the last one is at the end
- space-around: items evenly distributed with equal space around each line
- space-evenly: items are evenly distributed with equal space around them
- stretch (default): lines stretch to take up the remaining space

Difference between CSS Grid and Flexbox

- Flexbox is designed for one-dimensional layouts, and Grid for two-dimensional layouts.
- The approach of CSS Grid is the layout first, while the Flexbox approach is primarily the content.
- The Flexbox layout is best suited to application components and small-scale layouts, while the Grid layout is designed for larger-scale layouts that are not linear in design.
    - Flexbox offers greater control over alignment and space distribution between items. Being one-dimensional, Flexbox only deals with either columns or rows. This system works for smaller layouts, but cannot render complex displays such as text or document-centric properties that enable floats and columns.
    - Grid has two-dimension layout capabilities which allow flexible widths as a unit of length. This compensates for the limitations in Flex.

Positioning

The `position` property specifies the type of positioning method used for an element (static, relative, fixed, absolute or sticky).

The position Property

The `position` property specifies the type of positioning method used for an element.

There are five different position values:

- `static`

- relative
- fixed
- absolute
- sticky

Elements are then positioned using the top, bottom, left, and right properties. However, these properties will not work unless the position property is set first. They also work differently depending on the position value.

position: static

HTML elements are positioned static by default.

Static positioned elements are not affected by the top, bottom, left, and right properties.

An element with position: static; is not positioned in any special way; it is always positioned according to the normal flow of the page:

This element has position: static;

Here is the CSS that is used:

Example

```
div.static {
  position: static;
  border: 3px solid #73AD21;
}
```

position: relative;

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

This element has position: relative;

Here is the CSS that is used:

Example

```
div.relative {
  position: relative;
  left: 30px;
  border: 3px solid #73AD21;
}
```

position: fixed;

An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

A fixed element does not leave a gap in the page where it would normally have been located.

Notice the fixed element in the lower-right corner of the page. Here is the CSS that is used:

Example

```
div.fixed {
  position: fixed;
  bottom: 0;
  right: 0;
  width: 300px;
  border: 3px solid #73AD21;
}
```

position: absolute;

An element with position: absolute; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).

However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

Note: A "positioned" element is one whose position is anything except static.

Here is the CSS that is used:

Example

```
div.relative {
  position: relative;
  width: 400px;
  height: 200px;
  border: 3px solid #73AD21;
}

div.absolute {
  position: absolute;
  top: 80px;
  right: 0;
  width: 200px;
  height: 100px;
  border: 3px solid #73AD21;
}
```

position: sticky;

An element with position: sticky; is positioned based on the user's scroll position.

A sticky element toggles between relative and fixed, depending on the scroll position. It is positioned relative until a given offset position is met in the viewport - then it "sticks" in place (like position:fixed).

Example

```
div.sticky {
  position: -webkit-sticky; /* Safari */
  position: sticky;
  top: 0;
  background-color: green;
  border: 2px solid #4CAF50;
}
```