basic grid layouts

now that CSS Grid has become better supported in all major browsers, it is time to look at some of the more whimscal layouts that are possible.

This post will look at the following areas:

- Positioning elements in a grid
- Combining grids and flexboxes
- Area templates

All items should work in all browsers. According to caniuse.com, CSS Grid is supported in all major browsers.

The basic grids

We will use the following grids: a 12 equal columns and a compound 4x6.

Both grids will have equal rows of 200px and will have 1em gap between columns and rows.

The 12 column grid is the most basic one. The code looks like this

```
.container {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  grid-auto-rows: minmax(200px, auto);
  grid-gap: 1em;
}
```

I learned about compound grids from one of Andrew Clarke's Inspired By Design presentations for Smashing Magazine. The grid still presents the equivalent to 12 columns but it is organized differntly so you can play with more combinations of positions and space.

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

```
  grid-template-columns: 2fr 1fr 1fr 2fr 2fr 1fr 1fr 2fr;
  grid-auto-rows: minmax(200px, auto);
  grid-gap: 1em;
}
```

now that we have the grids, we can start playing with them :)

Positioning elements in a grid

One of the first things that attracted me to learning and working with CSS grid is the ability to position elements in specific areas of the grid.

Simple positioning in a grid

The first example positions items in a 12-column grid.

```
header {
  grid-column: 3 /-3;
  grid-row: 1;
}

#content {
  grid-column: 3 / span 2;
  grid-row: 2;

  line-height: 1.3
}

#fig1 {
  grid-column: 8 / span 2;
  grid-row: 2;
}

footer {
  grid-column: 3 / -3;
  grid-row: 8;
}
```

Using different combinations of `grid-column` and `grid-row` we can position items in the grid.

The `header` element stqarting in the 3rd column and ending in the 3rd column from the end. The content is placed in row one.

The `#content` element and all its children are placed in column 3 and span 2 columns. We place the element in row 2.

`#fig1` is placed in column 8 and span 2 columns. We place it in row 2 so it'll look even with the content text.

The `footer` element is places in the same columns as the `header`. We place it in row 8 so we can have space to put further content before it.

the result look like this:

Calendar: Combining grids with flexbox

A 30-day calendar is a good way to show how a combination of flexbox and grid works.

Rather than our standard grid, we'll use a 7 column grid with a 1em gap between columns. Instead of making the columns fractions of the screen width, we'll make them 300px wide and provide a media query to make them single column in smaller form factors.

We set the grid with seven repeating frames but they all have a fixed width instead of being proportionally sized

```
.calendar-container {
  display: grid;
  grid-template-columns: repeat(7, 300px);
  gap: 1em;
}
```

Each calendar entry has a class of `calendar-day` and is set up as a column-based flexbox. The calendar day is divided into two parts: the day number and the day's content.

```
.calendar-day {
  display: flex;
  gap: 1em;
  flex-flow: column;
  border: 2px solid black;
}
```

The `calendar--day-number` element also uses `display: flex` to create a row-based flexbox. We then center the day number in the row and align it to the end of the container (speaking logically for left to right languages).

We also add a background color (rebeccapurple) and a color for the number (white).

```
.calendar-day--number {
  display: flex;
  flex-flow: row;
  align-items: center;
  justify-content: flex-end;
  background-color:tran rebeccapurple;
  color: white;
  height: 50px;
  padding: 0;
}
```

The entry for the calendar day just adjust the left and right padding to make the text easier to read.

```
.calendar-day--content {
  padding: 0 1em 0 1em;
}
```

To make sure that the calendar will remain workable on smaller devices, we add a media query to make the calendar a single column when the device width is 600px or less.

We could also change the color or make the day a little easier to read.

```
@media (max-width: 600px) {
  .calendar-container {
    display: flex;
    flex-flow: column;
    gap: 1em;
  }
}
```

Placing content in grid areas

Most of the time I place items in the grid using explicit grid positioning for both column and row. This is a good way to keep things simple and avoid confusion.

CSS Grid provides another way to position content: grid template areas.

The idea is that when we define the grid we define areas where we want to place the content in the `grid-template-areas` property.

The following example creaates a 6 column grid and 100px rows. it then defines the following areas for the grid:

1. a header area spaning four columns and two rows
2. a left sidebar spanning 1 column and two rows
3. a content area spaning four columns and two rows
4. another sidebar spanning 1 column and two rows
5. a footer spanning six columns and two rows

the `...` notation is used to indicate an empty area

```
.container {
  display: grid;
  grid-gap: 1em;
  grid-template-columns: repeat(6, 1fr);
  grid-auto-rows: 100px;
  grid-template-areas:
    "... header header header header ..."
    "... header header header header ..."
    "sidebar content content content content sidebar2"
    "sidebar content content content content sidebar2"
    "footer footer footer footer footer footer"
    "footer footer footer footer footer footer";
}
```

We then place the content in the areas defined on the grid using `grid-area` rather than using the specific column/row placement we used in previous examples.

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

.sidebar2 {
  grid-area: sidebar2;
}
.content {
  grid-area: content;
}

.header {
  grid-area: header;
}

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

The code looks like this