CSS Grid – Table layout is back. Be there and be square.

By Surma

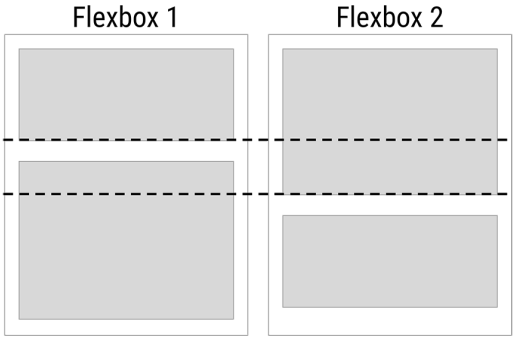
Surma is a contributor to Web**Fundamentals**

TL;DR

If you are familiar with Flexbox, Grid should feel familiar. Rachel Andrew maintains a great website dedicated to CSS Grid to help you get started. Grid is now available in Google Chrome.

Flexbox? Grid?

Over the past few years, CSS Flexbox has become widely used and browser support is looking really good (unless you are one of the poor souls that have to support IE9 and below). Flexbox made a lot of complex layout tasks easier, like equi-distant spacing between elements, top-to-bottom layouts or the holy grail of CSS wizardry: vertical centering.



But alas, screens commonly have a second dimension we need to worry about. Short of taking care of sizing the elements yourself, sadly, you can't have *both* a vertical and horizontal rhythm by just using flexbox alone. This is where CSS Grid comes to the rescue.

CSS Grid has been in development, behind a flag in most browsers for over *5 years* and extra time has been spent on interoperability to avoid a buggy launch like Flexbox had. So if you use Grid to implement your layout in Chrome, chances are you'll get the same result in

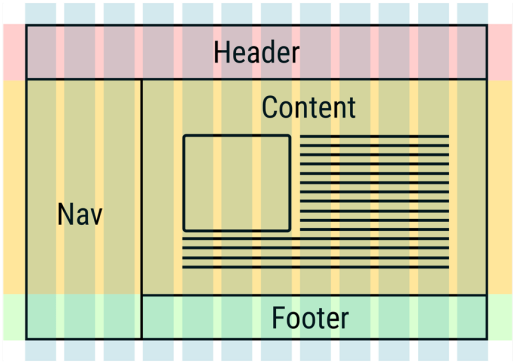
Firefox and Safari. At the time of writing, Microsoft's Edge implementation of Grid is out of date (the same as was already present in IE11.) and the update is "under consideration".

Despite the similarities in concept and syntax, don't think of Flexbox and Grid as competing layout techniques. Grid arranges in two dimensions, while Flexbox lays out in one. There is synergy when using the two together.

Defining a grid

To get familiar with the individual properties of Grid I heartily recommend Rachel Andrew's Grid By Example or CSS Tricks' Cheat Sheet. If you are familiar with Flexbox, a lot of the properties and their meaning should be familiar.

Let's take a look at a standard 12-column grid layout. The classic 12-column layout is popular as the number 12 is divisible by 2, 3, 4 and 6, and is therefore useful for many designs. Let's implement this layout:



Let's start with our markup code:

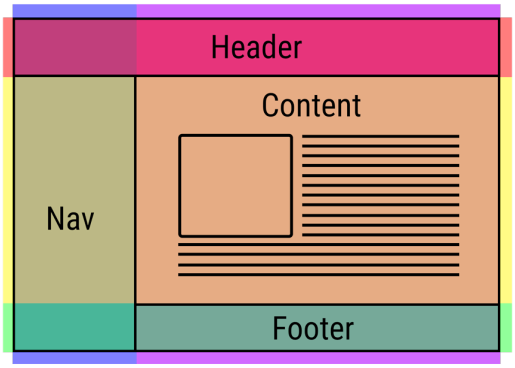
```
<!DOCTYPE html>
<body>
  <header></header>
  <nav></nav>
  <main></main>
  <footer></footer>
</body>
```

In our stylesheet we start by expanding our **body** so it covers the entire viewport and turning it into a *grid container*:

```
html, body {
  width: 100vw;
  min-height: 100vh;
  margin: 0;
  padding: 0;
}
body {
  display: grid;
}
```

Now we are using CSS Grid. Hooray!

The next step is to implement the rows and columns of our grid. We could implement all 12 columns in our mockup, but as we are not using every column, doing this would make our CSS unnecessarily messy. For the sake of simplicity, we'll implement the layout this way:



The header and the footer are variable in width and the content is variable in both dimensions. The nav will be variable in both dimensions as well, but we are going to impose a minimum width of 200px on it. (Why? To show off the features of CSS Grid, of course.)

In CSS Grid, the set of columns and rows are called *tracks*. Let's start with defining our first set of tracks, the rows:

```
body {
  display: grid;
  grid-template-rows: 150px auto 100px;
}
```

`grid-template-rows` takes a sequence of sizes that define the individual rows. In this case, we give the first row a height of 150px and the last one of 100px. The middle row is set to **`auto`** which means it will adjust to the necessary height to accommodate the *grid items* (the children of the *grid container*) in that row. Since our body is stretched across the entire

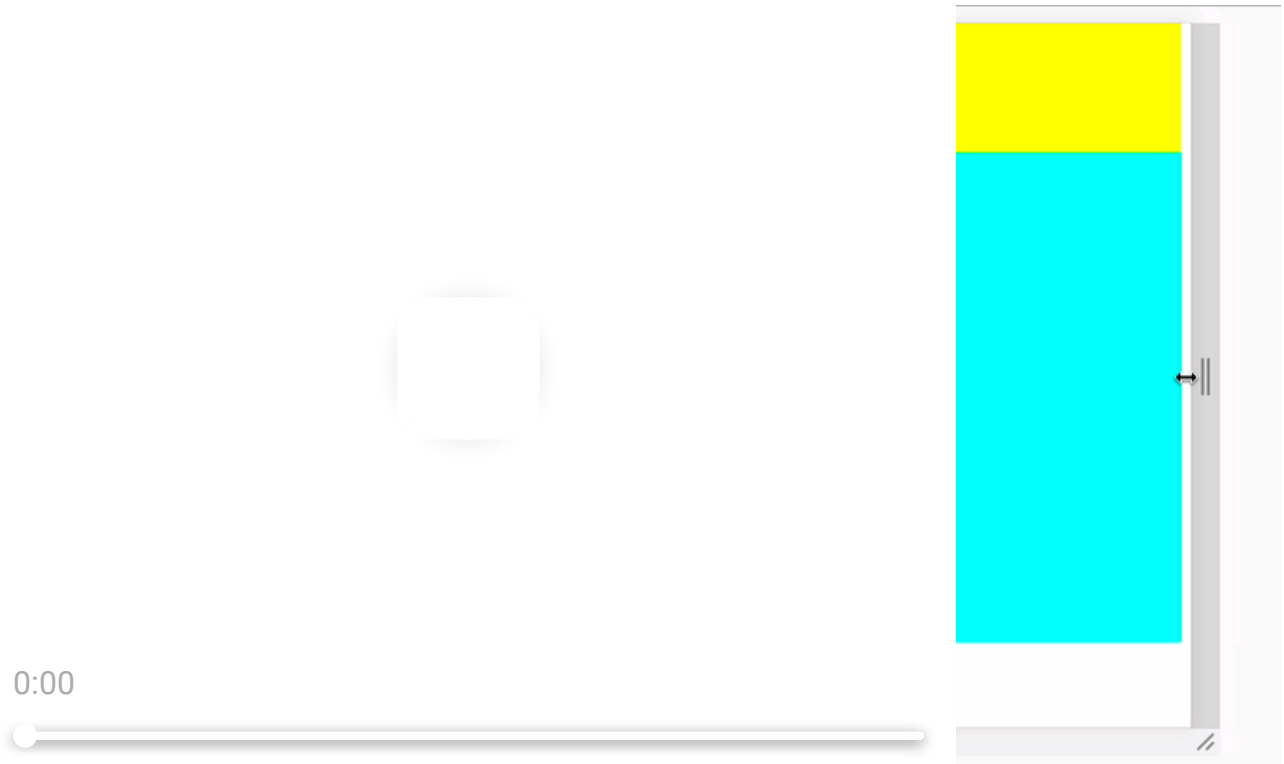
viewport, the track containing the content (yellow in the picture above) will at least fill all available space, but will grow (and make the document scroll) if that's necessary.

For the columns we want to take a more dynamic approach: we want both nav and content to grow (and shrink), but we want nav to never shrink below 200px and we want content to be larger than nav. In Flexbox we would use flex-grow and flex-shrink, but in Grid it's a little different:

```
body {
  display: grid;
  grid-template-rows: 150px auto 100px;
  grid-template-columns: minmax(200px, 3fr) 9fr;
}
```

We are defining 2 columns. The first column is defined using the **`minmax()`** function, which takes 2 values: The minimum and the maximum size of that track. (It's like **`min-width`** and **`max-width`** in one.) The minimum width is, as we discussed before, 200px. The maximum width is **`3fr`**. **`fr`** is a grid-specific unit that allows you distribute available space to the grid elements. (fr probably stands for "fraction unit", but might also mean "free unit" soon.) Our values here mean that both columns will grow to fill the screen, but the content column will always be 3 times as wide as the nav column (provided the nav column stays wider than 200px).

While the *placement* of our grid items is not correct yet, the *size* of the rows and columns behaves correctly and yields the behavior we were aiming for:

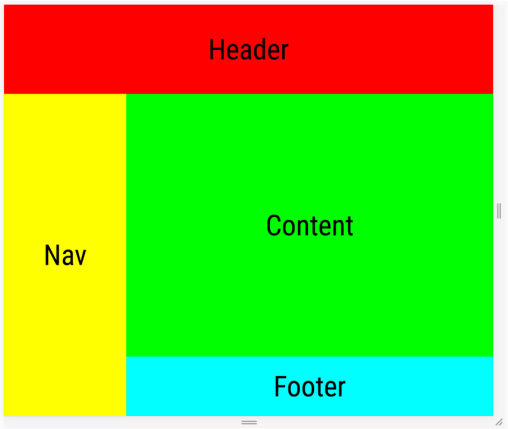


Placing the items

One of the most powerful features of Grid is to be able to place items without regard to the DOM order. (Although, because screen readers navigate the DOM, we highly recommend that to be properly accessible you should be mindful of how you reorder elements.) If no manual placement is done, the elements are placed in the Grid in DOM order, arranged left to right and top to bottom. Each element occupies one *cell*. The order in which the grid is filled can be changed using `grid-auto-flow`.

So, how do we place elements? Arguably the easiest way to place grid items is by defining which columns and rows they cover. Grid offers two syntaxes to do this: In the first syntax you define start and end points. In the second one you define a start point and a span:

```
header {
  grid-column: 1 / 3;
}
nav {
  grid-row: 2 / span 2;
}
```



We want our header to start in the first column and end *before* the 3rd column. Our nav should start in the second row and span 2 rows in total.

Technically, we are done implementing our layout, but I want to show you a few convenience features that Grid provides for you to make placement easier. The first feature is that you can name your track borders and use those names for placement:

```
body {
  display: grid;
  grid-template-rows: 150px [nav-start] auto 100px [nav-end];
```

```
  grid-template-columns: [header-start] minmax(200px, 3fr) 9fr [header-end];
}
header {
  grid-column: header-start / header-end;
}
nav {
  grid-row: nav-start / nav-end;
}
```

The code above will yield the same layout as the code before.

Even more powerful is the feature of naming entire regions in your grid:

```
body {
  display: grid;
  grid-template-rows: 150px auto 100px;
  grid-template-columns: minmax(200px, 3fr) 9fr;
  grid-template-areas: "header header"
                       "nav    content"
                       "nav    footer";
}
header {
  grid-area: header;
}
nav {
  grid-area: nav;
}
```

grid-template-areas takes a string of space-separated names, allowing the developer to give each cell a name. If two adjacent cells have the same name, they are going to be coalesced to the same area. This way you can provide more semantics to your layout code and make media queries more intuitive. Again, this code will generate the same layout as before.

Is there more?

Yes, yes there is, way too much to cover in a single blog post. Rachel Andrew, who is also a GDE, is an Invited Expert in the CSS Working Group and has been working with them from the very start to make sure Grid simplifies web design. She even wrote a book on it. Her website Grid By Example is a valuable resource for getting familiar with Grid. Many people think Grid is a revolutionary step for web design and it is now enabled by default in Chrome so you can start using it today.

Last updated July 2, 2018.