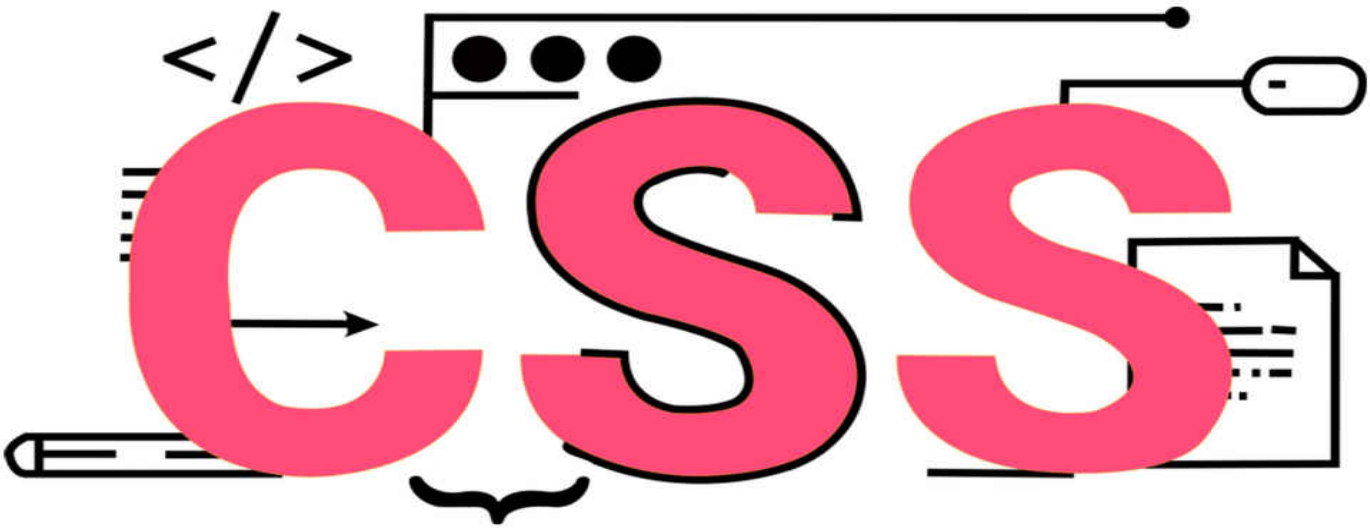
HANDS-ON PROJECTS ON



GRID LAYOUT

FRANCINE LEON

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
➥design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

TIPS, NOTES, AND WARNINGS

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1: Redesigning a Site to Use CSS Grid Layout

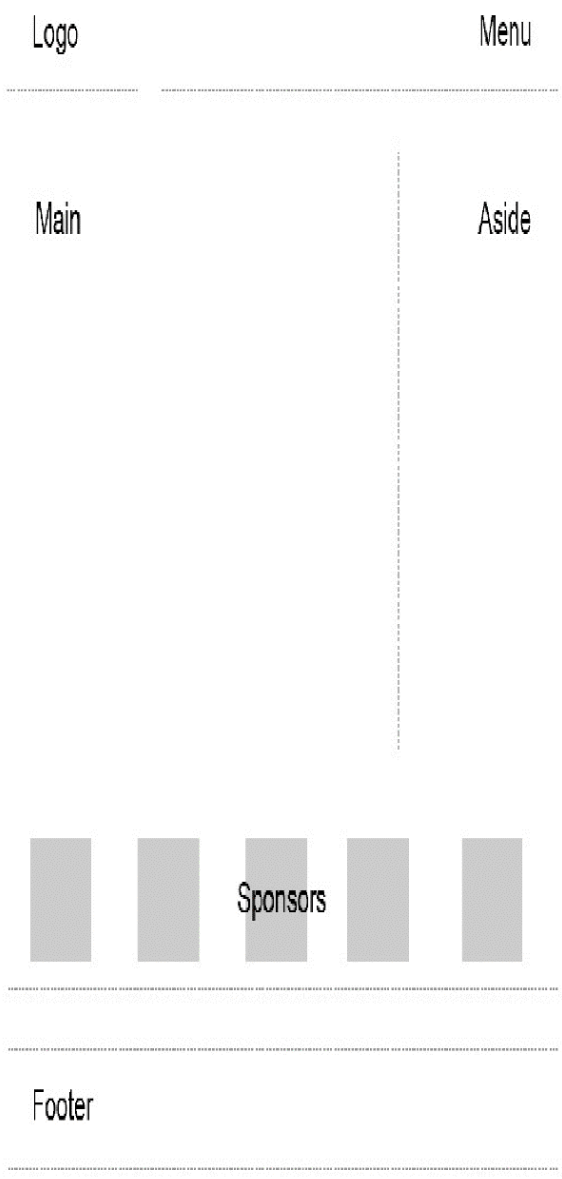
BY ILYA BODROV

CSS Grid is a new hot trend in web development these days. Forget about table layouts and floats: a new way to design websites is already here! This technology introduces two-dimensional grids which define multiple areas of layout with a handful of CSS rules. Grid can make third-party frameworks such as 960gs or Bootstrap grid redundant, as you may easily do everything yourself! This feature is supported by all major browsers, though Internet Explorer implements an older version of the specification.

In this article we are going to see CSS Grid in action by creating a responsive multi-column website layout.

What We Are Going to Build

So, we were asked to create a typical website layout with a header, main content area, sidebar to the right, a list of sponsors, and a footer:


Logo
Menu
Main
Aside
Sponsors
Footer

Another developer has already tried to solve this task and came up with a solution that involves floats, `display: table`, and some clearfix hacks. We are going to refer to this existing layout as "initial".

Live Code

See the Pen Multi-Column Layout With Floats.

Until recently, floats were considered to be the best option to create such layouts. Prior to that, we had to utilize HTML tables but they had a number of downsides. Specifically, such table layout is very rigid, it requires lots of tags (`table`, `tr`, `td`, `th` etc), and semantically these tags are used to present table data, not to design layouts.

But CSS continues to evolve, and now we have CSS Grid. Conceptually it is similar to an old table layout but can use semantic HTML elements with a more flexible layout.

Planning The Grid

First things first: we need to define a basic HTML structure for our document. Before that, let's briefly talk about how the initial example works. It has the following main blocks:

- `.container` is the global wrapper that has small margins to the left and to the right.
- `.main-header` is the header that contains the `.logo` (occupies 20% of space, floats to the left) and the `.main-menu` (occupies 79% of space, floats to the right). The header is also assigned with a hacky fix to clear the floats.
- `.content-area-wrapper` wraps the main `.content-area` (occupies 66.6% of space minus `1rem` reserved for margin, floats to the left) and the `.sidebar` (occupies 33.3% of the space, floats to the

right). The wrapper itself is also assigned with a clearfix.

- `.sponsors-wrapper` contains the logos of the sponsors. Inside, there is a `.sponsors` section with the `display` property set to `table`. Each sponsor, in turn, is displayed as a table cell.
- `.footer` is our footer and spans to 100% of space.

Our new layout will be very similar to the initial one, but with one exception: we won't add the `.main-header` and `.content-area-wrapper` wrappers because the clearfixes won't be required anymore. Here is the new version of the HTML:

```
<div class="container">
   <header class="logo">
     <h1><a href="#">DemoSite</a></h1>
   </header>

   <nav class="main-menu">
     <ul>
       <li class="main-menu__item"><a href="#">Our
clients</a></li>
       <li class="main-menu__item"><a
href="#">Products</a></li>
       <li class="main-menu__item"><a
href="#">Contact</a></li>
     </ul>
   </nav>

   <main class="content-area">
     <h2>Welcome!</h2>

     <p>
       Content
     </p>
   </main>

   <aside class="sidebar">
     <h3>Additional stuff</h3>

     <ul>
       <li>Items</li>
       <li>Are</li>
       <li>Listed</li>
       <li>Here</li>
       <li>Wow!</li>
     </ul>
   </aside>

   <section class="sponsors-wrapper">
     <h2>Our sponsors</h2>

     <section class="sponsors">
```

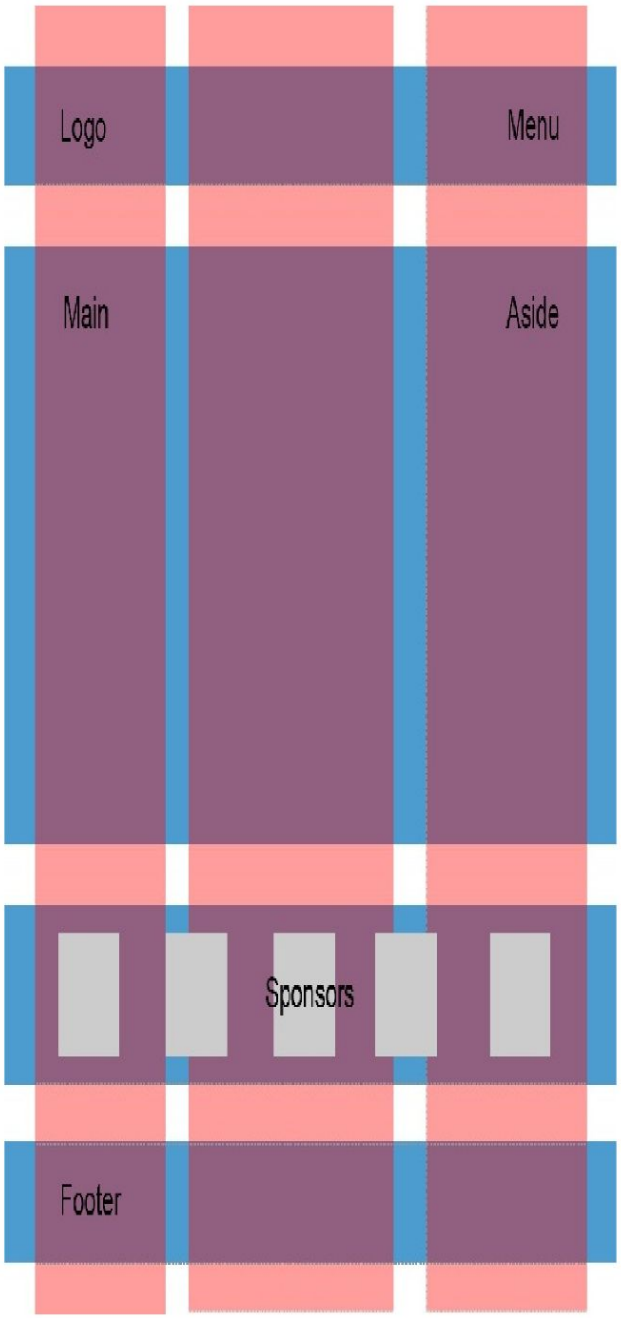
```
        <figure class="sponsor">
          <img
src="https://via.placeholder.com/150x150">
        </figure>

        <figure class="sponsor">
          <img
src="https://via.placeholder.com/200x150">
        </figure>

        <figure class="sponsor">
          <img
src="https://via.placeholder.com/100x200">
        </figure>

        <figure class="sponsor">
          <img
src="https://via.placeholder.com/100x100">
        </figure>

        <figure class="sponsor">
          <img
src="https://via.placeholder.com/200x200">
        </figure>
      </section>

    </section>

    <footer class="footer">
      <p>
        &copy; 2018 DemoSite. White&amp;amp;Sons LLC.
All rights (perhaps) reserved.
      </p>
    </footer>
  </div>
```

Note that you may utilize the `body` as the global `.container` — that's just a matter of preference in this case. All in all, we have six main areas:

1. Logo
2. Menu
3. Main content
4. Sidebar
5. Sponsors
6. Footer

Usually it is recommended to to implement mobile-first approach: that is, start from the mobile layout and then designing for larger screens. This is not necessary in this case because we are adapting an initial layout which

already falls back to a linearized view on small-screen devices. Therefore, let's start by focusing on the grid's implementation, and after that talk about responsiveness and fallback rules. So, return to our scheme and see how the grid columns can be arranged:


Logo
Menu
Main
Aside
Sponsors
Footer

So, I propose having three columns (highlighted with red color) and four rows (highlighted with blue). Some areas, like logo, are going to occupy only one column, whereas others, like main content, are going to span multiple columns. Later we can easily modify the layout, move the areas around, or add new ones.

Following the scheme, give each area a unique name. These will be used in the layout defined below:

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

.main-menu {
  grid-area: menu;
}

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

.sidebar {
  grid-area: sidebar;
}

.sponsors-wrapper {
  grid-area: sponsors;
}

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

Now set the `display` property to `grid`, define three columns and add small margins to the left and right of the main container:

```
.container {
  display: grid;
  margin: 0 2rem;
  grid-template-columns: 2fr 6fr 4fr;
}
```

`display: grid` defines a grid container and sets a

special formatting context for its children. `fr` is a special unit that means "fraction of the free space of the grid container". `2 + 6 + 4` gives us `12`, and `6 / 12 = 0.5`. It means that the middle column is going to occupy 50% of the free space.

I would also like to add some spacing between the rows and columns:

```
.container {
  // ...
  grid-gap: 2rem 1rem;
}
```

Having done this we can work with individual areas. But before wrapping up this section let's quickly add some common styles:

```
* {
  box-sizing: border-box;
}

html {
  font-size: 16px;
  font-family: Georgia, serif;
}

body {
  background-color: #fbfbfb;
}

h1, h2, h3 {
  margin-top: 0;
}

header h1 {
  margin: 0;
}

main p {
  margin-bottom: 0;
}
```

Good! Now we can proceed to the first target which is going to be the header.

Designing the Header

Our header occupies the first row that should have a specific height set to `3rem`. In the initial layout this is solved by assigning the `height` property for the header wrapper:

```
.main-header {
   height: 3rem;
 }
```

Also note that the logo and the menu are vertically aligned to the middle which is achieved using the `line-height` trick:

```
.logo {
   // ...
   height: 100%;
   line-height: 3rem;
 }
```

With CSS Grid, however, things are going to be simpler: we won't require any CSS hacks.

Start by defining the first row:

```
.container {
   // ...
   grid-template-rows: 3rem;
 }
```

Our logo should occupy only one column, whereas the menu should span two columns. We can express our intent with the help of `grid-template-areas` property which references the `grid-area` names assigned above:

```
.container {
   // ...
```

```
    grid-template-areas:
            "logo menu menu";
  }
```

What is going on here? Well, by saying `logo` only once we are making sure that it occupies only one, the left-most column. `menu menu` means that the menu occupies two columns: the middle and the right-most one. See how straightforward this rule is!

Now align the logo on the Y axis:

```
.logo {
    grid-area: logo;
    align-self: center;
  }
```

The menu should be centered vertically and pulled to the right:

```
.main-menu {
    grid-area: menu;
    align-self: center;
    justify-self: end;
  }
```

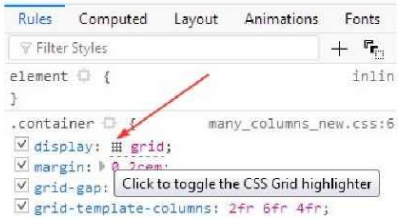
Our menu is built with the `ul` and `li` tags, so let's also style them a bit by removing markers, nullifying margins/paddings, and setting turning the menu to a flex container:

```
.main-menu ul {
    margin: 0;
    padding: 0;
    display: flex;
  }

  .main-menu__item {
    list-style-type: none;
    padding: 0;
    font-size: 1.1rem;
    margin-right: 0.5rem;
  }
```

```
.main-menu .main-menu__item:last-of-type {
  margin-right: 0;
}
```

That's pretty much it. To observe the result, I am going to use Firefox browser with a handy CSS Grid highlighter tool enabled (there are similar tools for other browsers available: for instance, Gridman for Chrome). To gain access to this tool, press `F12` and select the `.container` element which should have a `grid` label:

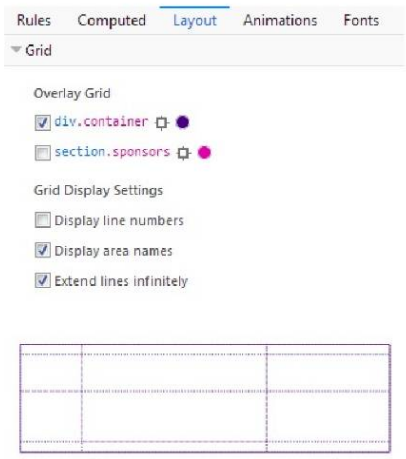
After that, proceed to the CSS rules tab, and find the `display: grid` property. By pressing on the small icon next to the `grid` value you may enable or disable the highlighter:



Here is the result:

Highlighter displays all your rows and columns, as well as the margins between them and the areas' names. You

may customize the output inside the Layout section which also lists all the grids on the page:



So, we've dealt with the header, therefore let's proceed to the main content area and the sidebar.

Main Content and Sidebar

Our main content area should span two columns, whereas the sidebar should occupy only one. As for the row, I would like its height to be set automatically. We can update the `.container` grid accordingly:

```
.container {
   //  ...
   grid-template-rows: 3rem auto;
   grid-template-areas:
           "logo menu menu"
           "content content sidebar";
 }
```

I'd like to add some padding for the sidebar to give it some more visual space:

```
.sidebar {
   grid-area: sidebar;
   padding: 1rem;
}
```

Here is the result as viewed in Firefox's Grid tool:

Sponsors

The sponsors section should contain five items with equal widths and heights. Each item, in turn, will have one image.

In the initial layout this block is styled with `display: table` property, but we won't rely on it. Actually, the sponsors section may be great candidate for applying CSS grid as well!

First of all, tweak the `grid-template-areas` to include the `sponsors` area:

```
.container {
   // ...
   grid-template-areas:
           "logo menu menu"
           "content content sidebar"
           "sponsors sponsors sponsors"
}
```

Now turn the `.sponsors` section to a grid as well:

```
.sponsors {
   display: grid;
 }
```

As long as we need five items with equal widths, a `repeat` function can be utilized to define the columns:

```
.sponsors {
   display: grid;
   grid-template-columns: repeat(5, 1fr);
 }
```

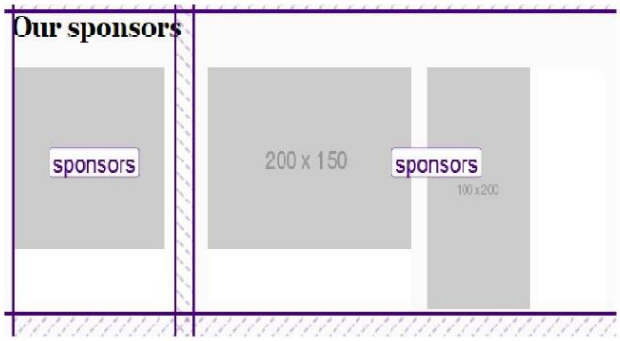
As for the row, its height should be set automatically. The gap between the columns should be equal to `1rem`:

```
.sponsors {
   display: grid;
   grid-template-columns: repeat(5, 1fr);
   grid-template-rows: auto;
   grid-column-gap: 1rem;
 }
```

Style each item:

```
.sponsor {
   margin-left: 0;
   margin-right: 0;
   background-color: #fff;
   border-radius: 0.625rem;
 }
```

Here is the intermediate result:



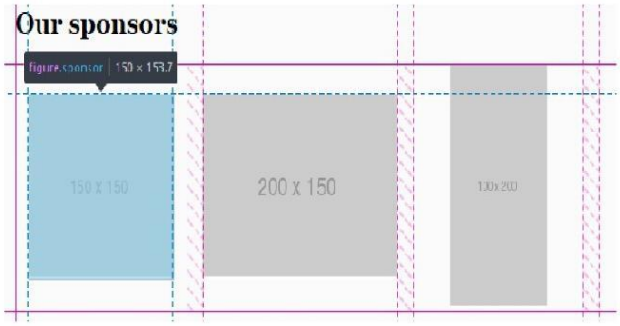
This example illustrates that you may safely nested grids, and this will work without any problems. Another solution might be using Flexbox, but in this case the sponsors may wrap if there is not enough width for them.

Now I would like to center the images both vertically and horizontally. We might try doing the following:

```
.sponsor {
   place-self: center;
 }
```

`place-self` aligns the element on X and Y axises. It is a shorthand property to `align-self` and `justify-self`.

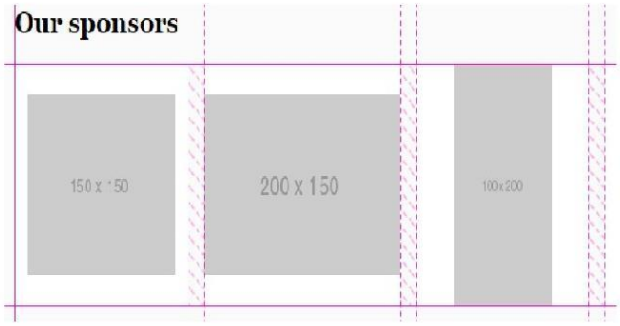
The images will indeed be aligned but unfortunately the white background is gone. This is because each `.sponsor` now has width and height equal to the image's dimensions:



It means that we need a different approach here, and one of the possible solutions is to employ Flexbox:

```
.sponsor {
   // ...
   display: inline-flex;
   align-items: center;
   justify-content: center;
 }
```

Now everything is displayed properly, and now we know that Grid plays nicely with Flexbox:



Footer

Our last section is footer, and it is actually the simplest section. All we have to do is span it to all three columns:

```
.container {
   // ...
   grid-template-areas:
           "logo menu menu"
           "content content sidebar"
           "sponsors sponsors sponsors"
           "footer footer footer";
 }
```

Basically, the layout is finished! However, we are not done yet: the site also has to be responsive. So, let's take care of this task in the next section.

Making the Layout Responsive

Having CSS Grid in place, it is actually very easy to introduce responsiveness, because we can quickly reposition the areas.

LARGE SCREENS

Let's start with large screens (in this article I'll be sticking to the same breakpoints as defined in Bootstrap 4). I would like to decrease horizontal margin of the main container and the gap between individual sponsors:

```
@media all and (max-width: 992px) {
   .container {
     margin: 0 1rem;
   }

   .sponsors {
     grid-column-gap: 0.5rem;
   }
 }
```

MEDIUM SCREENS

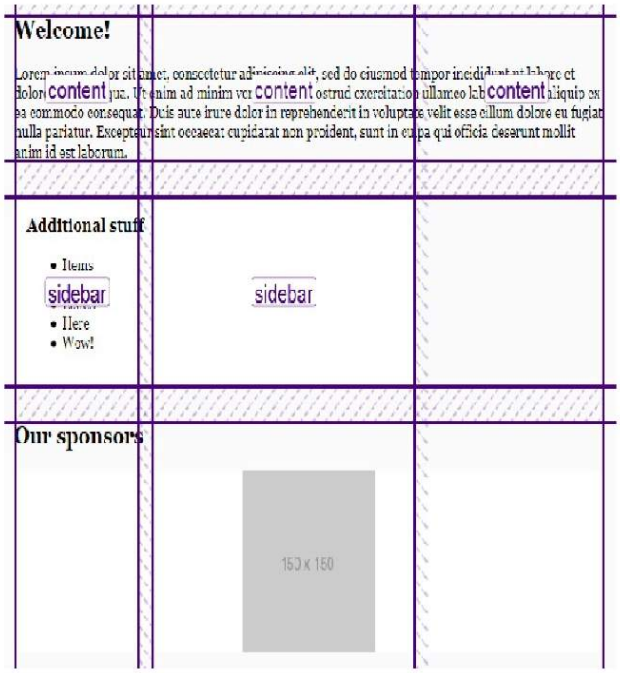
On the medium screens, I would like the main content area and the sidebar to occupy all three columns:

```
@media all and (max-width: 768px) {
   .container {
     grid-template-areas:
             "logo menu menu"
             "content content content"
             "sidebar sidebar sidebar"
             "sponsors sponsors sponsors"
             "footer footer footer";
   }
 }
```

Let's also decrease font size and stack the sponsors so they are displayed one beneath another. The gap between the columns should be zero (because actually there will be only one column). Instead, I'll set a gap between the rows:

```
@media all and (max-width: 768px) {
   // ...
   html {
     font-size: 14px;
   }

   .sponsors {
     grid-template-columns: 1fr;
     grid-column-gap: 0;
     grid-row-gap: 1rem;
   }
 }
```

This is how the site looks on medium screens now:



SMALL SCREENS

On small screens we are going to display each area on a separate row, which means that there will be only one column now:

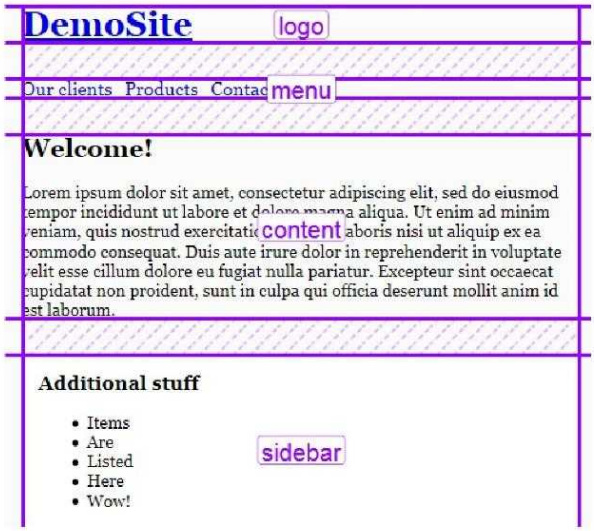
```
@media all and (max-width: 540px) {
   .container {
     grid-template-columns: 1fr;
     grid-template-rows: auto;
     grid-template-areas:
             "logo"
             "menu"
             "content"
             "sidebar"
             "sponsors"
             "footer";
   }
 }
```

The menu should not be pulled to the right in this case,

and we also don't need the gap between the columns:

```
@media all and (max-width: 540px) {
   .container {
     // ...
     grid-gap: 2rem 0;
   }

   .main-menu {
     justify-self: start;
   }
 }
```

The job is done:



Note that you may even rearrange the grid items easily for various screens. Suppose we would like to put the menu to the bottom on small screens (so that the visitors do not have to scroll up after they've finished reading material on the page). To do that, simply tweak the `grid-template-areas`:

```
@media all and (max-width: 540px) {
   .container {
     // ...
```

```
      grid-template-areas:
              "logo"
              "content"
              "sidebar"
              "sponsors"
              "footer"
              "menu";
    }
  }
```

WITHOUT MEDIA QUERIES?..

It is worth mentioning that we can make the sponsors block responsive without any media queries at all. This is possible with the help of `auto-fit` property and `minmax` function. To see them in action, tweak the styles for the `.sponsors` like this:

```
.sponsors {
   // ...
   grid-template-columns: repeat(auto-fill,
minmax(200px, 1fr));
 }
```

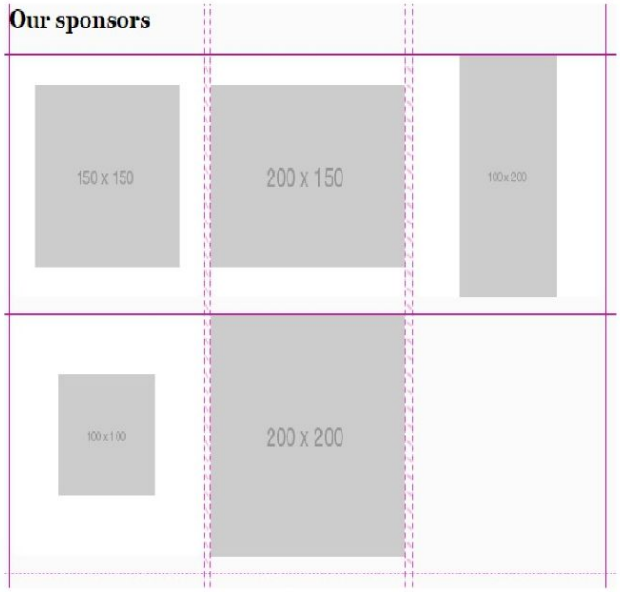
`repeat` function, as you already know, repeats the columns as many times as necessary.

`auto-fill` means "fill the row with as many columns as possible". If there is not enough space for the column, it will be placed to the next line.

`minmax` allows us to specify the minimum and maximum value for the columns' widths. In this case each column should span 1 fraction of free space, but no less than 200 pixels.

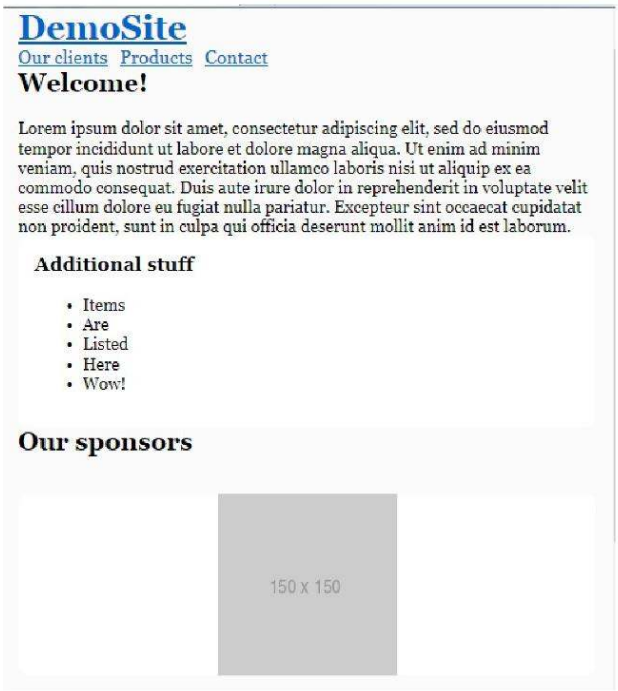
Al this means that on smaller screens the columns may be shrinked down to at most `200px` each. If there is still not enough space, one or multiple columns will be placed to a separate line. Here is the result of applying the above

CSS rules:



Fallbacks

Unfortunately, CSS Grid is not yet fully supported by all browsers, and you may guess which one is still implementing an older version of the specification. Yeah, it's Internet Explorer 10 and 11. If you open the demo in this browser, you'll see that the grid does not work at all, and the areas are simply stacked:



Of course, this is not the end of the world, as the site is still usable, but let's add at least some fallback rules. The good news is that if the element is floated and also has grid assigned, then the grid takes precedence. Also, the `display`, `vertical-align`, and some other properties also have no effect on grid items, so let's take advantage of that fact.

The stacked menu looks nice as is, but the sidebar should be probably placed next to the main content, not below it. We can achieve this by using `display: inline-block`:

```
.content-area {
   display: inline-block;
   vertical-align: top;
 }
```

```
 .sidebar {
   display: inline-block;
   vertical-align: top;
 }
```

In all browsers that support grid, these properties will have no effect, but in IE they'll be applied as expected. One more property we need to tweak is the `width`:

```
.content-area {
   width: 69%;
   display: inline-block;
   vertical-align: top;
 }

 .sidebar {
   width: 30%;
   display: inline-block;
   vertical-align: top;
 }
```

But having added these styles, our grid layout will now look much worse, because the `width` property is not ignored by grid items. This can be fixed with the help of `@supports` CSS query. IE does not understand these queries, but this is not needed anyways: we'll use it to fix the grid!

```
@supports (display: grid) {
   .content-area, .sidebar  {
     width: auto;
   }
 }
```

Now let's take care of the sponsor items and add some top margin for each block:

```
.sponsor {
   vertical-align: middle;
 }

 .main-menu, .content-area, .sidebar, .sponsors-
wrapper, .footer  {
   margin-top: 2rem;
```

```
  }
```

We don't need any top margin when the grid is supported, so nullify it inside the `@supports` query:

```
@supports (display: grid) {
   // ...
   .main-menu, .content-area, .sidebar, .sponsors-
wrapper, .footer, .sponsor {
     margin-top: 0;
   }
 }
```

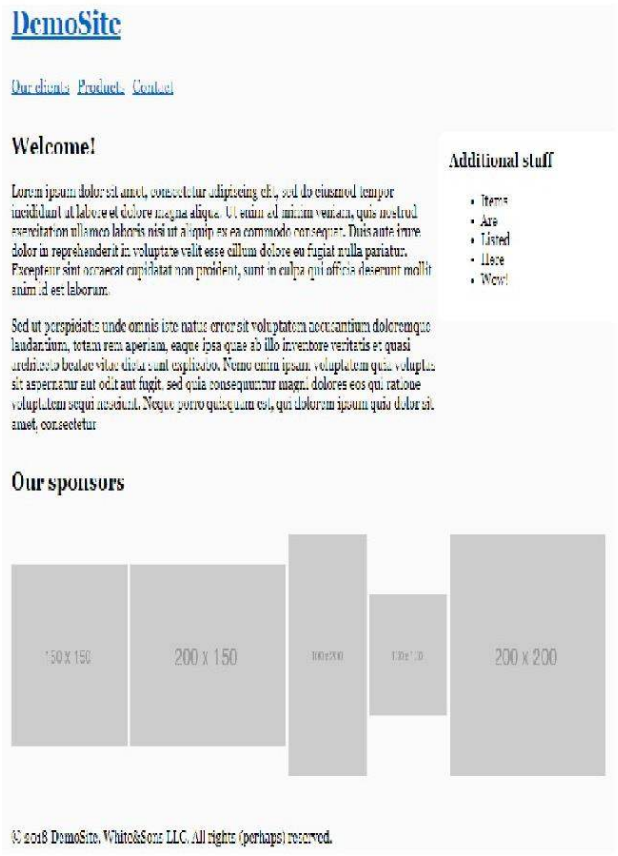
Lastly, let's add some responsiveness for IE. We'll simply stretch main content, sidebar, and each sponsor to full width on smaller screens:

```
@media all and (max-width: 760px) {
   .content-area, .sidebar {
     display: block;
     width: 100%;
   }

   .sponsor {
     width: 100%;
     margin-top: 1rem;
   }
 }
```

Don't forget to fix the sponsor's width for the browsers that support grid:

```
@supports (display: grid) {
   // ..

   .sponsor {
     width: auto;
   }
 }
```

Here is how the site looks in Internet Explorer now:



Live Code

See the Pen **Multi-Column Layout With Floats**.

Conclusion

In this article we have seen CSS Grid in action and utilized it to redesign an existing float-based layout. Comparing these two solutions, we can see that the HTML and CSS code of the "grid" solution is smaller (not counting the fallbacks, of course), simpler and more expressive. With the help of the `grid-template-`

`areas` property it is easy to understand how individual areas are laid out, and we may quickly reposition them or adjust their sizes. On top of that, we don't need to rely on various hacky tricks (like clearfix, for instance).

So, as you see, CSS Grid is a great alternative to floats, and it is production-ready. You may need to provide some fallback rules for Internet Explorer (that implements an older version of the specification), but as you see they are not very complex and in general the site is still usable even without any backwards compatibility at all.

Chapter 2: Redesigning a Card-based Tumblr Layout with CSS Grid

BY GIULIO MAINARDI

In this tutorial we're going to re-implement a grid-based design concept inspired by the What're you into? Tumblr page, where the user can select a set of topics to tailor her recommended content.

Only the visual design of the grid is executed, not the selection functionality, as shown in the Pen we will be building:

Live Code

See the Pen MBdNav.

The main goal is to implement the design with CSS Grid, but a fallback layout with floats is outlined in the Support section below.

Markup

Essentially, the page content consists of list of cards:

```
<ul class="grid">
  <li class="card header">
    <h1>Which foods do you like?</h1>
    <p>Tell us what you crave for and we'll get
you the tasty bits</p>
  </li>
  <li class="card">
    <a href="...">
      <h2>Pizza</h2>
      <img src="..." alt="A salami pizza">
    </a>
  </li>
  <!-- ... -->
</ul>
```

A card that represents a topic proposed to the user(food in our example) has a title and an illustrative image, both wrapped in a link element. Other s could be adopted, see for instance the excellent article on the card component on Inclusive Components, where the pros and cons of such alternatives are analyzed.

Structural Layout

In this section, the foundations of the grid design will be implemented. The next section will style the cards. This Pen shows the bare-bones layout using placeholders for grid items. Run it on a browser that supports CSS Grid.

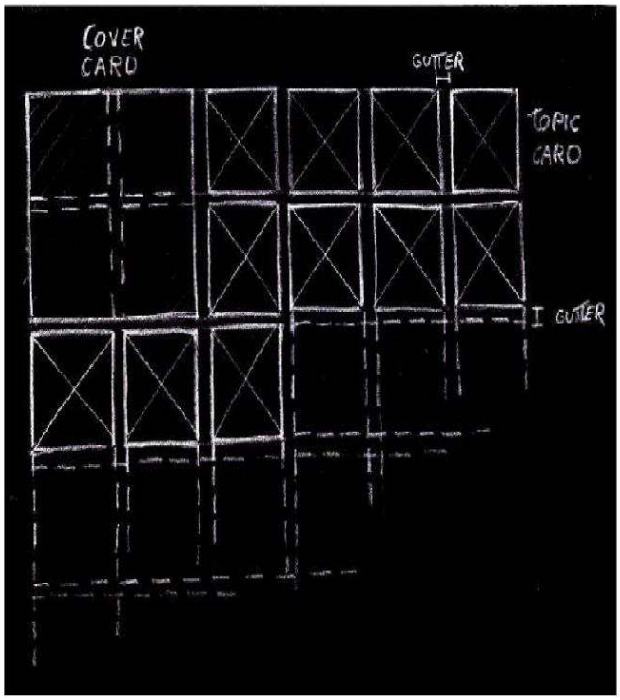
Live Code

See the Pen JBqgGm.

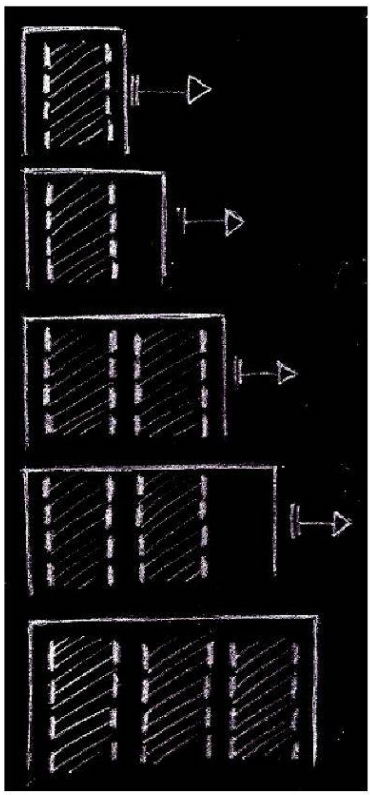
Before going ahead with the code, it is important to specify the features and the responsive behavior of the grid. Let's try to write down some properties it must satisfy.

DESIGN SPECS

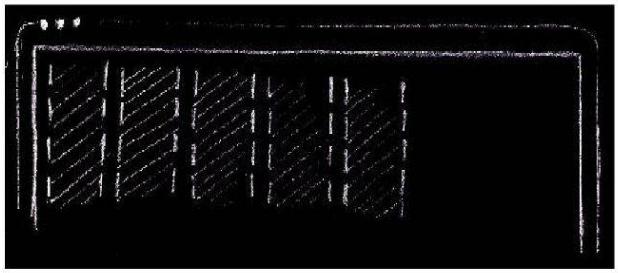
Two kinds of cards are featured in the design: a series of topic cards, and an introductory cover card. We arrange them on an underlying grid composed of square cells of fixed size. Each topic card occupies just one of these cells, while the cover spans a larger area of adjacent cells, whose extent depends on the viewport width. Furthermore, rows and columns are separated by the same fixed size gutter.



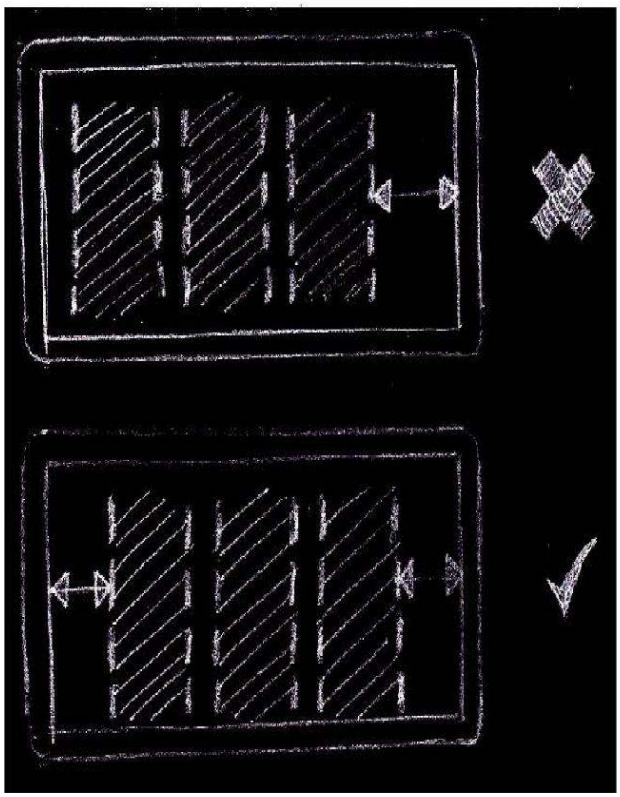
The grid has as many (fixed sized)columns as they fit into the viewport:



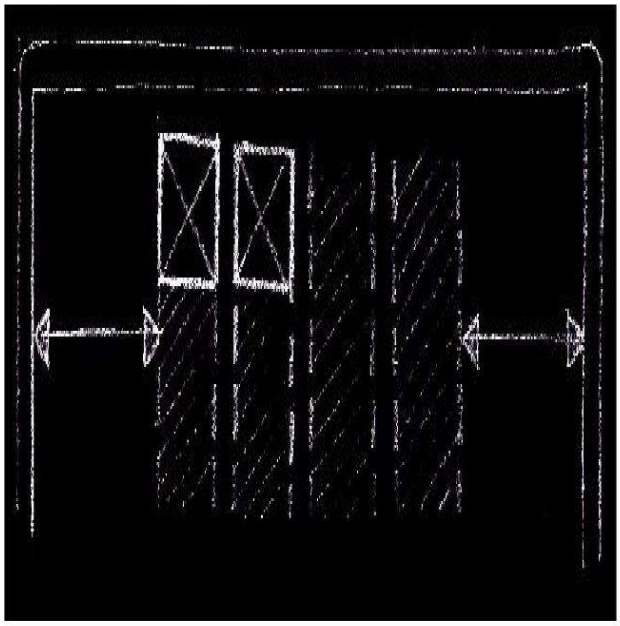
But we don't want a zillion columns on large screens, so let's limit the maximum number of columns:



The columns are always horizontally centered in the viewport.



Only the columns are centered, not the grid items. This means that the cards on an incomplete row are aligned to the left of the grid, not at the center of the viewport:



Check out these requirements in the above Pen. Also, it's useful to inspect the layout with the CSS Grid tools provided by some browsers, such as the Firefox's Grid Inspector.

Keeping in mind this checklist, we can fire up our favorite development environment and start coding.

IMPLEMENTATION

Let's introduce a couple of Sass global variables to represent the layout parameters defined in the specs, namely:

- `$item-size` for the size of the side of the grid cells
- `$col-gutter` for the gutter between the tracks of the grid
- `$vp-gutter` for a safety space to leave between the grid items and the viewport edges

- `$max-cols` for the maximum number of columns the grid can have

We could use CSS custom properties for these variables, avoiding the need of a preprocessor and allowing us to edit them with the in-browser development tools and watch the changes happen instantly. But we are going to use these values even for a fallback layout suitable for older browsers, where CSS variables are not supported. Moreover, we use expressions with these values even in media query selectors, where custom properties and the `calc()` function are not fully available even on recent browsers.

```
$item-size: 210px;
$col-gutter: 10px;
$vp-gutter: $col-gutter;
$max-cols: 5;
```

We must establish a grid formatting context on the grid element:

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

The `grid-gap` property separates the grid tracks by the specified amount of space. But these gutters are only inserted between the tracks and not before the first track and after the last one. An horizontal padding on the grid container prevents the columns from touching the viewport edges:

```
.grid {
  grid-gap: $col-gutter;
  padding: 0 $vp-gutter;
}
```

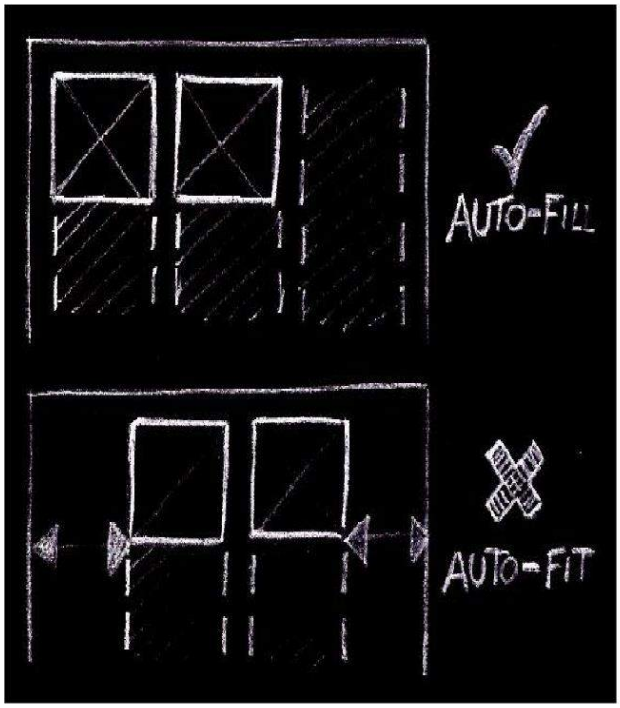
The columns of the grid can be defined with the `grid-template-columns` property and the `repeat` function with the `auto-fill` value as the repetition number and

the `$item-size` variable for the track list argument:

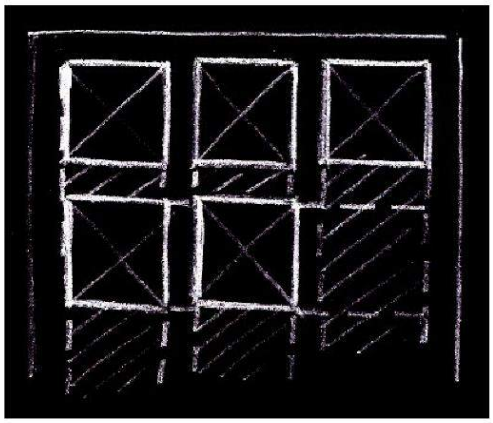
```
.grid {
  grid-template-columns: repeat(auto-fill, $item-
size);
}
```

This tells the browser to fill the grid container(the `.grid` element) width with as many fixed size columns as possible, keeping account of the vertical gutters.

It is worth pointing out that we could have used the `auto-fit` mode, and in many combinations of viewport sizes and number of grid items we could not tell the difference with `auto-fill`. But when there are only a few items in the grid, with just an incomplete row, with `auto-fit` the items would be centered, instead of starting from the left of the grid, as detailed in the design specs. This happens because while with `auto-fill` the grid has always as many columns as possible, with `auto-fit` empty columns are removed, and the centering of the remaining columns places the items at the center of the viewport:



If the first row of the grid is complete, no columns are removed and there is no difference between the two modes:



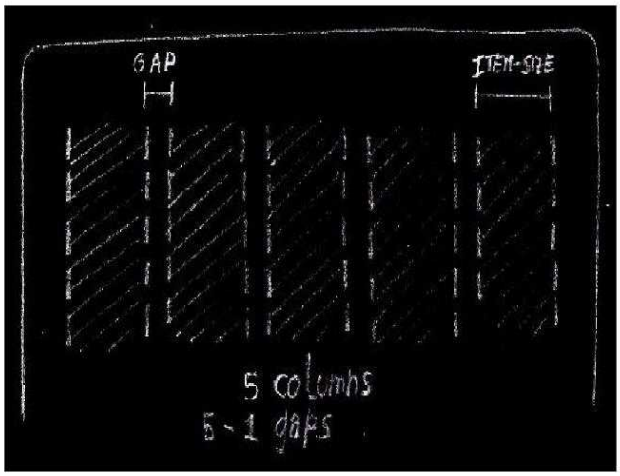
Back to the grid columns. Up to this point, the number of columns had no limit, it can arbitrarily grow as the viewport width increases. But according to the spec, the grid must have a maximum number of columns. It's possible to fix this with the `max-width` property:

```
.grid {
  max-width: grid-width($max-cols);
}
```

`grid-width` is a custom Sass function that returns the width of a grid with n columns:

```
@function grid-width($num-cols) {
  @return $num-cols * $item-size + ($num-cols - 1)
* $col-gutter;
}
```

The first multiplication accounts for the size required by the columns, while the second one represents the space required by the gutters.



According to the specs, the grid must be always horizontally centered. We can combine the old auto margins trick with `justify-content` to accomplish

this task:

```
.grid {
  justify-content: center;
  margin: 40px auto;
}
```

`justify-content` centers the columns when there is available space left inside the grid container. This happens when the container bleeds from one viewport edge to the other. The lateral auto margins centers the `.grid` container itself when it has reached its maximum width.

Now for the rows. They are not explicitly specified, as done with the columns. Rather, they are implicitly added by the browser as needed, and we just tell it their size with the `grid-auto-rows` property. Reusing the `$item-size` variable, each grid cell is shaped like a square, as per the specs:

```
.grid {
  grid-auto-rows: $item-size;
}
```

Let's move on by sizing the cards. On small viewports, when the grid configures itself on a single column, the cover card spans only a grid cell, while when there are two or more columns, it must span a 4x4 grid area:

```
@include when-n-cols(2) {
  .grid .header {
    grid-row: span 2;
    grid-column: span 2;
  }
}
```

`when-n-cols()` is a Sass mixin to express a media query suitable for a grid with the given number of columns:

```
@mixin when-n-cols($n) {
  @media screen and (min-width: #{grid-width($n) +
2 * $vp-gutter + $scrollbar-size}) {
    @content;
  }
}
```

The CSS rules represented by `@content` are active whenever the viewport width is equal or greater than the width of a grid with `$n` columns plus the two safety spaces to separate the grid items from the viewport edges. `$scrollbar-size` is just an upper bound on the size of a vertical scrollbar, to account for the fact that the width reported in the media query is the entire viewport width, including an eventual vertical scrollbar.

Regarding the topic cards, there is nothing to do, because the default Grid behavior makes them the same size as their assigned grid cells.

Ok, we got it! Refer to the structural Pen at the beginning of this section to see all these code snippets put together in the complete code.

The Cards

Here we build the cards, their inner content to be more precise.

Let's address the topic cards first. To make them clickable, the link element is expanded to fill the entire card area:

```
.grid .card a {
  display: block;
  width: 100%;
  height: 100%;

  position: relative;
}
```

Then we make sure the card image covers all the card surface:

```
.grid .card img {
  display: block;
  width: 100%;
  height: 100%;
  object-fit: cover;
}
```

In the example, the thumbnails have a square aspect ratio, therefore they scale nicely inside their grid items. To handle arbitrary image sizes, `object-fit: cover` scales(preserving the aspect ratio) and eventually clips the image to fit inside the container.

It's the turn of the card title:

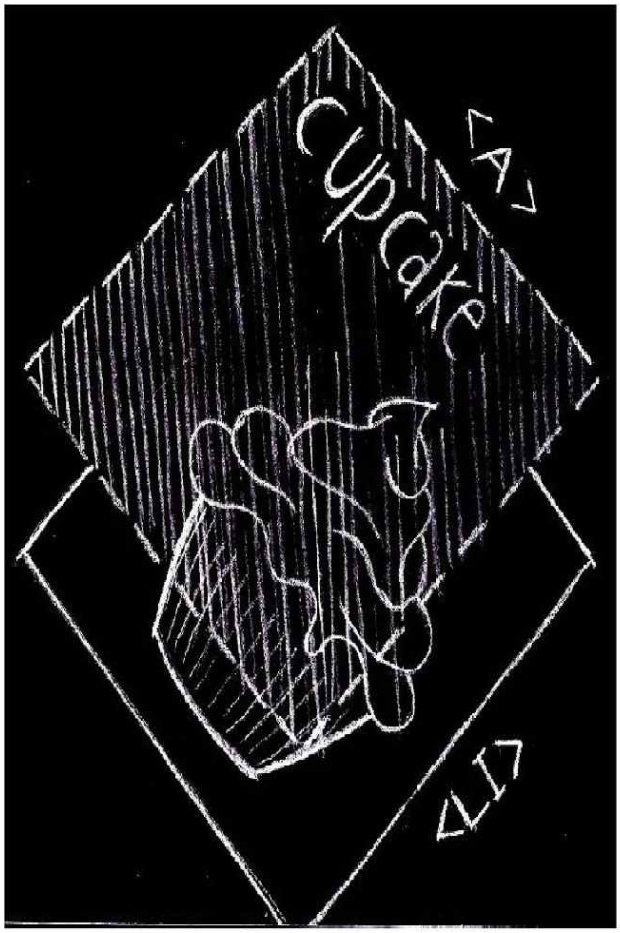
```
.grid .card h2 {
  margin: 0;

  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;

  padding: 10px;

  text-decoration: none;
  font-family: Raleway, sans-serif;
  font-size: 1em;
  letter-spacing: 1px;

  color: #fff;
}
```

With the absolute positioning the heading element is removed from the flow and positioned above the image, at the top left corner of the card.

In order to improve the contrast between the label and an arbitrary underlying card image, a partially transparent layer is sandwiched between these two graphic elements. Here I'm using the same exact

technique employed on the original Tumblr page, where this overlay consists in a radial gradient which starts completely transparent at the center of the card and ends with a partially opaque black towards the borders, in a circular fashion, giving the image a sort of subtle spotlight effect. Let's render this layer as a background image of the card link, which it has just been extended to cover all the card surface:

```
.grid .card h2 {
  background-image: radial-gradient(ellipse at
center, transparent 0, rgba(0,0,0, .36) 100%);
}
```



As for the cover card, plenty of techniques could be used here, but let's keep it simple. The card features two centered blocks of text. With a side padding their horizontal extent is limited and with `text-align` their content is centered. After that, the blocks are vertically centered just by pushing them down with a bit of top padding applied to the card container.

```
.grid .header {
  box-sizing: border-box;
```

```
  text-align: center;
  padding-top: 23px;

  font-size: 1.6875em;
  line-height: 1.3;

  background: radial-gradient(ellipse at center,
transparent 0, rgba(0,0,0, 0.48) 100%) hsla(0, 0%,
27%, 1);
}

.grid .header h1 {
  margin-top: 0;
  margin-bottom: 0.67em;

  font-family: 'Cherry Swash', cursive;
  text-transform: capitalize;
  font-size: 1em;
  font-weight: normal;

  padding-left: 28px;
  padding-right: 28px;
}

.grid .header p {
  margin: 0;

  font-family: 'Raleway', sans-serif;
  font-size: 0.52em;

  padding-left: 34px;
  padding-right: 34px;
}
```

With a media query the font size is increased and the padding adjusted when the grid displays two ore more columns:

```
@include when-n-cols(2) {
  .grid .header {
    font-size: 2.5em;

    padding-top: 100px;
  }

  .grid .header h1 {
    padding-left: 80px;
    padding-right: 80px;
  }

  .grid .header p {
    padding-left: 120px;
    padding-right: 120px;
  }
}
```

It's the time to add some interactivity to the topic cards. They must reduce the size on hover:

```
.grid .card:hover {
  transform: scale(0.95);
}

.grid .header:hover {
  transform: none;
}
```

With a CSS transition this change of visual state is smoothed out:

```
.grid .card {
  transition-property: transform;
  transition-duration: 0.3s;
}
```

The cards can be navigated with a keyboard, so why don't customize their focus style to match the same look&feel of the mouse over? We must scale a .card container when the A link inside it receive the focus. Mmmhh...we need to style an element only when one of its children get the focus...wait a moment...there's the :focus-within pseudo-class for that:

```
.grid .card a:focus {
  outline: none;
}
.grid .card:focus-within {
  transform: scale(0.95);
}
```

First the default link focus style is reset, and then the same transform as before is used when the card(its link) gets the focus.

Support

CSS GRID

Nowadays CSS Grid has a wide support, but what can we do with the other browsers? In the demo pen there is a fallback layout, implemented with floats, that behaves exacly as the Grid layout. Let's have a quick look at how it works and interacts with the Grid implementation.

The cards are sized, floated to the left and have a bottom margin for the horizontal grid gutter:

```
.card {
  float: left;
  width: $item-size;
  height: $item-size;
  margin: 0;
  margin-bottom: $col-gutter;
}
```

At this point, we could just set a `max-width` on the `.grid` container to limit the number of columns on large screen, use auto margins to center the grid, and the layout would be almost the same as the Grid one, but with the important difference than when the container doesn't reach its maximum width, the cards are aligned to the left of the viewport. This is a fallback layout for browsers not supporting CSS Grid, so we could be happy with this. Else, we could go on and add some more code to fix this difference. Let's have a try.

We set the width of the `.grid` container when there is only one column, center it into the viewport, and separate it from the screen edges:

```
.grid {
  width: grid-width(1);
  margin: 40px auto;
  padding-left: $vp-gutter;
  padding-right: $vp-gutter;
}
```

Note how we reused the `grid-width` Sass function introduced above.

Now, reusing the media query mixin, we define the width of the `.grid` container when there are 2 columns:

```
@include when-n-cols(2) {
  width: grid-width(2);
  .card:nth-child(2n) {
    margin-right: $col-gutter;
  }
  .header {
    $header-size: grid-width(2);
    width: $header-size;
    height: $header-size;
  }
}
```

We also doubled the size of the header card and assigned the right margin for the grid horizonal gutter.

This pattern is repeated for a grid of 2, 3, ..., `$max-cols` columns, taking care of resetting and assigning the `margin-right`'s of the proper cards. For instance, when the grid has three columns:

```
@include when-n-cols(3) {
  width: grid-width(3);
  .card {
    margin-right: $col-gutter;
  }
  .card:nth-child(2),
  .card:nth-child(3),
  .card:nth-child(3n + 6) {
    margin-right: 0;
  }
}
```

Please refer to the pen for the rest of code.

Now that we have two blocks of CSS which implement the same layout on the same page, we must be sure that they don't conflict with each other. In particular, on a browsers that supports Grid, the float layout must not disturb the Grid layout. Thanks to the inherent CSS Grid

overriding capabilities, it is sufficient to reset the grid container width, to set its `max-width`, and to reset the margins of the cards(`grid-gap` takes care of the grid gutters now):

```
@supports (display: grid) {
  .grid {
    width: auto;
    max-width: grid-width($max-cols);
    .card {
      margin: 0 !important;
    }
  }
}
```

These overrides have to occur only when CSS Grid is supported, else they would break the float layout. This conditional overriding is performed with the `@support` at-rule.

The code in the pen is organized so that these two blocks of code for the layout are independent, that is, one of them can be removed without affecting the other. To achieve this, the CSS rule common to both layout implementations were grouped together:

```
.grid {
  margin: 0;
  padding: 0;
  list-style: none;

  margin: 40px auto;
  padding-left: $vp-gutter;
  padding-right: $vp-gutter;
}
```

So, if some day we want to get rid of the float layout, it is easy to do so.

To end this discussion on the fallback layout, note how its code looks more complicated and not intuitive as the CSS Grid. This highlights the power of a CSS technique specifically developed for layout, as opposed to an older

CSS feature(floats) whose original intent was (ab)used for many years for the lack of more specific methods.

OTHER FEATURES

Regarding `:focus-within`, it is not supported on Microsoft browsers. With the CSS defined above, on these user agents the user misses the visual feedback when a grid item link receives the focus. Here is a way to fix this:

```
.grid .card a:focus {
  outline: 2px solid #d11ee9;
}
.grid .card:focus-within {
  transform: scale(0.95);
}
.grid .card:focus-within a:focus {
  outline: none;
}
```

A not-supporting browser uses only the first rule, which customize the look of the default focus outline. Even a supporting browser uses this rule, but the outline is reset again in the third line.

Another alternative is to use the `:hover` and `:focus` pseudo classes on the link element, not on the card root element. In fact, for more generality, I preferred to scale the entire card, but in this particular case, where the link stretches to cover the full extent of the card, we could just do this:

```
.grid .card a:hover {
  transform: scale(0.95);
}
.grid .card a:focus {
  outline: none;
  transform: scale(0.95);
}
```

This way, there is no need of `focus-within`, and the

card on focus behaves the same way as when it is hovered, even in older browsers.

object-fit has some troubles on Microsoft browsers too, hence, if the images have not the same aspect ratio as the cards, they may appear stretched.

Finally, on IE 9 and below the CSS gradients are not implemented. As a consequence, the transparent layer which improves the contrast between the card title and the underlying image is not visible. On IE 9 this can be fixed by adding a background color specified with the `rgba()` color function:

```
.grid .card h2 {
  background: rgba(0,0,0, 0.2);
  background: radial-gradient(ellipse at center,
transparent 0, rgba(0,0,0, .36) 100%);
}
```

Final Words

We have already pointed out the power of CSS Grid in the previous section. Furthermore, we may also note how the responsive behavior of the grid outlined in the design specs was achieved without media queries. In fact, the only two queries utilized in the previous snippets were both for the cover card, one to assign the card to its grid area, and the other for its content.

So, after some coding, and lots of pizzas and cakes, we have come to the end. But an end is just a beginning of something else, and we could go ahead with further work, such as adding some entrance animations to the cards, performing an accessibility review, and trying to carry out a selection functionality similar to the original Tumblr page.

Thanks to Freepik for the tasty pictures. Follow the card links to view the original images.

Chapter 3: Easy and Responsive Modern CSS Grid Layout

BY AHMED BOUCHEFRA

In a previous article we explored four different techniques for easily building responsive grid layouts. This article was written back in 2014 before CSS Grid was available so in this tutorial, we'll be using a similar HTML structure but with modern CSS Grid layout.

Throughout this tutorial, we'll create a demo with a basic layout using floats and then enhance it with CSS Grid. We'll demonstrate many useful utilities such as centering elements, spanning items, and easily changing the layout on small devices by redefining grid areas and using media queries.

Live Code

You can find the code in this CodePen.

Before, we dive into creating our responsive grid demo, let's first introduce CSS Grid.

CSS Grid is a powerful 2-dimensional system, which was added to most modern browsers in 2017, that's dramatically changing the way we are creating HTML layouts. Grid Layout allows us to create grid structures in CSS and not HTML.

CSS Grid is supported in most modern browsers except

for IE11 which supports an older version of the standard that could give a few issues. You can use caniuse.com to check for support.

A Grid Layout has a parent container with the `display` property set to `grid` or `inline-grid`. The child elements of the container are grid items which are implicitly positioned thanks to a powerful Grid algorithm. You can also apply different classes to control the placement, dimensions, position and other aspects of the items.

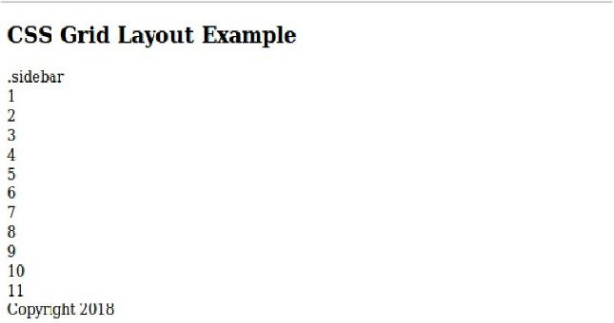
Let's start with a basic HTML page. Create an HTML file and add the following content:

```
<header>
    <h2>CSS Grid Layout Example</h2>
</header>
<aside>
  .sidebar
</aside>

<main>
  <article>
    <span>1</span>
  </article>
  <article>
    <span>2</span>
  </article>
  <!--... -->
  <article>
    <span>11</span>
  </article>
</main>

<footer>
  Copyright 2018
</footer>
```

We use HTML semantics to define the header, sidebar, main and footer sections of our page. In the main section, we add a set of items using the `<article>` tag. `<article>` is an HTML5 semantic tag that could be used for wrapping independent and self-contained content. A single page could have any number of

`<article>` tags.

This is a screen shot of the page at this stage:



Next, let's add basic CSS styling. Add a `<style>` tag in the head of the document and add the following styles:

```
body {
  background: #12458c;
  margin: 0rem;
  padding: 0px;
  font-family: -apple-system, BlinkMacSystemFont,
           "Segoe UI", "Roboto", "Oxygen",
"Ubuntu", "Cantarell",
           "Fira Sans", "Droid Sans", "Helvetica
Neue",
           sans-serif;

}
header {
  text-transform: uppercase;
  padding-top: 1px;
  padding-bottom: 1px;
  color: #fff;
  border-style: solid;
  border-width: 2px;
}
aside {
  color: #fff;
  border-width:2px;
  border-style: solid;
  float: left;
  width: 6.3rem;
}
footer {
  color: #fff;
  border-width:2px;
```

```
    border-style: solid;
    clear: both;
  }
  main {
    float: right;
    width: calc(100% - 7.2rem);
    padding: 5px;
    background: hsl(240, 100%, 50%);
  }
  main > article {
    background: hsl(240, 100%, 50%);
    background-image:
  url('https://source.unsplash.com/daily');
    color: hsl(240, 0%, 100%);
    border-width: 5px;
  }
```

This is a small demonstration page so we'll style tags directly to aid readability rather than applying class naming systems.

We use floats to position the sidebar to the left and the main section to the right and we set the width of the sidebar to a fixed *6.3rem* width then we calculate and set the remaining width for the main section using CSS `calc()` function. The main section contains a gallery of items organized as vertical blocks.

The layout is not perfect. For example, the sidebar does not have the same height as the main content section. There are various CSS techniques to solve the problems but most are hacks or workarounds. Since this layout is a fallback for Grid, it will be seen by a rapidly diminishing

number of users. The fallback is usable and good enough.

The latest versions of Chrome, Firefox, Opera and Safari have support for CSS Grid so that means if your visitors are using these browsers you don't need to worry about providing a fallback. Also you need to account for evergreen browsers. The latest versions of Chrome, Firefox, Edge, and Safari are **evergreen browsers**, i.e. they automatically update themselves silently without prompting the user. To ensure your layout works in every browser, you can start with a default float-based fallback then use Progressive Enhancement techniques to apply a modern Grid layout. Those with older browsers will not receive an identical experience but it will be good enough.

Progressive Enhancement: You Don't Have to Override Everything

When adding the CSS Grid layout on top of your fallback layout, you don't actually need to override all tags or use completely separate CSS styles:

- In a browser that doesn't support CSS Grid, the grid properties you add will be simply ignored.
- If you are using floats for laying out elements, keep in mind that a grid item takes precedence over float i.e if you add a `float: left|right` style to an element which is also a grid element (a child of a parent element that has a `display: grid` style) the float will be ignored in favor of grid.
- Specific feature support can be checked in CSS using @supports rules. This allows us to override fallback styles where necessary while older browsers ignore the @supports block.

Now, let's add CSS Grid to our page. First let's make the `<body>` a grid container and set the grid columns, rows

and areas:

```
body {
  /*...*/
  display: grid;
  grid-gap: 0.1vw;
  grid-template-columns: 6.5rem 1fr;
  grid-template-rows: 6rem 1fr 3rem;
  grid-template-areas: "header   header"
                       "sidebar content"
                       "footer footer";
}
```

We use `display:grid` property to mark `<body>` as a grid container. We set a grid gap of *0.1vw*. Gaps lets you create gutters between Grid cells instead of using margins.

We also use `grid-template-columns` to add two columns, the first column takes a fixed width which is *6.5rem* and the second column takes the remaining width. `fr` is a fractional unit and *1fr* equals 1 part of the available space.

Next, we use `grid-template-rows` to add three rows: The first rows takes a fixed height which equals *6rem*, the third row takes a fixed height of *3rem* and the remaining available space (*1fr*) is assigned to the second row.

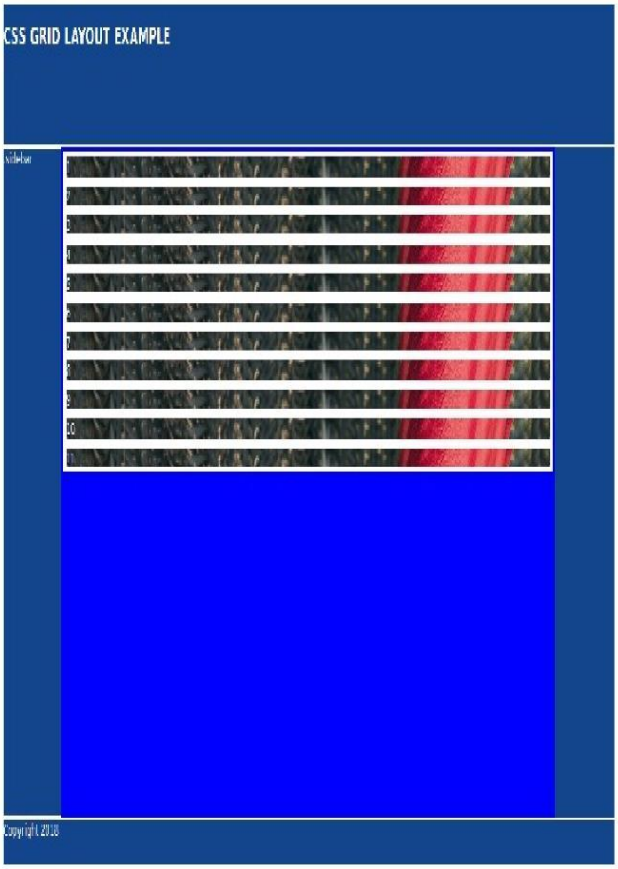
We then use `grid-template-areas` to assign the virtual cells, resulted from the intersection of columns and rows, to areas. Now we need to actually define those areas specified in the areas template using `grid-area`:

```
header {
  grid-area: header;
  /*...*/
}
aside {
  grid-area: sidebar;
  /*...*/
}
```

```
footer {
  grid-area: footer;
  /*...*/
}
main {
  grid-area: content;
  /*...*/
}
```

Most of our fallback code doesn't have any side effects on the CSS Grid except for the width of the main section `width: calc(100% - 7.2rem);` which calculates the remaining width for the main section after subtracting the width of the sidebar plus any margin/padding spaces.

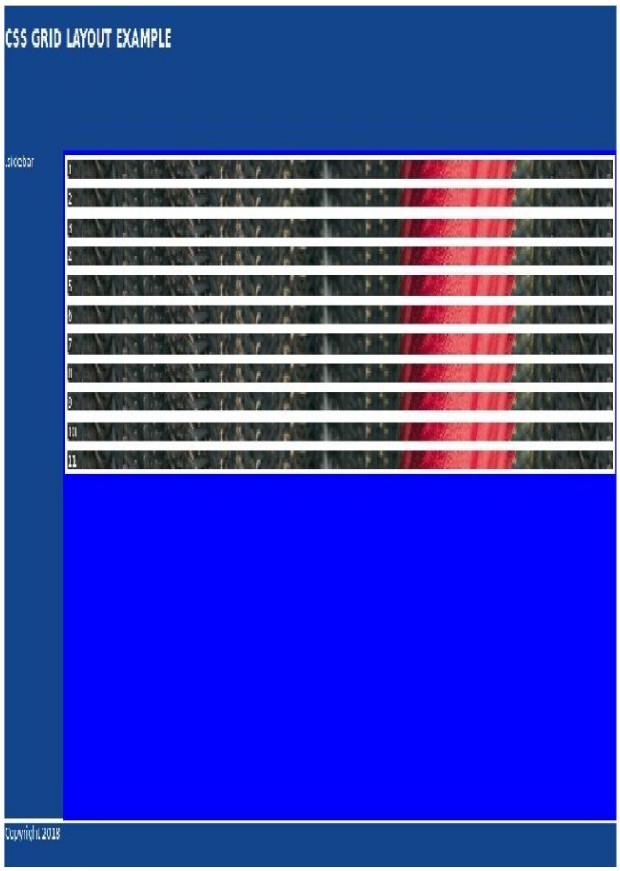
This is a screen shot of the result. Notice how the main area is not taking the full remaining width:



To solve this issue, we can add `width: auto;` when Grid is supported:

```
@supports (display: grid) {
  main {
    width:  auto;
  }
}
```

This is the screen shot of the result now:



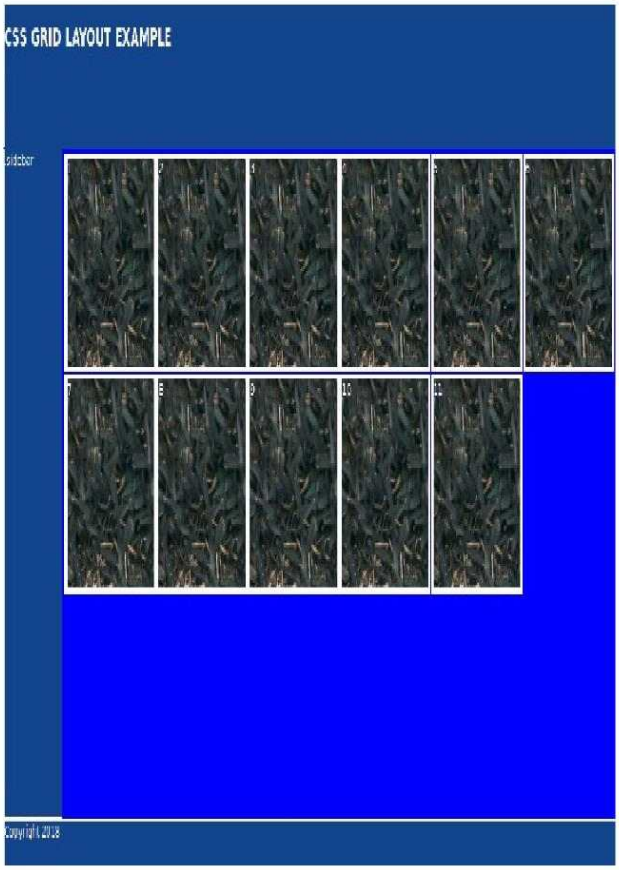
Adding a Nested Grid

A Grid child can be a Grid container itself. Let's make the main section a Grid container:

```
main {
  /*...*/
  display: grid;
  grid-gap: 0.1vw;
  grid-template-columns: repeat(auto-fill,
minmax(12rem, 1fr));
  grid-template-rows: repeat(auto-fill,
minmax(12rem, 1fr));
}
```

We use a grid gap of *0.1vw* and we define columns and rows using the `repeat(auto-fill, minmax(12rem, 1fr));` function. The `auto-fill` option tries to fill the available space with as many columns or rows as possible creating implicit columns or rows if needed. If you want to fit the available columns or rows into the available space you need to use `auto-fit`. Read a good explanation of the differences between `auto-fill` and `auto-fit`.

This is the screen shot of the result:



Using the Grid `grid-column`, `grid-row` and `span` Keywords

CSS Grid provides `grid-column` and `grid-row` which allows you to position grid items inside their parent grid using grid lines. They are short-hand for the properties `grid-row-start` (specifies the start position of the grid item within the grid row), `grid-row-end` (specifies the end position of the grid item within the grid row), `grid-column-start` (specifies the start position of the grid item within the grid column) and `grid-column-end` (specifies the end position of the grid item within the grid column).

You can also use the keyword `span` to specify how much columns or rows to span.

Let's make the second child of the main area span 4 columns and 2 rows and position it from column line 2 and row line 1 (which is also its default location).

```
main article:nth-child(2) {
  grid-column: 2/span 4;
  grid-row: 1/span 2;
}
```

This is a screen shot of the result:

Using Grid Alignment Utilities

We want to center the text inside header, sidebar and footer and the numbers inside the `<article>` elements.

CSS Grid provides six properties `justify-items`, `align-items`, `justify-content`, `align-content`, `justify-self` and `align-self` which can be used to align and justify grid items. They are actually part of CSS box alignment module.

Inside header, aside, article and footer selectors add:

```
display: grid;
align-items: center;
justify-items: center;
```

- `justify-items` is used to justify the grid items along the row axis or horizontally.
- `align-items` aligns the grid items along the column axis or vertically. They can both take the *start*, *end*, *center* and *stretch* values.

This is a screen shot after centering elements:

Restructuring the Grid Layout in Small Devices

Our demo layout is convenient for medium and large screens but might not be the best way to structure the page in small screen devices. Using CSS Grid, we can easily change this layout structure, to make it linear in small devices, by redefining Grid areas and using Media Queries.

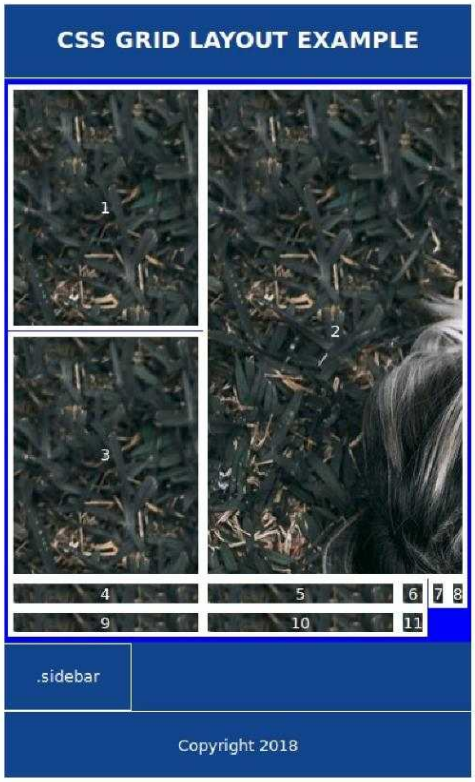
This is a screen shot before adding code to re-structure the layout on small devices:

Now, add the following CSS code:

```
@media all and (max-width: 575px) {
  body {
    grid-template-rows: 6rem  1fr 5.5rem  5.5rem;
    grid-template-columns: 1fr;
```

```
      grid-template-areas:
        "header"
        "content"
        "sidebar"
        "footer";
      }
  }
```

On devices with `<= 575px` we use 4 rows with *6rem, 1fr, 5.5rem, and 5.5rem* widths respectively and one column that takes all the available space. We also redefine Grid areas so the sidebar can take the third row after the main content area on small devices:

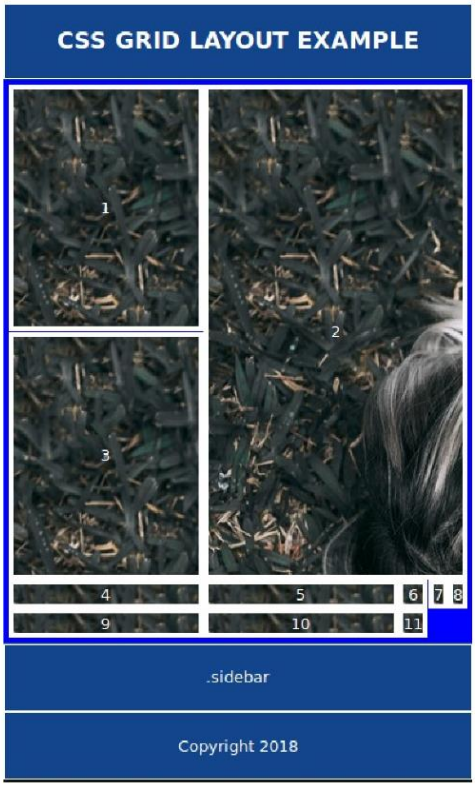


Notice the width of the sidebar, it's not taking the full available width. This is caused by the fallback code so all we need to do is overriding the `width: 6.3rem;` pair

with `width: auto;` on browsers supporting Grid:

```
@supports (display: grid) {
  main, aside {
    width: auto;
  }
}
```

This is a screen shot of the final result:



Live Code

You can find the final code in this CodePen.

Conclusion

Throughout this tutorial, we've created a responsive demo layout with CSS Grid. We've demonstrated using a fallback code for old browsers, adding CSS Grid progressively, restructuring the layout in small devices and centering elements using the alignment properties.

Chapter 4: Progressively Enhanced CSS Layouts from Floats to Flexbox to Grid

BY DIOGO SOUZA

It can be difficult to achieve complex, yet flexible and responsive grid layouts. Various techniques have evolved over the years but most, such as faux columns, were hacks rather than robust design options.

Most of these hacks were built on top of CSS *float* property. When flexbox layout module was introduced to the list of *display* property options, a new world of options became possible. Now you can not only define the direction the container is going to stack the items but also wrap, align (items and lines), order, shrink, etc. them in a container.

With all that power in hands, developers started to create their own combinations of rules for all sorts of layouts. Flexibility reigned like a charm. However, flexbox was designed to approach the one-dimensional layouts: either a row or a column. CSS Grid Layout, in turn, permitted two-dimensional row and column layouts.

Progressive Enhancement vs Graceful Degradation

It's difficult to create a website which supports every user's browser. Two options are commonly used:

Graceful degradation ensures a website continues to function even when something breaks. For example, `float: right` may fail if an element is too big for the screen but it wraps to the next empty space so the block remains usable.

Progressive enhancement takes the opposite approach. The page starts with minimum functionality and features are added when they are supported. The example above could use a CSS media query to verify the screen is a minimum width before allowing an element to float.

When it comes to grid layouts, each browser determines the appearance of their components. In this article, you're going to understand with some real samples how to evolve some web contents from an old strategy to a new one. More specifically, how to progressively enhance the model from a float-based layout to flexbox, and then CSS Grid, respectively.

Your Old Float Layout for A Page

Take a look at the following HTML page:

```
<main>
  <article>
    article content
  </article>

  <aside>
    aside content
  </aside>
</main>
```

It's a small, perhaps the most common, example of grid

disposition you can have in a webpage: two divs sharing the same container (*body*).

The following CSS can be used in all the examples we'll create to set the body style:

```
body {
  font-family: Segoe UI;
  font-style: normal;
  font-weight: 400;
  font-size: 1rem;
}
```

Plus, the CSS snippet for each of our divs, enabling the floating effect:

```
main {
  width: 100%;
}

main, article, aside {
  border: 1px solid #fcddd1;
  padding: 5px;
  float: left;
}

article {
  background-color: #fff4dd;
  width: 74%;
}

aside {
  width: 24%;
}
```

Live Code

You can see the example in action here: Float Layout Example

While it is common, you can float as many elements as you want, one after another, in a way all of them suit the

whole available width. However, this strategy has some downsides:

- It's hard to manage the heights of the container elements;
- Vertical centering, if needed, could be painfully hard to manage;
- Depending on the content you have inside the elements (along with each inner container's CSS properties), the browser could wrap elements to the next free line and break the layout.

One solution is the `display: table` layout:

```
main {
  display: table;
}

main, article, aside {
  display: table-cell;
}
```

However, this does not work in older browsers and more complex layouts become increasingly difficult.

Flexbox Approach

The float box module, known by the name of **flexbox**, is one of the available layout models that work distributing space and powerfully aligning the items of a container (the box) in a one-dimensional way. Its one dimension characteristic, though, does not impede you to design multidimensional layouts (rows and columns), but flexbox may not result in reliable row stacking.

Besides the float approach being very popular and broadly adopted by popular grid frameworks, flexbox presents a series of benefits over float:

- Vertical alignment and height equality for the container's items on each wrapped row;
- The container (box) can increase/decrease based on the available space, and you determine whether it is a column or a row;

- Source independence. That is, the order of the items does not matter, they just need to be inside of the box.

To initiate a flexbox formatting strategy all you need to do is set the CSS *display* property with *flex* value:

```
main {
  width: 100%;
  display: flex;
}

main, article, aside {
  border: 1px solid #fcddd1;
  padding: 5px;
  float: left;
}

article {
  background-color: #fff4dd;
  width: 74%;
}

aside {
  width: 24%;
}
```

The following image shows the result:

Live Code

Here's a live example: See the Pen Flexbox Layout Example

Enhancing to CSS Grid Layout

The CSS **Grid** layout follows up closely the flexbox one, the big difference is that it works directly with the two-dimensional world; that is if you need to design a layout that deals with both rows and columns, the grid layout would probably suit better. It has the same aligning and space distribution factors of flexbox, but now acting

directly to the two dimensions of your container (box). In comparison to the float property, it has even more advantages: easy elements disposition, alignment, row/column/cell control, etc.

Working with CSS Grid is as simple as changing the *display* property of your container element to `grid`. Inside the container, you can also create columns and rows with divs, for example. Let's consider an example of an HTML page with four inner container divs.

Regarding the CSS definitions, let's start with the grid `container` div:

```
div.container {
  display: grid;
  grid-template-columns: 24% 75%;
  grid-template-rows: 200px 300px;
  grid-column-gap: 15px;
  grid-row-gap: 15px;
}
```

The property `grid-template-columns` defines the same configuration you had before: two grid columns occupying 24% and 75% of the whole container width, respectively. The `grid-template-rows`, in turn, do the same applying 200px and 300px as height for the first and second rows, respectively.

Use the properties `grid-column-gap` and `grid-row-gap` to allocate space around the grid elements.

In regards to applying grid alignment properties to specific cells, let's take a look:

```
.div1 {
  background-color: #fff4dd;
  align-self: end;
  justify-self: end;
}

.div4 {
```

```
    align-self: center;
  }
```

For the div of class `div1`, you're aligning and justifying it at the *end* of the grid cell. The properties `align-self` and `justify-self` are valid for all flex items you have in the layout, including, in this case, the grid cells. `div4` was set only the centered alignment, for you to check the difference between both;

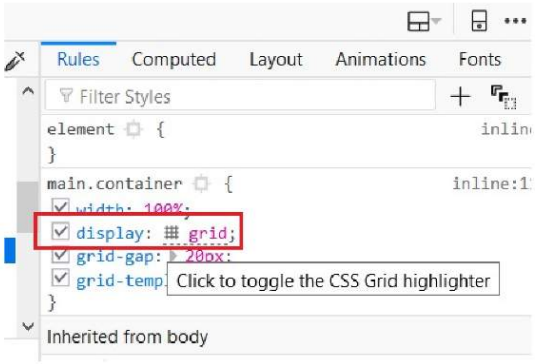
Here's how the elements are positioned now:

You can check out the final grid layout example here:

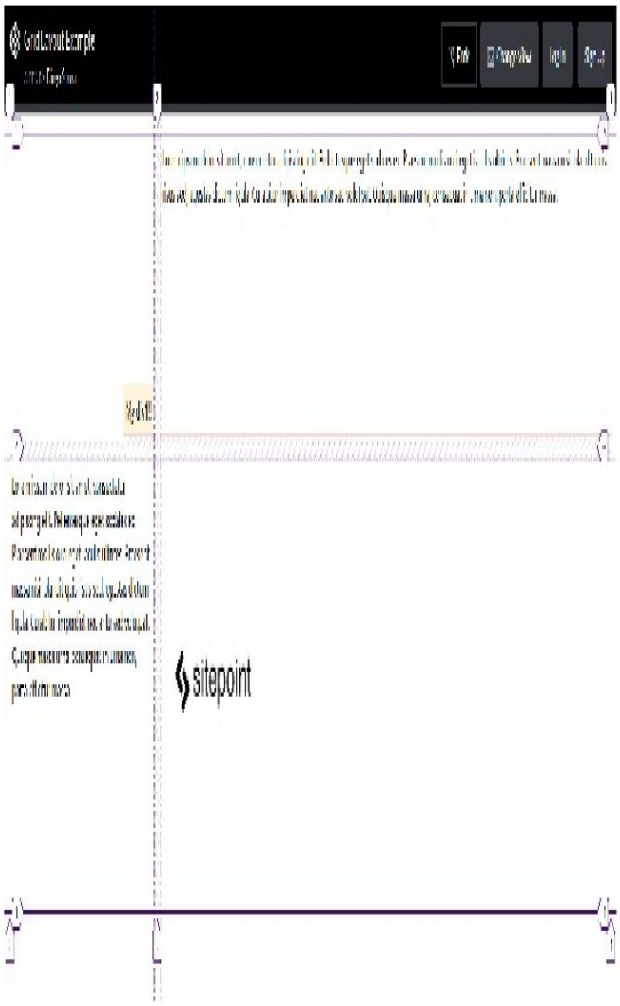
Live Code

You can check out the final grid layout example here: See the Pen Grid Layout Example.

Most browsers also offer native support for grid layout inspections, which is great to see how it handles the grid mechanism internally. Let's try our grid example on Firefox with its Grid Inspector available via Firefox DevTools. In order to open it, right-click the container element and, at the CSS pane's Rules view, find and click the grid icon right after the `display: grid`:



This will toggle the Grid highlighter. You can also control more display settings at the CSS pane's Layout view like the line numbers or the area names exhibition:

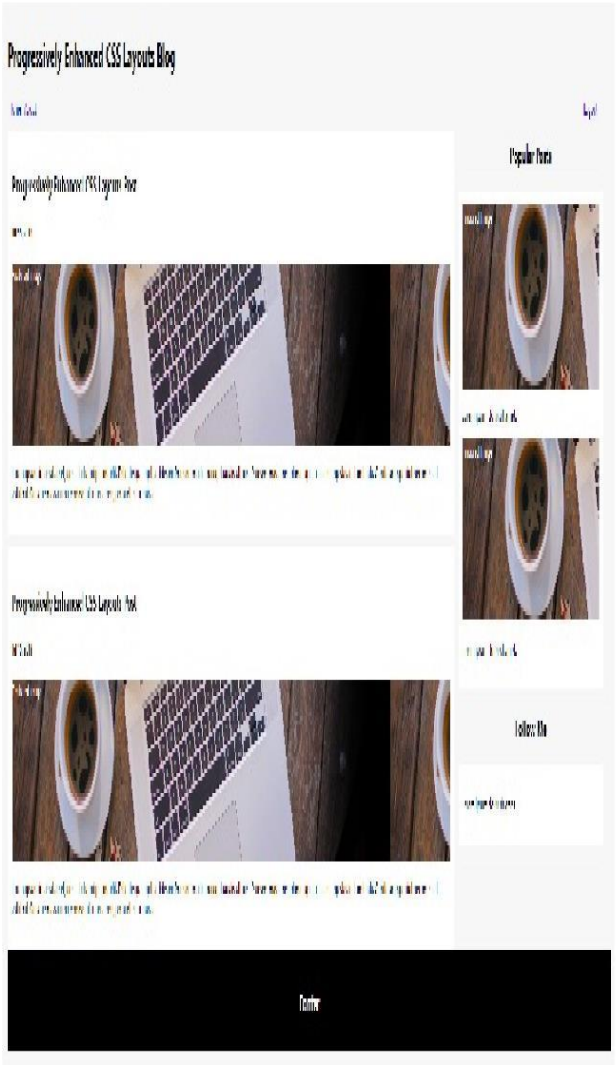


Progressively Enhancing a Blog Layout

In this next example, we're going to use a blog page as a reference to enhance from a totally float-based page to a CSS Grid layout, exploring the way the old layout can be completely transformed to a layout that embraces both

flexbox and grid worlds.

In the following updates, we'll keep an eye to the old browsers that don't support Flexbox or CSS Grid, as well as how the blog behaves on mobile versions. This is the look and feel of our blog page totally based on divs and CSS float property:



Live Code

You can check out this example: See the Pen Blog Layout with Float.

Note that the HTML structure is common for who's

already familiar with semantic tags: a div (the container) containing all the inner elements that'll compose the final layout based on who's floating who.

It's basically made of a simple `header` with a menu that defines how its items (links) will be displayed and aligned:

```
nav.menu {
  width: 98%;
}

ul {
  list-style: none;
  padding: 0px;
  margin: 0px;
}

.left {
  float: left;
}

.right {
  float: right;
}

a {
  padding: 4px 6px;
}
```

The content of the page is divided into two ones, the `section` and `aside` elements:

```
main {
  float: left;
  width: 74%;
}

aside {
  float: left;
  width: 24%;
  margin-left: 15px;
  margin-bottom: 15px;
}

aside h2 {
  text-align: center;
}
```

Nothing exceptional, just left floating the divs and

determining the maximum width of each one on top of the full width. Note also the need for clearing (`clear`) things whenever we need to disable the floating effect after a div:

```
header:after {
  content: "";
  display: table;
  clear: both;
}
```

The end of the CSS brings a `@media` rule that'll break the sidebar to the next column when the page is opened in small screen devices. The end of the HTML brings a simple footer with just a text.

Note: the beginning of the HTML brings also the import of the HTML5 Shiv script in order to enable the use of HTML5 sectioning elements in legacy Internet Explorer.

MENU ENHANCEMENTS WITH FLEXBOX

As seen before, flexbox works better with one-dimensional layouts like a horizontal menu, for example. The same menu structure could be enhanced to this one:

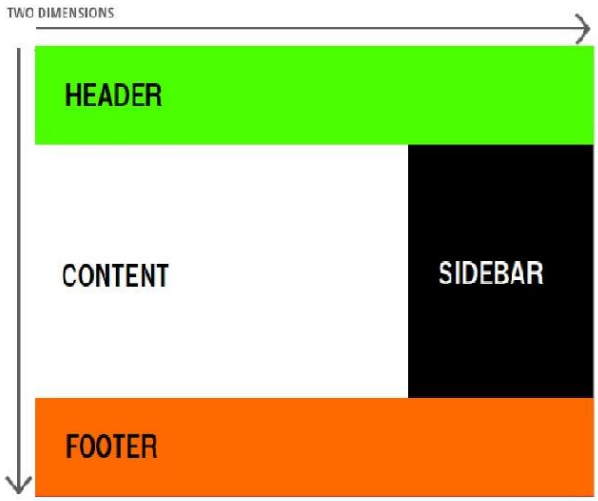
```
nav.menu {
  width: 98%;
}

nav.menu ul {
  display: flex;
}

nav.menu ul > li:last-child {
  margin-left: auto;
}
```

The use of flexbox property automatically disables the float for newer browsers. However, since we preserved the old CSS configurations for the `float` property, the float will still apply when a browser does not support flexbox.

At the same time, the `display: flex` was added to layout the menu using flexbox along with the `margin-left: auto` to the last menu item. This will ensure that this item will be pushed to the right of the layout, separating them into distinct groups.

ENHANCING TO GRID AREAS

CSS Grid gives us more flexibility when it comes to the disposition of inner elements. Once we defined that flexbox was a perfect choice for a horizontal container (the menu), a grid would be perfect for such a two-dimensional blog layout, that basically divides into three parts:



The Grid Area feature consists of the combination of two properties: `grid-template-areas` and `grid-area`. The first one defines how the whole layout height and width would be divided into groups by explicitly putting the names of each group as they'd be in the final layout:

```
div.container {
  width: 100%;
  display: grid;
  grid-gap: 20px;
  grid-template-areas:
    "header header header"
    "content content sidebar"
    "footer footer footer"
}
```

Here, we're dividing the whole container space into 3 rows and 3 columns. The names of the areas repeated alongside the definition say how much space each of them will occupy vertically and horizontally.

Then, we can use the second property by setting for each of the grid areas the respective `grid-area` value:

```
header {
  grid-area: header;
}

main {
  grid-area: content;
  /* other properties */
}

aside {
  grid-area: sidebar;
  /* other properties */
}

footer {
  grid-area: footer;
  /* other properties */
}
```

Just like with flexbox, grid items automatically disable float declarations in browsers which support CSS Grid. The rest of the properties remain the same (the float will still apply when a browser does not support grid.).

Live Code

This example has a Live Codepen.io Demo you can play with: See the Pen Blog Layout with CSS Grid Areas.

ENHANCING GRID TEMPLATES

The property `grid-template` is also very useful when we need to define a template that'll follow a pattern for our grid's definitions.

Let's take a look at the following `.container` CSS:

```
div.container {
  width: 100%;
  display: grid;
  grid-gap: 20px;
  grid-template-columns: auto auto auto;
}
```

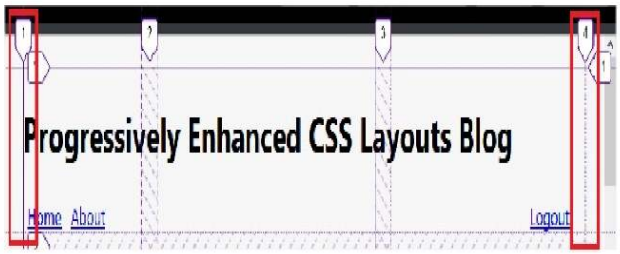
Here, we're basically saying that the size of the columns is determined by the size of the container and on the size

of the content of the items in the column.

Then, each of our grid areas will have to adapt:

```
header {
  grid-column: 1/4;
}

section {
  grid-column: 1/3;
  /* other properties */
}

aside {
  grid-column: 3/4;
  /* other properties */
}

footer {
  grid-column: 1/4;
  /* other properties */
}
```

Here, `grid-column: 1/4` instructs the browser to start a column at track 1 and end at track 4. Tracks are the separators between each grid cell so this element would occupy cells one to three:



Live Code

This example has a live Codepen.io Demo you can play with: See the Pen Blog Layout with CSS Grid Templates.

You can go now and test it in different browser versions as well as your mobile phone to attest to how the pages were progressively enhanced.

More About CSS Grid Layout

For more information, refer to the SitePoint's CSS Grid Layout Introduction and CSS Flexbox Introduction guides.

Mozilla has also provided a great article about CSS Grid Layout and Progressive Enhancement that will certainly add a lot. Good studies!

Chapter 5: Make Forms Great with CSS Grid

BY CRAIG BUCKLER

Form design is a fundamental yet frustrating part of web design and development. Ask anyone who's ever tried to style a `<select>` box or align a label consistently in all browsers.

In 2016, I wrote Make Forms Fun with Flexbox which identified how several form difficulties could be solved with Flexbox. A key benefit was HTML source order consistency with the `<label>` always following its associated field tag in a container:

```
<div>
  <input id="name" name="name" type="text" />
  <label for="name">name</label>
</div>

<div>
  <select id="experience" name="experience"><!--
options --></select>
  <label for="experience">experience</label>
</div>

<div>
  <input id="html" name="html" type="checkbox" />
  <label for="html">HTML</label>
</div>
```

Flexbox could then be used to:

- reposition the label if necessary, i.e. move it to the left of the field on text inputs, select boxes, and textareas.
- vertically align the label and field.

It also became possible to style labels based on the state of their field using adjacent sibling selectors, e.g. apply bold to a label when its associated checkbox is checked:

```
input:checked + label {
  font-weight: bold;
}
```

Flawed Flexboxed Forms

Unfortunately, there are a number of problems using Flexbox to layout a form. Flexbox creates a one-dimensional layout where each item follows another and wraps to a new line when necessary. Field/label pairs must be placed in a container elements with `display: flex;` applied to guarantee each appears on a new row.

It was also necessary to define a fixed label width, such as `10em`. If a long label required more room, its text would either overflow or resize the element and push the field out of alignment with others.

Finally, forms are normally laid out in a grid. Shouldn't we be using CSS Grid now it's fully supported in all mainstream browsers? *Absolutely!*

Development Approach

Most CSS Grid articles demonstrate the concepts and may provide graceful degradation fallbacks for older browsers. That approach is ideal when the layout is mostly decorative, e.g. positioning page content, headers, footers and menus. It rarely matters when *oldBrowserX* shows linear blocks in an unusual order because the page content remains usable.

Form layout is more critical: a misaligned label could lead the user to enter information in the wrong box. For this reason, this tutorial takes a progressive enhancement approach:

1. An initial floated layout will work in all browsers including IE8+ (which did not support Flexbox either). It will not be perfect, but floats never were!
2. Enhance the layout using CSS Grid in all modern browsers.

The examples below contain very few CSS classes and styling is applied directly to HTML elements. That is not the BEM way, but it is intentional to keep the code clean and understandable without distractions.

You could consider using similar code as the base for all forms on your site.

The HTML

A typical HTML form can be kept clean since there is no need for containing (`<div>`) elements around field/label pairs:

```
<form action="get">
  <fieldset>
    <legend>Your web development skillset</legend>

    <div class="formgrid">

      <input id="name" name="name" type="text" />
      <label for="name">name</label>

      <select id="experience" name="experience">
        <option value="1">1 year or less</option>
        <option value="2">2 years</option>
        <option value="3">3 - 4 years</option>
        <option value="5">5 years or more</option>
      </select>
      <label for="experience">experience</label>

      <input id="html" name="html" type="checkbox"
/>
      <label for="html">HTML</label>
```

```
      <input id="css" name="css" type="checkbox"
/>
      <label for="css">CSS</label>

      <input id="javascript" name="javascript"
type="checkbox" />
      <label for="javascript">JavaScript</label>

      <textarea id="skills" name="skills" rows="5"
cols="20"></textarea>
      <label for="skills">other skills</label>

      <button type="submit">SUBMIT</button>

    </div>

  </fieldset>
</form>
```

The only additional element is `<div class="formgrid">`. Browsers cannot apply `display: grid` or `display: flex` to `fieldset` elements. That may eventually be fixed but an outer container is currently required.

Form Float Fallback

After some initial font and color styling, the float layout will allocate:

- 70% of the space to fields which are floated right, and
- 30% of the space to labels which are floated left

```
/* fallback 30%/70% float layout */
input, output, textarea, select, button {
  clear: both;
  float: right;
  width: 70%;
}

label {
  float: left;
  width: 30%;
  text-align: right;
  padding: 0.25em 1em 0 0;
}
```

Checkbox and radio buttons are positioned before the label and floated left. Their intrinsic width can be used (`width:auto`) but a left margin of 30% is required to align correctly:

```
button, input[type="checkbox"],
input[type="radio"] {
  width: auto;
  float: left;
  margin: 0.5em 0.5em 0 30%;
}

input[type="checkbox"] + label,
input[type="radio"] + label {
  width: auto;
  text-align: left;
}
```

Live Code

The layout works in all browsers including IE8+: See the Pen form grid 1: float layout.

A less conscientious developer would go home for the day but this layout has several problems:

- the padding and margin tweaks are fragile and can look inconsistent across browsers
- if longer labels or different-sized fonts are ever required, the CSS spacing will require adjustment, and
- the design breaks at smaller screen sizes and labels can overflow fields.

Get Going with Grid

The Grid module adds 18 new CSS properties in order to create a layout with rows and columns. Elements within the grid can placed in any row/column, span multiple rows and/or columns, overlap other elements, and be aligned horizontally and/or vertically. There are similarities to Flexbox, but:

- Flexbox is one-dimensional. Elements come one after the other and may or may not wrap to a new "row". Menus and photo galleries are a typical use-case.
- Grid is two-dimensional and respects both rows and columns. If an element is too big for its cell, the row and/or column will grow accordingly. Grid is ideal for page and form layouts.

It is possibly better to compare CSS Grid with table-based layouts but they're considerably more flexible and require less markup. It has a steeper learning curve than other CSS concepts but you're unlikely to require all the properties and the minimum is demonstrated here. The most basic grid is defined on a containing element:

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

More practically, layouts also require the number of columns, their sizes, and the gap between rows and columns, e.g.

```
.container {
  display: grid;
  grid-template-columns: 10% 1fr 2fr 12em;
  grid-gap: 0.3em 0.6em;
}
```

This defines four columns. Any measurement unit can be used as well as the `fr` fractional unit. This calculates the remaining space in a grid and distributes accordingly. The example above defines total of `3fr` on columns two and three. If 600 pixels of horizontal space was available:

- `1fr` equates to `(1fr / 3fr) * 600px` = `200px`
- `2fr` equates to `(2fr / 3fr) * 600px` = `400px`

A gap of `0.3em` is defined between rows and `0.6em` between columns.

All child elements of the `.container` are now grid items. By default, the first child element will appear at row 1, column 1. The second in row 1, column 2, and the sixth in row 2, column 2. It's possible to size rows using a property such as `grid-template-rows` but heights will be inferred by the content.

Grid support is excellent. It's not available in Opera Mini but even IE11 offers an older implementation of the specification. In most cases, fallbacks are simple:

- Older browsers can use flexbox, floats, inline-blocks, or `display:table` layouts. All Grid properties are ignored.
- When a browser supports grid, all flexbox, floats, inline-blocks and table layout properties assigned to a grid item are disabled.

Grid tools and resources:

- MDN Grid Layout
- A Complete Guide to Grid
- Grid by Example
- Grid "fallbacks" and overrides
- CSS Grid Playground
- CSS Grid Garden
- Layoutit!

Firefox and Chrome-based browsers have excellent DevTool Grid layout and visualisation tools.

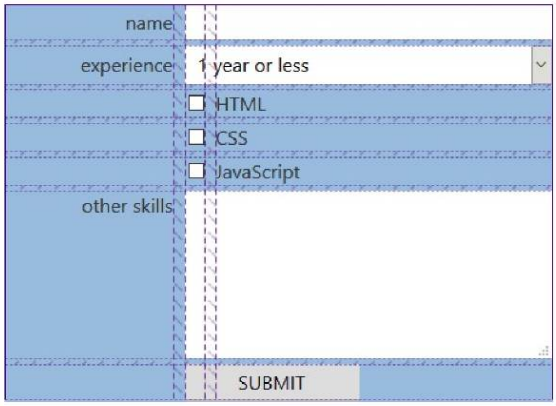
Form Grid

To progressively enhance the existing form, Grid code will be placed inside an `@supports` declaration:

```
/* grid layout */
@supports (display: grid) {
  ...
}
```

This is rarely necessary in most grid layouts. However, this example resets all float paddings and margins; that must only occur when a CSS Grid is being applied.

The layout itself will use a three column design:



where:

- standard labels appear in column one
- checkbox and radio buttons span columns one and two (but are aligned right)
- checkbox and radio labels appear in column three
- all other fields span columns two and three.

The outer container and child field properties:

```
.formgrid {
  display: grid;
  grid-template-columns: 1fr 1em 2fr;
  grid-gap: 0.3em 0.6em;
  grid-auto-flow: dense;
  align-items: center;
}

input, output, textarea, select, button {
  grid-column: 2 / 4;
  width: auto;
  margin: 0;
}
```

`grid-column` defines the starting and ending grid tracks. Tracks are the *edges* between cells so the three-column layout has four tracks:

1. the first track on the left-hand side of the grid before column one
2. the track between columns one and two
3. the track between columns two and three
4. the final track on the right-hand edge of the grid after column three

`grid-column: 2 / 4;` positions all fields between tracks 2 and 4 - *or inside columns two and three*.

The first HTML element is the name `<input>`. It spans columns two and three which means column one (track 1 / 2) is empty on that row. By default, the `name` label would therefore drop to row 2, column 1. However, by setting `grid-auto-flow: dense;` in the container, the browser will attempt to fill empty cells earlier in the grid before progressing to a new row.

Checkboxes and radio buttons can now be set to span tracks 1 to 3 (columns one and two) but align themselves to the right-hand edge using `justify-self: end`:

```
input[type="checkbox"], input[type="radio"] {
  grid-column: 1 / 3;
  justify-self: end;
  margin: 0;
}
```

Labels on the grid will handle themselves and fit into whichever row cell is empty. However, the default widths and spacing from the float layout are now unnecessary:

```
label, input[type="checkbox"] + label,
input[type="radio"] + label {
  width: auto;
  padding: 0;
  margin: 0;
}
```

Finally, `<textarea>` labels can be vertically positioned at the top of the cell rather than centered:

```
textarea + label {
  align-self: start;
}
```

Live Code

The final grid layout: See the Pen form grid 2: grid applied

Unlike floats, the design will not break at small dimensions or require tweaking when different fonts, sizes or labels are added.

Grid Enlightenment

It's taken several years to become viable, but CSS Grid is well supported and offers layout possibilities which would have been difficult with floats or flexbox. Forms are an ideal use-case and the resulting CSS is short yet robust.

If you're looking to learn another CSS technique, Grid should be at the top of your list.