Legacy layout methods

Previous | Overview: CSS layout | Next

Grid systems are a very common feature used in CSS layouts, and before CSS Grid Layout they tended to be implemented using floats or other layout features. You imagine your layout as a set number of columns (e.g. 4, 6, or 12), and then fit your content columns inside these imaginary columns. In this article we'll explore how these older methods work, in order that you understand how they were used if you work on an older project.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of how CSS works (study Introduction to CSS and Styling boxes.)
Objective:	To understand the fundamental concepts behind the grid layout systems used prior to CSS Grid Layout being available in browsers.

Layout and grid systems before CSS Grid Layout

It may seem surprising to anyone coming from a design background that CSS didn't have an inbuilt grid system until very recently, and instead we seemed to be using a variety of sub-optimal methods to create grid-like designs. We now refer to these as "legacy" methods.

For new projects, in most cases CSS Grid Layout will be used in combination with one or more other modern layout methods to form the basis for any layout. You will however encounter "grid systems" using these legacy methods from time to time. It is worth understanding how they work, and why they are different to CSS Grid Layout.

This lesson will explain how grid systems and grid frameworks based on floats and flexbox work. Having studied Grid Layout you will probably be surprised how complicated this all seems! This knowledge will be helpful to you if you need to create fallback code for browsers that do not support newer methods, in addition to allowing you to work on existing projects which use these types of systems.

It is worth bearing in mind, as we explore these systems, that none of them actually create a grid in the way that CSS Grid Layout creates a grid. They work by giving items a size, and pushing them around to line them up in a way that *looks* like a grid.

A two column layout

Let's start with the simplest possible example — a two column layout. You can follow along by creating a new `index.html` file on your computer, filling it with a simple HTML template, and inserting the below code into it at the appropriate places. At the bottom of the section you can see a live example of what the final code should look like.

First of all, we need some content to put into our columns. Replace whatever is inside the body currently with the following:

```
<h1>2 column layout example</h1>
<div>
  <h2>First column</h2>
  <p> Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus a
</div>

<div>
```

```
  <h2>Second column</h2>
  <p>Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuad
</div>
```

Each one of the columns needs an outer element to contain its content and let us manipulate all of it at once. In this example case we've chosen `<div>`s, but you could choose something more semantically appropriate like `<article>`s, `<section>`s, and `<aside>`, or whatever.

Now for the CSS. First, of all, apply the following to your HTML to provide some basic setup:

```
body {
  width: 90%;
  max-width: 900px;
  margin: 0 auto;
}
```

The body will be 90% of the viewport wide until it gets to 900px wide, in which case it will stay fixed at this width and center itself in the viewport. By default, its children (the `<h1>` and the two `<div>`s) will span 100% of the width of the body. If we want the two `<div>`s to be floated alongside one another, we need to set their widths to total 100% of the width of their parent element or smaller so they can fit alongside one another. Add the following to the bottom of your CSS:

```
div:nth-of-type(1) {
  width: 48%;
}

div:nth-of-type(2) {
  width: 48%;
}
```

Here we've set both to be 48% of their parent's width — this totals 96%, leaving us 4% free to act as a gutter between the two columns, giving the content some space to breathe. Now we just need to float the columns, like so:

```
div:nth-of-type(1) {
  width: 48%;
  float: left;
}

div:nth-of-type(2) {
  width: 48%;
  float: right;
}
```

Putting this all together should give us a result like so:

2 column layout example

First column

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla luctus aliquam dolor, eu lacinia lorem placerat vulputate. Duis felis orci, pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta. Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula. Curabitur vehicula tellus neque, ac ornare ex malesuada et. In vitae convallis lacus. Aliquam erat volutpat. Suspendisse ac imperdiet turpis.

Second column

Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuada ultrices. Phasellus turpis est, posuere sit amet dapibus ut, facilisis sed est. Nam id risus quis ante semper consectetur eget aliquam lorem. Vivamus tristique elit dolor, sed pretium metus suscipit vel. Mauris ultricies lectus sed lobortis finibus. Vivamus eu urna eget velit cursus viverra quis vestibulum sem. Aliquam tincidunt eget purus in interdum. Cum sociis natoque penatibus et magnis dis parturient montes.

You'll notice here that we are using percentages for all the widths — this is quite a good strategy, as it creates a **liquid layout**, one that adjusts to different screen sizes and keeps the

same proportions for the column widths at smaller screen sizes. Try adjusting the width of your browser window to see for yourself. This is a valuable tool for responsive web design.

> **Note**: You can see this example running at 0_two-column-layout.html (see also the source code).

Creating simple legacy grid frameworks

The majority of legacy frameworks use the behavior of the `float` property to float one column up next to another in order to create something that looks like a grid. Working through the process of creating a grid with floats shows you how this works and also introduces some more advanced concepts to build on the things you learned in the lesson on floats and clearing.

The easiest type of grid framework to create is a fixed width one — we just need to work out how much total width we want our design to be, how many columns we want, and how wide the gutters and columns should be. If we instead decided to lay out our design on a grid with columns that grow and shrink according to browser width, we would need to calculate percentage widths for the columns and gutters between them.

In the next sections we will look at how to create both. We will create a 12 column grid — a very common choice that is seen to be very adaptable to different situations given that 12 is nicely divisible by 6, 4, 3, and 2.

A simple fixed width grid

Lets first create a grid system that uses fixed width columns.

Start out by making a local copy of our sample simple-grid.html file, which contains the following markup in its body.

```
<div class="wrapper">
  <div class="row">
    <div class="col">1</div>
    <div class="col">2</div>
    <div class="col">3</div>
```

```
      <div class="col">4</div>
      <div class="col">5</div>
      <div class="col">6</div>
      <div class="col">7</div>
      <div class="col">8</div>
      <div class="col">9</div>
      <div class="col">10</div>
      <div class="col">11</div>
      <div class="col">12</div>
    </div>
    <div class="row">
      <div class="col span1">13</div>
      <div class="col span6">14</div>
      <div class="col span3">15</div>
      <div class="col span2">16</div>
    </div>
  </div>
```

The aim is to turn this into a demonstration grid of two rows on a twelve column grid — the top row demonstrating the size of the individual columns, the second row some different sized areas on the grid.

In the `<style>` element, add the following code, which gives the wrapper container a width of 980 pixels, with padding on the right hand side of 20 pixels. This leaves us with 960 pixels for our total column/gutter widths — in this case, the padding is subtracted from the total content width because we have set `box-sizing` to `border-box` on all elements on the site (see Changing the box model completely for more explanation).

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

body {
  width: 980px;
  margin: 0 auto;
}
```

```
9
10  .wrapper {
11    padding-right: 20px;
12  }
```

Now use the row container that is wrapped around each row of the grid to clear one row from another. Add the following rule below your previous one:

```
1  .row {
2    clear: both;
3  }
```

Applying this clearing means that we don't need to completely fill each row with elements making the full twelve columns. The rows will remain separated, and not interfere with each other.

The gutters between the columns are 20 pixels wide. We create these gutters as a margin on the left side of each column — including the first column, to balance out the 20 pixels of padding on the right hand side of the container. So we have 12 gutters in total — 12 x 20 = 240.

We need to subtract that from our total width of 960 pixels, giving us 720 pixels for our columns. If we now divide that by 12, we know that each column should be 60 pixels wide.

Our next step is to create a rule for the class `.col`, floating it left, giving it a `margin-left` of 20 pixels to form the gutter, and a `width` of 60 pixels. Add the following rule to the bottom of your CSS:

```
1  .col {
2    float: left;
3    margin-left: 20px;
4    width: 60px;
5    background: rgb(255, 150, 150);
6  }
```

The top row of single columns will now lay out neatly as a grid.

> **Note**: We've also given each column a light red color so you can see exactly how much space each one takes up.

Layout containers that we want to span more than one column need to be given special classes to adjust their `width` values to the required number of columns (plus gutters in between). We need to create an additional class to allow containers to span 2 to 12 columns. Each width is the result of adding up the column width of that number of columns plus the gutter widths, which will always number one less than the number of columns.

Add the following at the bottom of your CSS:

```
/* Two column widths (120px) plus one gutter width (20px) */
.col.span2 { width: 140px; }
/* Three column widths (180px) plus two gutter widths (40px) */
.col.span3 { width: 220px; }
/* And so on... */
.col.span4 { width: 300px; }
.col.span5 { width: 380px; }
.col.span6 { width: 460px; }
.col.span7 { width: 540px; }
.col.span8 { width: 620px; }
.col.span9 { width: 700px; }
.col.span10 { width: 780px; }
.col.span11 { width: 860px; }
.col.span12 { width: 940px; }
```

With these classes created we can now lay out different width columns on the grid. Try saving and loading the page in your browser to see the effects.

> **Note**: If you are having trouble getting the above example to work, try comparing it against our finished version on GitHub (see it running live also).

Try modifying the classes on your elements or even adding and removing some containers, to see how you can vary the layout. For example, you could make the second row look like this:

```
<div class="row">
  <div class="col span8">13</div>
```

```
  <div class="col span4">14</div>
</div>
```

Now you've got a grid system working, you can simply define the rows and the number of columns in each row, then fill each container with your required content. Great!

Creating a fluid grid

Our grid works nicely, but it has a fixed width. We really want a flexible (fluid) grid that will grow and shrink with the available space in the browser viewport. To achieve this we can turn the reference pixel widths into percentages.

The equation that turns a fixed width into a flexible percentage-based one is as follows.

```
target / context = result
```

For our column width, our **target width** is 60 pixels and our **context** is the 960 pixel wrapper. We can use the following to calculate a percentage.

```
60 / 960 = 0.0625
```

We then move the decimal point 2 places giving us a percentage of 6.25%. So, in our CSS we can replace the 60 pixel column width with 6.25%.

We need to do the same with our gutter width:

```
20 / 960 = 0.02083333333
```

So we need to replace the 20 pixel `margin-left` on our `.col` rule and the 20 pixel `padding-right` on `.wrapper` with 2.08333333%.

Updating our grid

To get started in this section, make a new copy of your previous example page, or make a local copy of our simple-grid-finished.html code to use as a starting point.

Update the second CSS rule (with the `.wrapper` selector) as follows:

```
body {
  width: 90%;
  max-width: 980px;
  margin: 0 auto;
}

.wrapper {
  padding-right: 2.08333333%;
}
```

Not only have we given it a percentage `width`, we have also added a `max-width` property in order to stop the layout becoming too wide.

Next, update the fourth CSS rule (with the `.col` selector) like so:

```
.col {
  float: left;
  margin-left: 2.08333333%;
  width: 6.25%;
  background: rgb(255, 150, 150);
}
```

Now comes the slightly more laborious part — we need to update all our `.col.span` rules to use percentages rather than pixel widths. This takes a bit of time with a calculator; to save you some effort, we've done it for you below.

Update the bottom block of CSS rules with the following:

```
/* Two column widths (12.5%) plus one gutter width (2.08333333%) */
.col.span2 { width: 14.58333333%; }
/* Three column widths (18.75%) plus two gutter widths (4.1666666) */
.col.span3 { width: 22.91666666%; }
/* And so on... */
.col.span4 { width: 31.24999999%; }
.col.span5 { width: 39.58333332%; }
```

```
.col.span6 { width: 47.91666665%; }
.col.span7 { width: 56.24999998%; }
.col.span8 { width: 64.58333331%; }
.col.span9 { width: 72.91666664%; }
.col.span10 { width: 81.24999997%; }
.col.span11 { width: 89.5833333%; }
.col.span12 { width: 97.91666663%; }
```

Now save your code, load it in a browser, and try changing the viewport width — you should see the column widths adjust nicely to suit.

> **Note**: If you are having trouble getting the above example to work, try comparing it against our finished version on GitHub (see it running live also).

Easier calculations using the calc() function

You could use the `calc()` function to do the math right inside your CSS — this allows you to insert simple mathematical equations into your CSS values, to calculate what a value should be. It is especially useful when there is complex math to be done, and you can even compute a calculation that uses different units, for example "I want this element's height to always be 100% of its parent's height, minus 50px". See this example from a MediaRecorder API tutorial.

Anyway, back to our grids! Any column that spans more than one column of our grid has a total width of 6.25% multiplied by the number of columns spanned plus 2.08333333% multiplied by the number of gutters (which will always be the number of columns minus 1). The `calc()` function allows us to do this calculation right inside the width value, so for any item spanning 4 columns we can do this, for example:

```
.col.span4 {
  width: calc((6.25%*4) + (2.08333333%*3));
}
```

Try replacing your bottom block of rules with the following, then reload it in the browser to see if you get the same result:

```
.col.span2 { width: calc((6.25%*2) + 2.08333333%); }
.col.span3 { width: calc((6.25%*3) + (2.08333333%*2)); }
```

```
.col.span4 { width: calc((6.25%*4) + (2.08333333%*3)); }
.col.span5 { width: calc((6.25%*5) + (2.08333333%*4)); }
.col.span6 { width: calc((6.25%*6) + (2.08333333%*5)); }
.col.span7 { width: calc((6.25%*7) + (2.08333333%*6)); }
.col.span8 { width: calc((6.25%*8) + (2.08333333%*7)); }
.col.span9 { width: calc((6.25%*9) + (2.08333333%*8)); }
.col.span10 { width: calc((6.25%*10) + (2.08333333%*9)); }
.col.span11 { width: calc((6.25%*11) + (2.08333333%*10)); }
.col.span12 { width: calc((6.25%*12) + (2.08333333%*11)); }
```

Note: You can see our finished version in fluid-grid-calc.html (also see it live).

Note: If you can't get this to work, it might be because your browser does not support the `calc()` function, although it is fairly well supported across browsers — as far back as IE9.

Semantic versus "unsemantic" grid systems

Adding classes to your markup to define layout means that your content and markup becomes tied to your visual presentation. You will sometimes hear this use of CSS classes described as being "unsemantic" — describing how the content looks — rather than a semantic use of classes that describes the content. This is the case with our `span2`, `span3`, etc., classes.

These are not the only approach. You could instead decide on your grid and then add the sizing information to the rules for existing semantic classes. For example, if you had a `<div>` with a class of `content` on it that you wanted to span 8 columns, you could copy across the width from the `span8` class, giving you a rule like so:

```
.content {
  width: calc((6.25%*8) + (2.08333333%*7));
}
```

Note: If you were to use a preprocessor such as Sass, you could create a simple mixin to insert that value for you.

Enabling offset containers in our grid

The grid we have created works well as long as we want to start all of the containers flush with the left hand side of the grid. If we wanted to leave an empty column space before the first container — or between containers — we would need to create an offset class to add a left margin to our site to push it across the grid visually. More math!

Let's try this out.

Start with your previous code, or use our fluid-grid.html file as a starting point.

Let's create a class in our CSS that will offset a container element by one column width. Add the following to the bottom of your CSS:

```
.offset-by-one {
  margin-left: calc(6.25% + (2.08333333%*2));
}
```

Or if you prefer to calculate the percentages yourself, use this one:

```
.offset-by-one {
  margin-left: 10.41666666%;
}
```

You can now add this class to any container you want to leave a one column wide empty space on the left hand side of it. For example, if you have this in your HTML:

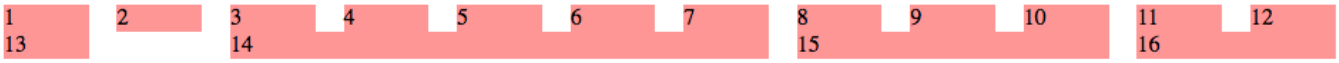
```
<div class="col span6">14</div>
```

Try replacing it with

```
<div class="col span5 offset-by-one">14</div>
```

Note: Notice that you need to reduce the number of columns spanned, to make room for the offset!

Try loading and refreshing to see the difference, or check out our fluid-grid-offset.html example (see it running live also). The finished example should look like this:



> **Note:** As an extra exercise, can you implement an `offset-by-two` class?

Floated grid limitations

When using a system like this you do need to take care that your total widths add up correctly, and that you don't include elements in a row that span more columns than the row can contain. Due to the way floats work, if the number of grid columns becomes too wide for the grid, the elements on the end will drop down to the next line, breaking the grid.

Also bear in mind that if the content of the elements gets wider than the rows they occupy, it will overflow and look a mess.

The biggest limitation of this system is that it is essentially one dimensional. We are dealing with columns, and spanning elements across columns, but not rows. It is very difficult with these older layout methods to control the height of elements without explicitly setting a height, and this is a very inflexible approach too — it only works if you can guarantee that your content will be a certain height.

Flexbox grids?

If you read our previous article about flexbox, you might think that flexbox is the ideal solution for creating a grid system. There are many flexbox-based grid systems available and flexbox can solve many of the issues that we've already discovered when creating our grid above.

However, flexbox was never designed as a grid system and poses a new set of challenges when used as one. As a simple example of this, we can take the same example markup we used above and use the following CSS to style the `wrapper`, `row`, and `col` classes:

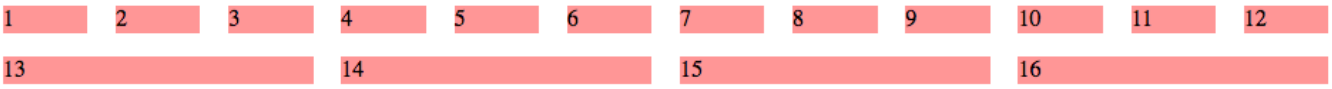
```
body {
  width: 90%;
  max-width: 980px;
  margin: 0 auto;
}

.wrapper {
  padding-right: 2.08333333%;
}


.row {
  display: flex;
}

.col {
  margin-left: 2.08333333%;
  margin-bottom: 1em;
  width: 6.25%;
  flex: 1 1 auto;
  background: rgb(255,150,150);
}
```

You can try making these replacements in your own example, or look at our flexbox-grid.html example code (see it running live also).

Here we are turning each row into a flex container. With a flexbox-based grid we still need rows in order to allow us to have elements that add up to less than 100%. We set that container to `display: flex`.

On `.col` we set the `flex` property's first value (`flex-grow`) to 1 so our items can grow, the second value (`flex-shrink`) to 1 so the items can shrink, and the third value (`flex-basis`) to `auto`. As our element has a `width` set, `auto` will use that width as the `flex-basis` value.

On the top line we get twelve neat boxes on the grid and they grow and shrink equally as we change the viewport width. On the next line, however, we only have four items and these also grow and shrink from that 60px basis. With only four of them they can grow a lot more than the items in the row above, the result being that they all occupy the same width on the second row.



To fix this we still need to include our `span` classes to provide a width that will replace the value used by `flex-basis` for that element.

They also don't respect the grid used by the items above because they don't know anything about it.

Flexbox is **one-dimensional** by design. It deals with a single dimension, that of a row or a column. We can't create a strict grid for columns and rows, meaning that if we are to use flexbox for our grid, we still need to calculate percentages as for the floated layout.

In your project you might still choose to use a flexbox 'grid' due to the additional alignment and space distribution capabilities flexbox provides over floats. You should, however, be aware that you are still using a tool for something other than what it was designed for. So you may feel like it is making you jump through additional hoops to get the end result you want.

Third party grid systems

Now that we understand the math behind our grid calculations, we are in a good place to look at some of the third party grid systems in common use. If you search for "CSS Grid framework" on the Web, you will find a huge list of options to choose from. Popular frameworks such as Bootstrap and Foundation include a grid system. There are also standalone grid systems, either developed using CSS or using preprocessors.

Let's take a look at one of these standalone systems as it demonstrates common techniques for working with a grid framework. The grid we will be using is part of Skeleton, a simple CSS framework.

To get started visit the Skeleton website, and choose "Download" to download the ZIP file. Unzip this and copy the skeleton.css and normalize.css files into a new directory.

Make a copy of our html-skeleton.html file and save it in the same directory as the skeleton and normalize CSS.

Include the skeleton and normalize CSS in the HTML page, by adding the following to its head:

```
<link href="normalize.css" rel="stylesheet">
<link href="skeleton.css" rel="stylesheet">
```

Skeleton includes more than a grid system — it also contains CSS for typography and other page elements that you can use as a starting point. We'll leave these at the defaults for now, however — it's the grid we are really interested in here.

> **Note**: Normalize is a really useful little CSS library written by Nicolas Gallagher, which automatically does some useful basic layout fixes and makes default element styling more consistent across browsers.

We will use similar HTML to our earlier example. Add the following into your HTML body:

```
<div class="container">
  <div class="row">
    <div class="col">1</div>
    <div class="col">2</div>
    <div class="col">3</div>
    <div class="col">4</div>
    <div class="col">5</div>
    <div class="col">6</div>
    <div class="col">7</div>
    <div class="col">8</div>
    <div class="col">9</div>
    <div class="col">10</div>
    <div class="col">11</div>
    <div class="col">12</div>
  </div>
  <div class="row">
    <div class="col">13</div>
    <div class="col">14</div>
    <div class="col">15</div>
    <div class="col">16</div>
  </div>
</div>
```

To start using Skeleton we need to give the wrapper `<div>` a class of `container` — this is already included in our HTML. This centers the content with a maximum width of 960 pixels. You can see how the boxes now never become wider than 960 pixels.

You can take a look in the skeleton.css file to see the CSS that is used when we apply this class. The `<div>` is centered using `auto` left and right margins, and a padding of 20 pixels is applied left and right. Skeleton also sets the `box-sizing` property to `border-box` like we did earlier, so the padding and borders of this element will be included in the total width.

```
.container {
  position: relative;
  width: 100%;
  max-width: 960px;
  margin: 0 auto;
  padding: 0 20px;
  box-sizing: border-box;
}
```

Elements can only be part of the grid if they are inside a row, so as with our earlier example we need an additional `<div>` or other element with a class of `row` nested between the `content` `<div>` and our actual content container `<div>`s. We've done this already as well.

Now let's lay out the container boxes. Skeleton is based on a 12 column grid. The top line boxes all need classes of `one column` to make them span one column.

Add these now, as shown in the following snippet:

```
<div class="container">
  <div class="row">
    <div class="one column">1</div>
    <div class="one column">2</div>
    <div class="one column">3</div>
    /* and so on */
  </div>
</div>
```

Next, give the containers on the second row classes explaining the number of columns they should span, like so:

```
<div class="row">
  <div class="one column">13</div>
  <div class="six columns">14</div>
  <div class="three columns">15</div>
  <div class="two columns">16</div>
</div>
```

Try saving your HTML file and loading it in your browser to see the effect.

> **Note**: If you are having trouble getting this example to work, try comparing it to our html-skeleton-finished.html file (see it running live also).

If you look in the skeleton.css file you can see how this works. For example, Skeleton has the following defined to style elements with “three columns” classes added to them.

```
.three.columns { width: 22%; }
```

All Skeleton (or any other grid framework) is doing is setting up predefined classes that you can use by adding them to your markup. It’s exactly the same as if you did the work of calculating these percentages yourself.

As you can see, we need to write very little CSS when using Skeleton. It deals with all of the floating for us when we add classes to our markup. It is this ability to hand responsibility for layout over to something else that made using a framework for a grid system a compelling choice! However these days, with CSS Grid Layout, many developers are moving away from these frameworks to use the inbuilt native grid that CSS provides.

Summary

You now understand how various grid systems are created, which will be useful in working with older sites and in understanding the difference between the native grid of CSS Grid Layout and these older systems.

Previous | Overview: CSS layout | Next

In this module