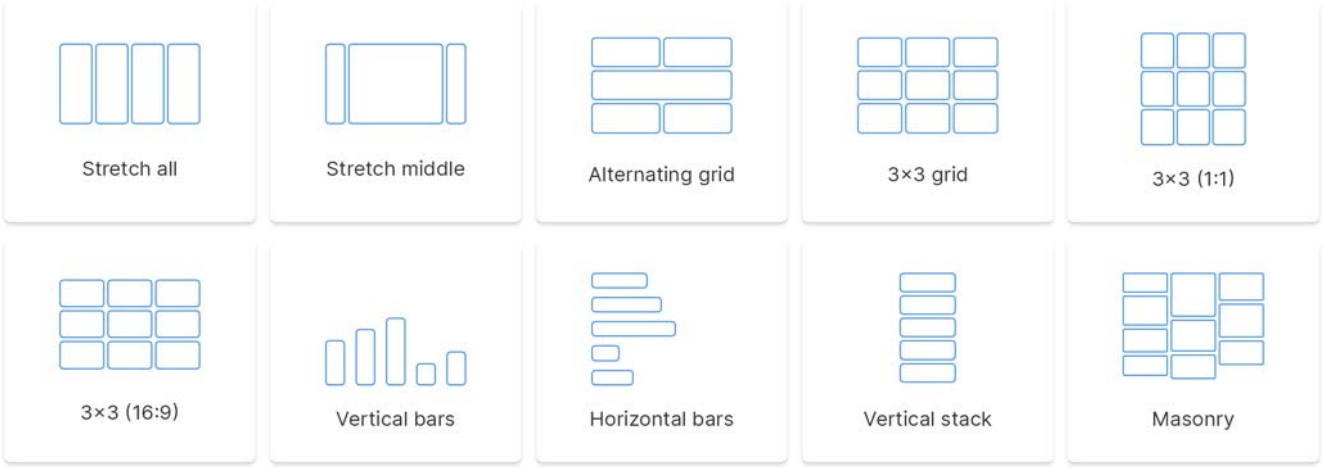
BLOG

Common CSS Flexbox Layout Patterns with Example Code

In theory, it's pretty straightforward to use flexbox (Flexible Box Module) to build complex layouts, but I've often found myself adding `display: flex` to an element and then promptly spending an eternity trying to figure out how to make anything behave like I expect it to. If you're anything like me: here's a list of 10 example flexbox layouts with CSS that you can copy and paste to get started with your own layouts. We'll walk through these basic layout patterns (click to jump to an example):



Every example assumes that your HTML contains an element with a class of `container` which then contains several children that all have a class of `item` (the number of children varies per example):

```
<div class="container">
  <div class="item"></div>
  <div class="item"></div>
  <div class="item"></div>
  ...
</div>
```

Stretch all, fixed spacing

The most basic flexbox pattern: getting a few boxes to stretch and fill the full width of their parent element. All you need to do is to set `display: flex` on the container, and a `flex-grow` value above `0` on the children:

```
.container {
  display: flex;
}

.item {
  flex-grow: 1;
  height: 100px;
}

.item + .item {
  margin-left: 2%;
}
```

We use a `+` selector to only add gaps between pairs of items (essentially just skipping the left margin for the first item in the list).

Increasing `flex-grow` will increase *the amount of space* that an element is

allowed to stretch to compared to any other element. If we set `flex-grow` to `2` on the middle element here, we would basically divide up the available space into 6 chunks (1 chunk for each item plus 1 extra for the middle item, 1+1+2+1+1). The middle item gets two chunks (`flex-grow: 2`) worth of space, and all other elements get one chunk (`flex-grow: 1`).

Stretch middle, fixed spacing

A common app and web pattern is to create a top bar where we want to stretch only the middle element, but keep the right most and left most elements fixed. If we just want one element to stretch, we can set a fixed width (e.g. `100px`) on an element that should stay static, and `flex-grow: 1` on the element that should stretch:

```
.container {
  display: flex;
}

.item {
  height: 100px;
  width: 100px; /* A fixed width as the default */
}

.item-center {
  flex-grow: 1; /* Set the middle element to grow and stretch */
}

.item + .item {
  margin-left: 2%;
}
```

Even if the middle element here has a defined `width` of `100px`, `flex-grow` will make it stretch to take up all the available space.

Alternating grid

A layout pattern that I use on my blog overview is to create a grid with some variation: after every row of two equally sized items there's one item that takes up an entire row:

To accomplish this we need to:

1. Set `flex-wrap: wrap` on the container (or all `items` would be rendered on a single row)

2. Set `justify-content: space-between` on the container, to only create space between the elements (and not between the edge of the parent element and items)

3. Set every item's `width` to `49%` (or something similar that is equal to or less than `50%`)

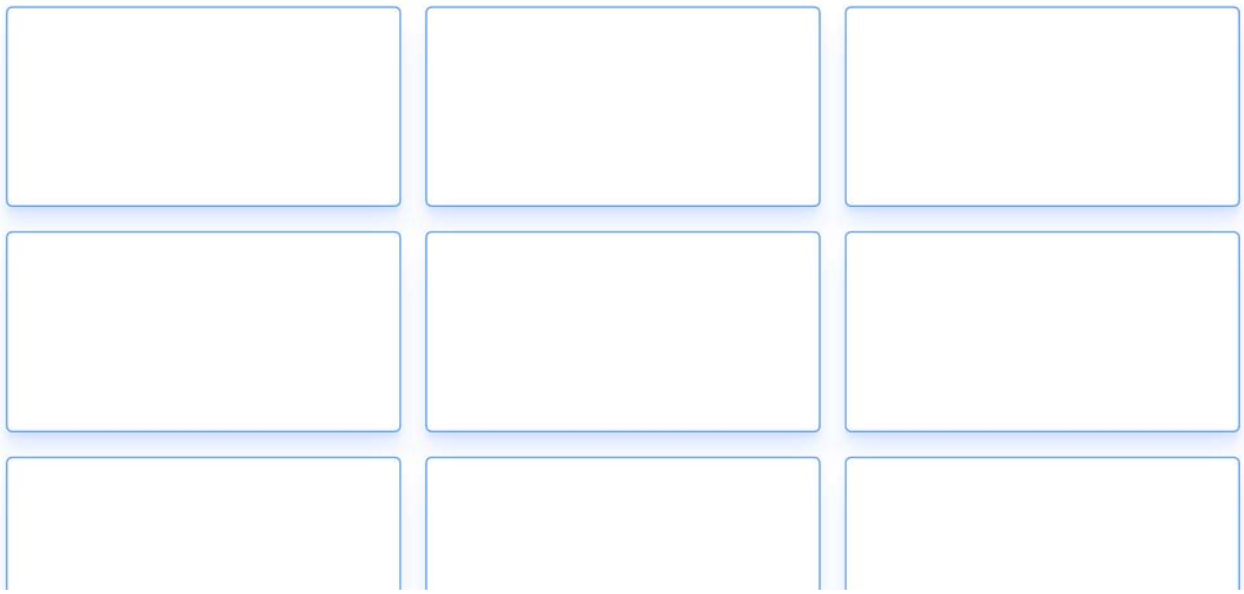
4. Set every third item's `width` to `100%` so that it fills that entire row. We can target every third item in the list with the `nth-child()` selector.

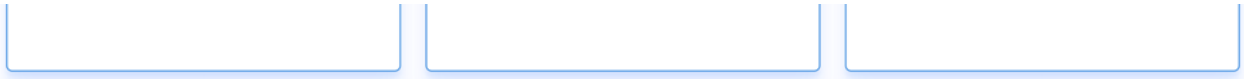
```
.container {
  display: flex;
  flex-wrap: wrap;
```

```
  justify-content: space-between;
}

.item {
  width: 48%;
  height: 100px;
  margin-bottom: 2%;
}

.item:nth-child(3n) {
  width: 100%;
}
```

If you want the first row to be full-width and the two following items to share a row, note that you can't write `.item:nth-child(1n) { width: 100% }`—that would target all items. You have to target the first item by selecting every third element, and then stepping back two items: `.item:nth-child(3n-2) { width: 100% }`.

3×3 grid



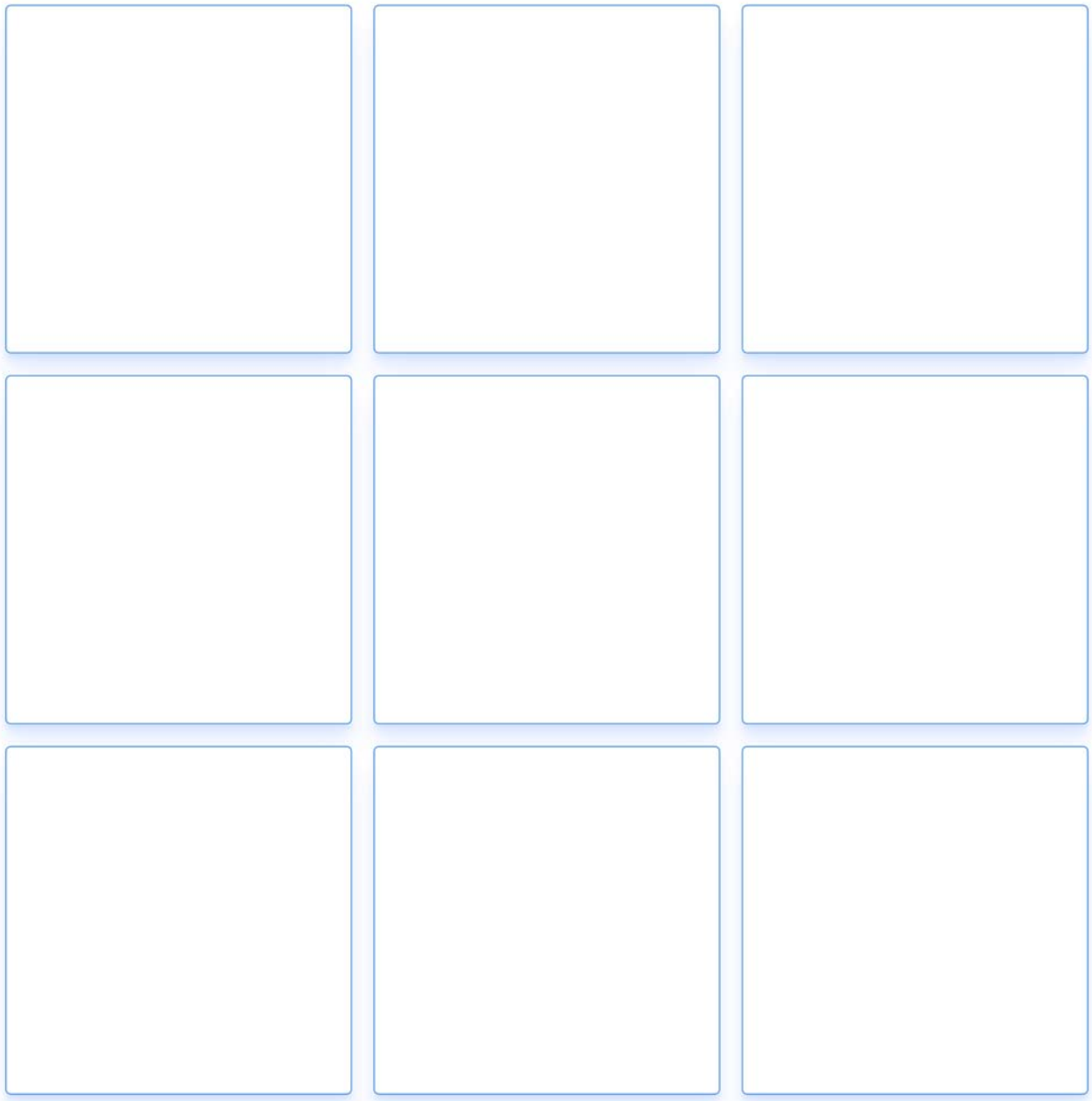


We can create a 3×3 grid by setting `flex-grow` to `0` and `flex-basis` to the preferred width for all children (here done using the `flex` short-hand: `flex: 0 32%`). The margins between the items are the leftovers from every row, i.e. (100%-32×3)/2=2%. I've matched the margin (`margin-bottom: 2%`) to achieve even spacing between all elements:

```
.container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}

.item {
  flex: 0 32%;
  height: 100px;
  margin-bottom: 2%; /* (100-32*3)/2 */
}
```

You can change the `flex-basis` to increase or decrease the number of items on each row. `flex: 0 24%` would put 4 items on every row, `flex: 0 19%` would put 5 items on every row, and so on so forth.

3×3 grid, constrained proportions (1:1)



We can create a grid full of items with constrained proportions by using a somewhat hacky `padding` CSS trick. If we use `%` when setting `padding` on an element the `padding` is set relative to that item's parent's `width`, `.container` in this case. We can use that effect to create a square by setting an item's `width` and `padding-bottom` to the same value (and effectively setting the `height` of that element through `padding-bottom`):

```
.container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}

.item {
  width: 32%;
  padding-bottom: 32%; /* Same as width, sets height */
  margin-bottom: 2%; /* (100-32*3)/2 */
  position: relative;
}
```

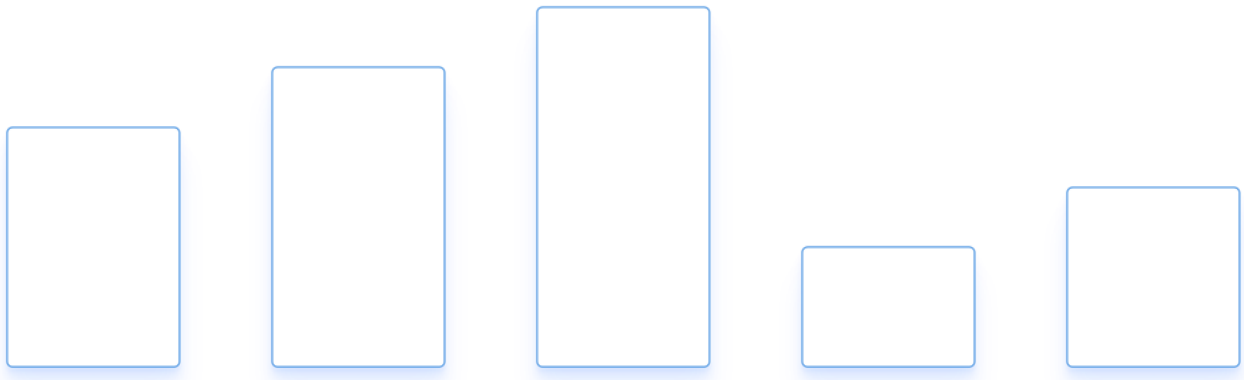
Since we've created an element that is effectively made up of only padding, and there's no place left for content, we need to set `position: relative` on the `.item` in this example and add a child element with `position: absolute`, and use that element to "reset" the canvas and add content.

3×3 grid, constrained proportions (16:9)

To change the proportions of the items all you need to do is change the proportions of the `width` and `padding-bottom` on the `.item`. Changing the `width` would affect the number of items on each row, so as to not affect the grid structure we can for example change `padding-bottom` to `18%` to create 16:9 (equivalent to 32:18) rectangles.

```
.container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}

.item {
  width: 32%;
  padding-bottom: 18%; /* 32:18, i.e. 16:9 */
  margin-bottom: 2%; /* (100-32*3)/2 */
}
```

Graph: vertical bars



If you want to use flexbox to create a simple graph visualization, all you need to do is to set `align-items` of the container to `flex-end` and define a

`height` for each bar. `flex-end` will make sure that the bars are anchored to the bottom of the graph.

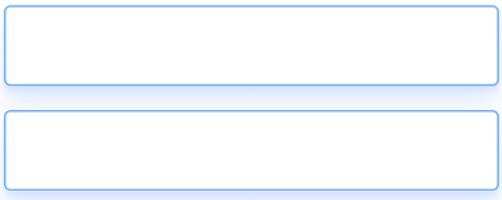
```
.container {
  display: flex;
  height: 300px;
  justify-content: space-between;
  align-items: flex-end;
}

.item { width: 14%; }
.item-1 { height: 40%; }
.item-2 { height: 50%; }
.item-3 { height: 60%; }
.item-4 { height: 20%; }
.item-5 { height: 30%; }
```

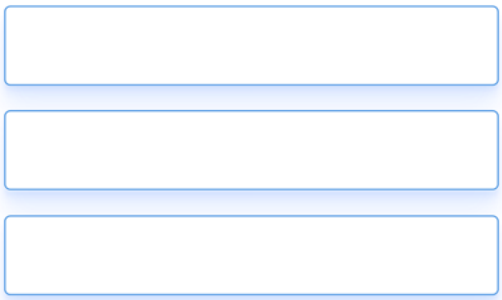
Graph: horizontal bars

Using the same technique as for vertical bars, we can simply add `flex-direction` on the `container` with a value of `column` to create a set of horizontal bars. `flex-direction` can have a value of `row` (default) or `column`, where a `row` runs horizontally (→) and a `column` runs vertically (↓). You can also reverse the direction of both by using `row-reverse` (←) and `column-reverse` (↑) respectively.

```
.container {
  display: flex;
  height: 300px;
  justify-content: space-between;
  flex-direction: column;
}

.item { height: 14%; }
.item-1 { width: 40%; }
.item-2 { width: 50%; }
.item-3 { width: 60%; }
.item-4 { width: 20%; }
.item-5 { width: 30%; }
```

Vertical stack (centered)





To create a vertical stack and make the items run from top to bottom all we need to do is to change the `flex-direction` from the default value of `row` to `column`:

```
.container {
  display: flex;
  flex-direction: column;
  align-items: center;
}

.item {
  height: 40px;
  margin-bottom: 10px;
}
```

Masonry (or mosaic)

You've probably seen masonry (or mosaic) layouts all over the internet, but it's likely that they were all powered by Masonry or a similar JavaScript library. Flexbox might seem like a great candidate to finally create this layout with CSS only, and it's certainly possible, but it's surprisingly hacky.

One of the main challenges of creating a masonry layout with flexbox is that to make an item affect the positioning of an item above and below it we need to change the `flex-direction` of the container to `column`, which makes items run from top to bottom. This creates a layout that looks great, and similar to the masonry effect, but it could be confusing for users; it creates an unexpected ordering of elements. If you read from left to right the elements would seem to be shuffled and appear in a seemingly random order, for example 1, 3, 6, 2, 4, 7, 8, etc.

Luckily, we can use the `order` property to change in which order elements are rendered. We can combine that property with some clever use of the `nth-child()` selector to order items dynamically depending on their original order. If we want to create a masonry layout with three columns we can divide all the items into three "groups", like so:

```
/* Re-order items into rows */
.item:nth-child(3n+1) { order: 1; }
.item:nth-child(3n+2) { order: 2; }
.item:nth-child(3n)   { order: 3; }

/* Force new columns */
.container::before,
.container::after {
  content: "";
  flex-basis: 100%;
  width: 0;
  order: 2;
}
```

I've written another post that explains in detail how this works and why. This will set the `order` to `1` for the 1st element, 4th element, 7th element, etc. Elements with the same `order` value will be ordered on ascending order according to the source code order, or which value was *set* first, so in this example we're producing three sets ordered like so: `1, 4, 7, 10` (3n+1) with `order: 1`, `2, 5, 8` (3n+2) with `order: 2`, and `3, 6, 9` (3n) with `order: 3`. These three groups represent our three columns. Put together the order becomes `1, 4, 7, 10, 2, 5, 8, 3, 6, 9`.

If we make sure to render each of those groups as one column (no more, no less), it'll create the illusion that the items have returned to their original

order when you read from left to right. If we visually parse the grid as rows, the first row will contain the first element from every group (`1, 2, 3`), the second row will contain the second element from every group (`4, 5, 6`), and so on so forth. We then insert elements between the columns that take up 100% of the parent's height, which force the columns to line break and not accidentally merge with adjacent columns. Here's the full CSS snippet:

```
.container {
  display: flex;
  flex-flow: column wrap;
  align-content: space-between;
  height: 580px;
}

.item {
  width: 32%;
  margin-bottom: 2%; /* (100-32*3)/2 */
}

/* Re-order items into rows */
.item:nth-child(3n+1) { order: 1; }
.item:nth-child(3n+2) { order: 2; }
.item:nth-child(3n)   { order: 3; }

/* Force new columns */
.container::before,
.container::after {
  content: "";
  flex-basis: 100%;
  width: 0;
  order: 2;
}
```

Visually this achieves something that is very close to the masonry effect.

How is the tab order affected? Luckily, not at all. `order` only changes the visual representation of objects, not the tab order, so tabbing through the grid will work as intended.

If you want to make a masonry layout with more than three columns (or want a better understanding of how this works) I recommend reading the dedicated post on how to create masonry layouts with CSS.

Published in:

css / tutorial

15/04/2019

Subscrib

Say hi.

hello@ tobiasahlin.com

I love to design and make things. I speak, teach, and consult at tech companies and startups, e.g. Spotify, Minecraft, GitHub, and Hyper Island. Say hi!

Tobias Ahlin Bjerrome
Stockholm, Sweden