

## Why choose flexbox?

The reason you'd choose to use flexbox is because you want to lay a collection of items out in one direction or another; you want to control the dimensions of the items in that one dimension, or control the spacing between items. These are the uses flexbox was designed for.

Sometimes other layout modes like Grid Layout would fit the need better.

## Navigation

A common pattern for navigation is to have a list of items displayed as a horizontal bar. This was hard to achieve before flexbox. It is the ideal flexbox use case.

When we have a set of items that we want to display horizontally, we may end up with additional space. We could either display the space outside of the items (spacing them out with whitespace between or around them) or absorb the extra space inside the items (then we would need a method of allowing the items to grow and take up this space).

## Space distributed outside the items

To distribute the space between or around the items we use the alignment properties in flexbox, and the `justify-content` property.

## Space distributed within the items

A different pattern for navigation would be to distribute the available space within the items themselves, rather than create space between them. In this case we would use the `flex` properties to allow items to grow and shrink in proportion to one another.

In the following picture, the `flex-basis` of `auto` causes the `width` of each `li` to take up as much space as is needed based on its contents. Note that there is no growth here because `flex-grow` is `0`.

Page 1 Page 2 Page 3 is longer Page 4

```
nav ul {  
  display: flex;  
}  
  
nav li {  
  flex: 0 1 auto;  
}
```

```
<nav>  
  <ul>  
    <li><a href="#">Page 1</a></li>  
    <li><a href="#">Page 2</a></li>  
    <li><a href="#">Page 3 is longer</a></li>  
    <li><a href="#">Page 4</a></li>  
  </ul>  
</nav>
```

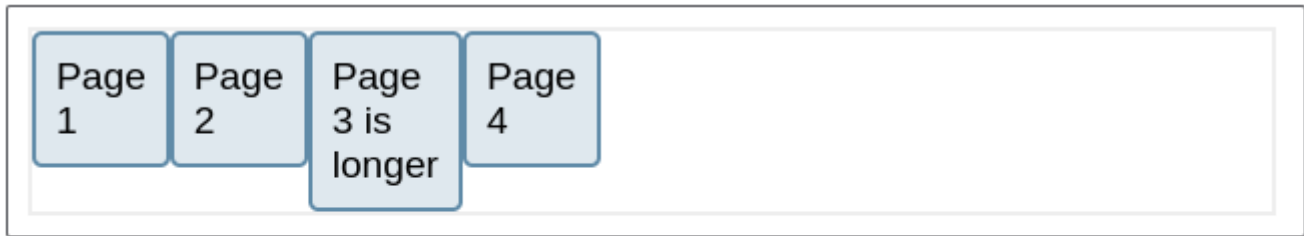
In this next picture, the elements eat the remaining space equally because `flex-grow` is `1`. The third `li` is still longer because it started longer. Since it eats the same amount of the extra space it ends up longer.

Page 1 Page 2 Page 3 is longer Page 4

```
nav ul {  
  display: flex;  
}  
  
nav li {  
  flex: 1 1 auto;  
}
```

```
<nav>  
  <ul>  
    <li><a href="#">Page 1</a></li>  
    <li><a href="#">Page 2</a></li>  
    <li><a href="#">Page 3 is longer</a></li>  
    <li><a href="#">Page 4</a></li>  
  </ul>  
</nav>
```

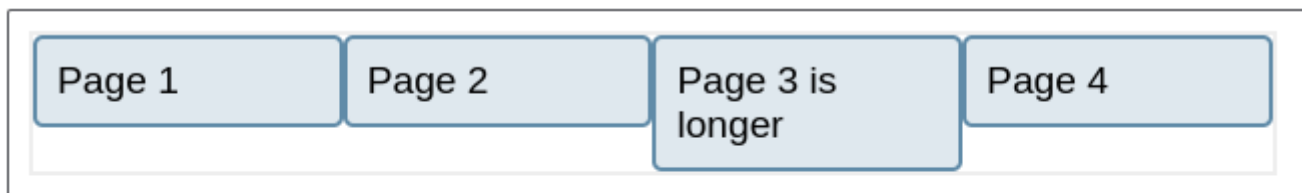
In this one, `flex-basis` is `0` so the `width` would tend towards `0` but there is a `min-width` that it won't go below (unless we change `min-width` explicitly to be `0` for instance). For an element containing text, the minimum width is the length of the longest unbreakable string of characters (the word 'longer' here).



```
nav ul {  
  display: flex;  
}  
  
nav li {  
  flex: 0 1 0;  
}
```

```
<nav>  
  <ul>  
    <li><a href="#">Page 1</a></li>  
    <li><a href="#">Page 2</a></li>  
    <li><a href="#">Page 3 is longer</a></li>  
    <li><a href="#">Page 4</a></li>  
  </ul>  
</nav>
```

Finally, `flex-basis` of `0` causes the items to shrink to `0` but then with `flex-grow: 1`, they expand equally by eating the same extra space so they are distributed evenly. Note that here the growth started from `0` not the `min-width` so they end up equally sized (i.e. the third `li` is not bigger).



```
nav ul {  
  display: flex;  
}
```

```
nav li {  
  flex: 1 1 0;  
}
```

```
<nav>  
  <ul>  
    <li><a href="#">Page 1</a></li>  
    <li><a href="#">Page 2</a></li>  
    <li><a href="#">Page 3 is longer</a></li>  
    <li><a href="#">Page 4</a></li>  
  </ul>  
</nav>
```

## Split navigation

Another way to align items on the main axis is to use auto margins.

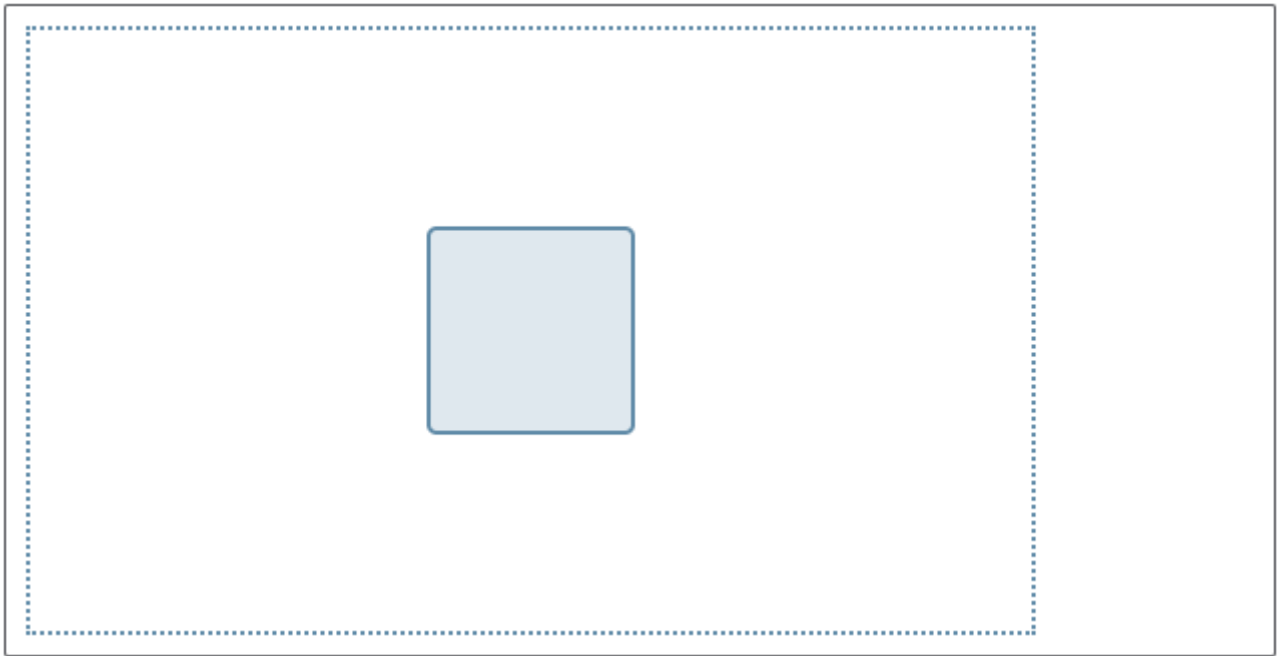
[Page 1](#)[Page 2](#)[Page 3 is longer](#)[Page 4](#)

```
nav ul {  
  display: flex;  
  gap: 20px;  
}  
  
.push-right {  
  margin-left: auto;  
}
```

```
<nav>  
  <ul>  
    <li><a href="#">Page 1</a></li>  
    <li><a href="#">Page 2</a></li>  
    <li><a href="#">Page 3 is longer</a></li>  
    <li class="push-right"><a href="#">Page 4</a></li>  
  </ul>  
</nav>
```

## Center item

Before flexbox, developers would joke that the hardest problem in web design was vertical centering. This has now been made straightforward using the alignment properties in flexbox.



```
.box {  
  display: flex;  
  align-items: center;  
  justify-content: center;  
}  
  
.box div {  
  width: 100px;  
  height: 100px;  
}
```

```
<div class="box">  
  <div></div>  
</div>
```

In the future this probably will be done using Box Alignment properties that will be implemented in Block layout. But for now however, we should use flexbox to accomplish this.

## Card layout pushing footer down

You may be using flexbox or CSS Grid to lay out a list of card components, but these layout methods only work on direct children of the flex or grid component. So if you have variable amounts of content, the card will stretch to the height of the grid area or flex container. Any content inside uses regular block layout, meaning that on a card with less content the footer will rise up on the bottom of the content rather than stick to the bottom of the card.

This card doesn't have much content.

Card footer

This card has a lot more content which means that it defines the height of the container the cards are in. I've laid the cards out using grid layout, so the cards themselves will stretch to the same height.

Card footer

Flexbox can solve this. We can mark each the cards themselves flex containers with `flex-direction: column`. Then set the content area to `flex: 1` aka `flex: 1 1 0` (the item can grow and shrink from a flex-basis of `0`). Since this is the only item that can grow, it takes up all available space in the card and pushes the footer to the bottom.

This card doesn't have much content.

Card footer

This card has a lot more content which means that it defines the height of the container the cards are in. I've laid the cards out using grid layout, so the cards themselves will stretch to the same height.

Card footer

```
.card {  
  display: flex;  
  flex-direction: column;  
}  
  
.card .content {  
  flex: 1 1 auto;  
}
```

```
<div class="cards">  
  <div class="card">  
    <div class="content">  
      <p>This card doesn't have much content.</p>  
    </div>  
    <footer>Card footer</footer>  
  </div>  
  <div class="card">  
    <div class="content">  
      <p>This card has a lot more content which means that it defines  
the height of the container the cards are in. I've laid the cards out  
using grid layout, so the cards themselves will stretch to the same  
height.</p>  
    </div>  
    <footer>Card footer</footer>  
  </div>  
</div>
```

## Media objects


The media object is a common pattern in web design; this pattern has an image or other element to one side and text to the right. Ideally a media object should be able to be flipped; moving the image from left to right.

We see this pattern everywhere, used for comments, and anywhere we need to display images and descriptions. With flexbox we can allow the part of the media object containing the image to



take its sizing information from the image, and then the body of the media object flexes to take up the remaining space.

In this example, we have used the alignment properties to align the items on the cross axis to `flex-start`, and then set the `.content` flex item to `flex: 1`. As with the column layout card pattern above, using `flex: 1` means this part of the card can grow.



This is the content of my media object. Items directly inside the flex container will be aligned to flex-start.

```
img {
  max-width: 100%;
}

.media {
  display: flex;
  align-items: flex-start;
}

.media .content {
  flex: 1;
  padding: 10px;
}
```

```
<div class="media">
  <div class="image"></div>
  <div class="content">This is the content of my media object. Items
  directly inside the flex container will be aligned to flex-start.</div>
</div>
```

To prevent the image from growing too large, add a `max-width` to the image. Since that side of the media object is using the initial values of flexbox, it can shrink but not grow, and uses a `flex-basis` of `auto`. Any `width` or `max-width` applied to the image will become the `flex-basis`.

You could also allow both sides to grow and shrink in proportion. If you set both sides to `flex: 1` (and remove `max-width` on the image), they will grow and shrink from a `flex-basis` of 0, so you will end up with two equal-sized columns. You could either take the content as a guide and set both to `flex-auto`, in which case they would grow and shrink from the size of the content or any size applied directly to the flex items such as a `width` on the image.

You could also give each side different `flex-grow` factors, for example setting the side with the image to `flex: 1` and the content side to `flex: 3`. This will mean they use a `flex-basis` of `0` but distribute that space at different rates according to the `flex-grow` factor you have assigned.

## Flipping the media object

To switch the display of the media object so that the image is on the right and the content is on the left we can use the `flex-direction` property set to `row-reverse`. The media object now displays the other way around.

This is the content of my media object. Items directly inside the flex container will be aligned to flex-start.



```
img {
  max-width: 100%;
}

.media {
  display: flex;
  align-items: flex-start;
}

.media.flipped {
  flex-direction: row-reverse;
}

.media .content {
  flex: 1;
  padding: 10px;
}
```

```
<div class="media flipped">
  <div class="image"></div>
  <div class="content">This is the content of my media object. Items
  directly inside the flex container will be aligned to flex-start.</div>
</div>
```

## Form controls

Flexbox is particularly useful when it comes to styling form controls. Forms have lots of markup and lots of small elements that we typically want to align with each other. A common pattern is

to have an `<input>` element paired with a `<button>`, perhaps for a search form or where you want your visitor to enter an email address.

Using flexbox, we use the flex properties to allow the `<input>` field to grow, while the button does not grow. This means we have a pair of fields, with the text field growing and shrinking as the available space changes.



```
.wrapper {  
  display: flex;  
}  
  
.wrapper input[type="text"] {  
  flex: 1 1 auto;  
}
```

```
<form class="example">  
  <div class="wrapper">  
    <input type="text" id="text">  
    <input type="submit" value="Send">  
  </div>  
</form>
```

You could add a label or icon to the left as easily as we popped the button unto the right. The stretchy input field now has a little less space to play with but it uses the space left after the two items are accounted for.

Label

Send

```
.wrapper {  
  display: flex;  
}  
  
.wrapper input[type="text"] {  
  flex: 1 1 auto;  
}
```

```
<form class="example">  
  <div class="wrapper">  
    <label for="text">Label</label>  
    <input type="text" id="text">  
    <input type="submit" value="Send">  
  </div>  
</form>
```

Patterns like this can make it much easier to create a library of form elements for your design, which easily accommodate additional elements being added. You are taking advantage of the flexibility of flexbox by mixing items that do not grow with those that do.

## Conclusion

Quite often you have more than one choice as to how to implement something using flexbox. You can mix items that cannot stretch with those that can, use the content to inform the size, or allow flexbox to share out space in proportion.