UNRAVELING FLEXBOX

The ultimate guide to building modern CSS layouts with flexbox

Landon Schropp

Version 1.4

INTRODUCTION

Have you ever spent hours agonizing over a CSS layout that just wouldn't work? Have you struggled with columns, vertical centering, floats or inline displays? Have you even, *gasp*, given up and used tables for your layouts?

It’s time to say goodbye to all that pain. Flexbox is a CSS layout specification that makes it easy to construct dynamic layouts. It's a set of tools that gives you more flexibility and power with CSS than you've ever had before. With flexbox, vertical centering, same-height columns, reordering, and direction agnosticism are a piece of cake.

There's a popular myth floating around that flexbox isn't ready for prime time yet. Wrong! **93% of people are now running a browser that supports flexbox, and that number is growing every day.** That's better than the support for the HTML5 `<video>` element! You can use flexbox today and it will work almost everywhere!

This book is your guide to mastering flexbox. It will teach you the ins and outs of all the properties and how they interact together. More importantly, it will show you how to apply them to real layouts.

What's in the Book?

This book is about teaching you to use flexbox in the *real world*. The examples in each chapter are as true to life as I could make them. Many of them are layouts I've previously built for paying clients. You can use what you learn here directly in your projects.

Here's the breakdown:

Chapter 1: Getting Dicey
In this chapter, you'll build your very first layout, the faces of dice!

Chapter 2: Crafting Twelve-Column Layouts
Learn how you can use flexbox to build twelve-column layouts you've always needed a grid system for in the past.

Chapter 3: Building a Video Player
Build a video player with flexbox that'll make YouTube's developers jealous.

Chapter 4: Say Goodbye to Vendor Prefixes
I'll show you how to set up your environment so you can ignore all vendor prefixes. You'll write your code once, and it will work everywhere!

Chapter 5: Breaking Free From Twelve-Column Layouts
You'll go beyond twelve-columns and build a cool calendar layout in the process.

Chapter 6: Perfect Pricing
Create a pricing layout that will feel right at home on any marketing site.

Chapter 7: Flexbox Forms
Flexbox isn't just for full-page layouts! In this chapter, you'll learn how to use flexbox to build small, reusable form controls.

Chapter 8: Responsive Design
Learn how to harness flexbox for responsive layouts that work great on both desktop and mobile.

Chapter 9: Wrapping Like a Boss
Say goodbye to floats and clearfixes. You'll be using flexbox's fantastic wrapping controls from now on.

Chapter 10: Progressive Enhancement
You'll learn how to take advantage of the flexbox goodness and still support Internet Explorer 9 and below!

Chapter 11: Ordering
The order of the elements on your screen doesn't have to match the order in the HTML. This chapter will show you how to reorder these elements with flexbox.

Chapter 12: Cross-Browser Testing
You'll learn how to test your code across every major browser and device.

Chapter 13: How to Write a Grid System
Have you ever wondered how grid systems like 960gs work? In this chapter you'll create your very own flexbox grid system.

Chapter 14: Minesweeper
You'll use everything you've learned in this book to build an awesome Minesweeper layout!

When a book contains too many details, it's difficult to catch the important points. In this book I've omitted styles that don't apply to flexbox, such as typography, colors and borders. If you'd like to see all of the styles for a chapter, take a look at the code examples.

Code Examples

The examples for this book are powered by Middleman, a static site generator that makes it easy to build HTML and CSS websites. There are several ways for you to access the example code:

- View the source on GitHub.
- Download the compiled build.
- Browse the hosted examples.
- Run the example server yourself.

The last option is trickier than the first three, so I'd only recommend it if you're feeling ambitious. If you're a Mac user, I've recorded a video to make the installation process easier for you. If you're a Windows user, there's currently a bug in Middleman preventing you from running the examples.

The first step is to install the project's dependencies:

- Ruby
- Git
- NodeJS
- Bundler

Next, clone the project's Git repository and switch into that directory.

```
git clone "https://github.com/"\
"LandonSchropp/unraveling_flexbox"
cd unraveling_flexbox
```

Use Bundler to install the project's gem dependencies.

```
bundle install
```

Finally, start up the Middleman server.

```
bundle exec middleman
```

If everything's set up correctly, you can navigate to http://localhost:4567 to view the examples.

Acknowledgements

I'd like to thank my wife, Danielle, for all her support in writing this book. I've spent too many evenings hunched over my computer instead of hanging out with her. Not only did she tolerate my insanity, but she gave a large portion of her own time to editing this book. Love you Danielle!

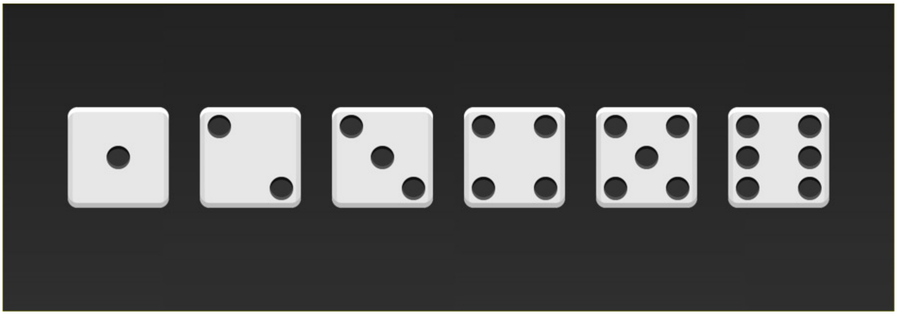
I'd also like to thank my beta readers, especially Joshua, Darrin, Andrew, Duc and Christine. You guys made a huge difference in the quality of this book, and I really appreciate it!

Enough Chitchat

Let's dive in. Welcome to Unraveling Flexbox!

CHAPTER 1

Getting Dicey


The six dice faces

The best way to learn flexbox is to roll up your sleeves and write some code. In this chapter, I'll walk you through your very first flexbox layout: the faces of dice!

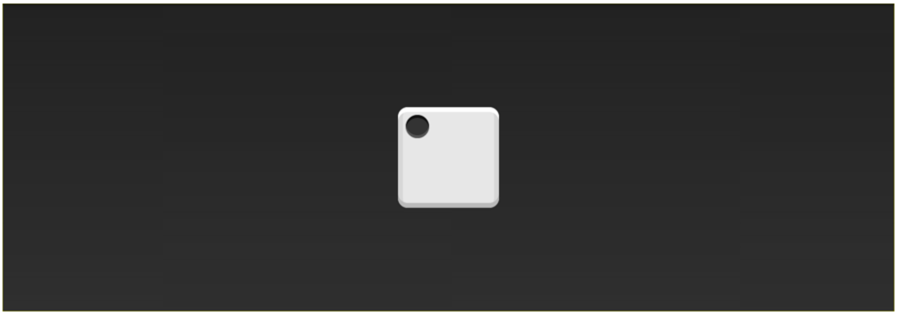
The First Face

A standard playing die consists of six faces (sides). Each face has a number of pips (dots) which determine the value of the side. The first side consists of a single pip in the center of the face.

Let's start by writing the HTML for the first face.

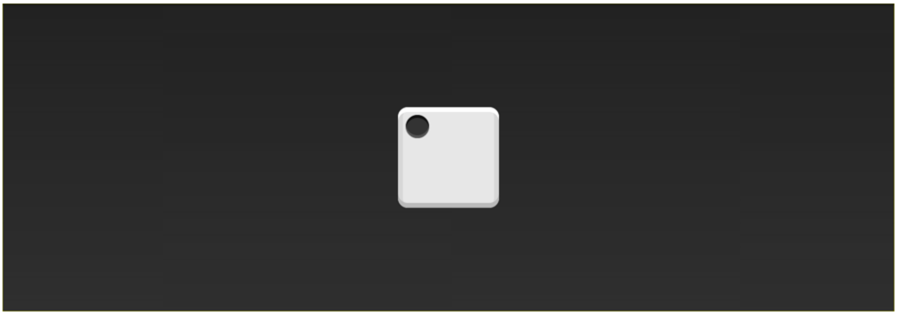
```
<div class="first-face">
  <span class="pip"></span>
</div>
```

To make life a little easier, I’ve added the basic styles for the faces and the pips. Here's what it looks like:

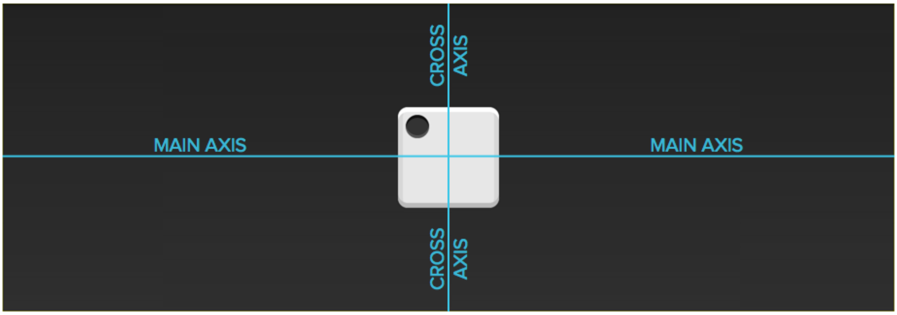


The first step is to tell the browser to make the face a flexbox container.

```
.first-face {
  display: flex;
}
```



It doesn't look any different, but there's a lot going on under the hood.

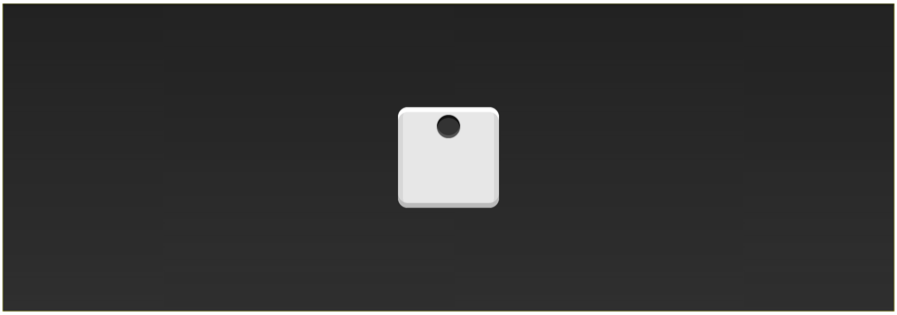


The flexbox container's main axis and cross axis

The `first-face` container now has a horizontal main axis. The main axis of a flex container can be horizontal or vertical. The default is horizontal. If we added another pip to the face, it would show up to the right of the first one. The container also has a vertical cross axis. The cross axis is always perpendicular to the main axis.

The `justify-content` property defines the alignment along the main axis. Since we want to center the pip along the main axis, we'll use the `center` value.

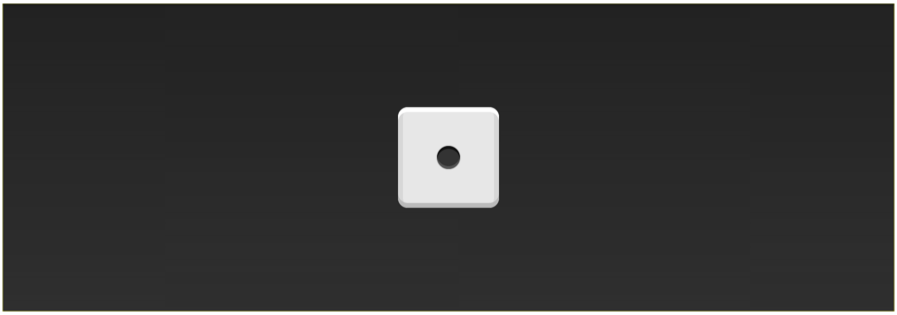
```
.first-face {
  display: flex;
  justify-content: center;
}
```



All right! Since the main axis is horizontal, the pip is now centered in the parent element.

The `align-items` property dictates how the items are laid out along the cross axis. Because we want the pip to center along this axis, use the `center` value here too.

```
.first-face {
  display: flex;
  justify-content: center;
  align-items: center;
}
```



And just like that, the pip is centered! Horizontally and Vertically centering an element was one of the hardest tricks to accomplish in CSS before flexbox, and you've done it in a few lines of code!

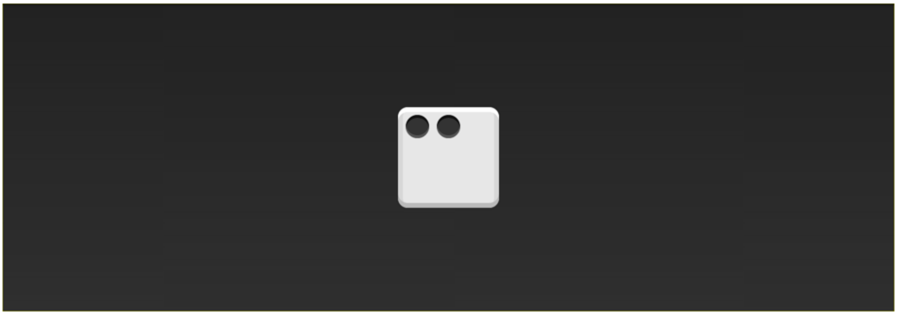
Getting Trickier

On the second face of a die, the first pip is in the top left corner and the second is in the bottom right. That's also pretty easy to do with flexbox!

Again, start with the markup and the basic CSS.

```
<div class="second-face">
  <span class="pip"></span>
  <span class="pip"></span>
</div>
```

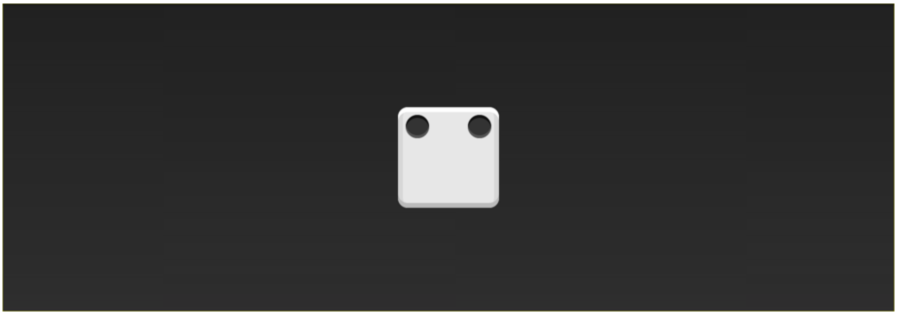
```
.second-face {
  display: flex;
}
```



Now you have two pips right next to each other. This time around, the pips should be on opposite sides of the face. There's a value for `justify-content` that will let us do just that: `space-between`.

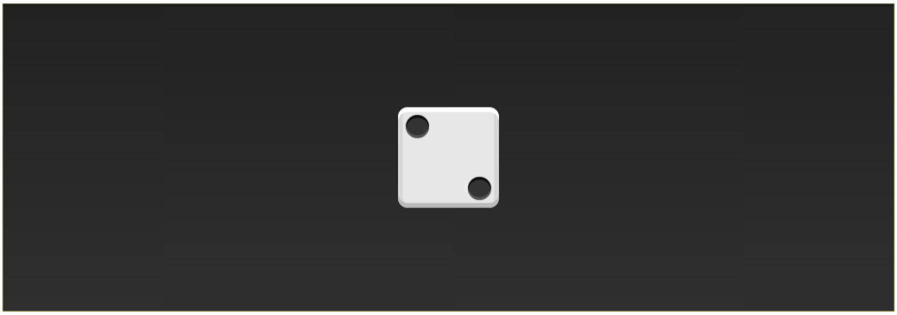
The `space-between` value evenly fills the space between flex items. Since there are only two pips, this pushes them away from each other.

```
.second-face {
  display: flex;
  justify-content: space-between;
}
```



Here's where we run into a problem. Unlike before, you can't set `align-items` because it will affect both pips. Luckily, flexbox includes `align-self`. This handy property lets you align individual items in a flex container along the cross axis! The value you want for this property is `flex-end`.

```
.second-face {
  display: flex;
  justify-content: space-between;
}

.second-face .pip:nth-of-type(2) {
  align-self: flex-end;
}
```



Looks good!

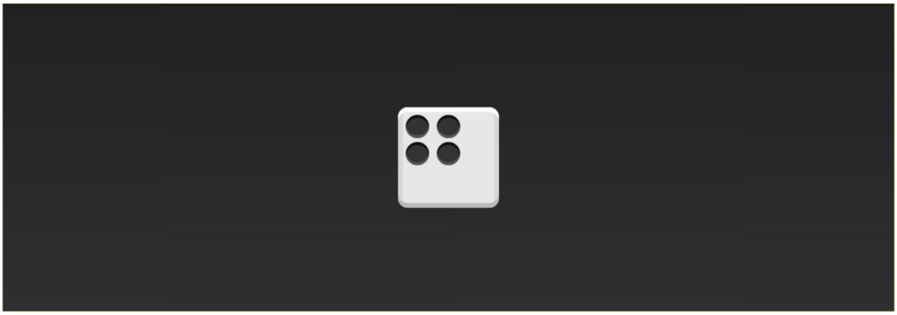
Horizontal and Vertical Nesting

Let's skip the third face and tackle the fourth. This one is a little trickier than the others because we need to support two columns, each with two pips.

There are two things about flexbox that will save you here: flex containers can have vertical or horizontal content, and flex containers can be nested.

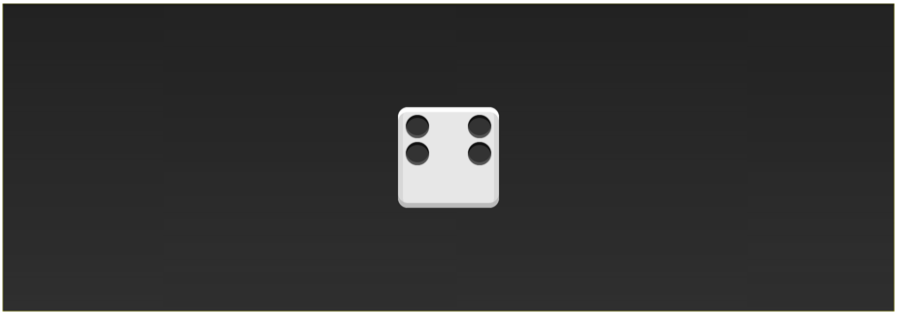
Unlike before, the markup will now include columns.

```
<div class="fourth-face">
  <div class="column">
    <span class="pip"></span>
    <span class="pip"></span>
  </div>
  <div class="column">
    <span class="pip"></span>
    <span class="pip"></span>
  </div>
</div>
```



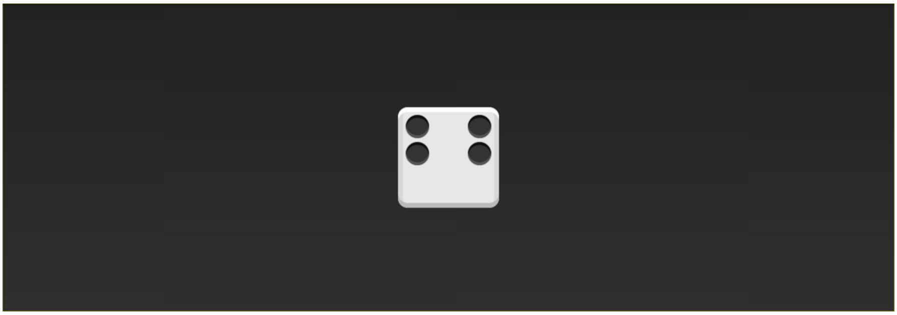
Since you want the two columns to be on opposite sides, go ahead and use `justify-content: space-between` like you did before.

```
.fourth-face {
  display: flex;
  justify-content: space-between;
}
```



Next, you need to make the columns flex containers. It might seem like they already are, but remember that you haven't set `display: flex` yet. You can use the flex-direction property to to set the direction of the main axis to column.

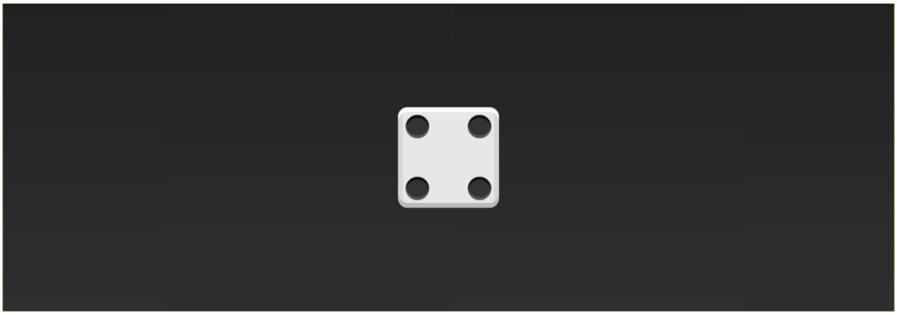
```
.fourth-face {
  display: flex;
  justify-content: space-between;
}

.fourth-face .column {
  display: flex;
  flex-direction: column;
}
```



It doesn't look any different, but the columns are now flex containers. Notice how you stuck a flex container directly inside another flex container? That's okay! Flexbox doesn't care if the containers are nested.

The final step is to space the pips apart from each other. Since the main axis for the columns is vertical, you can use `justify-content` again.

```
.fourth-face {
  display: flex;
  justify-content: space-between;
}

.fourth-face .column {
  display: flex;
  flex-direction: column;
  justify-content: space-between;
}
```



Note: This face could have been built without columns by using wrapping. I'll cover wrapping in more detail in Chapter 9.

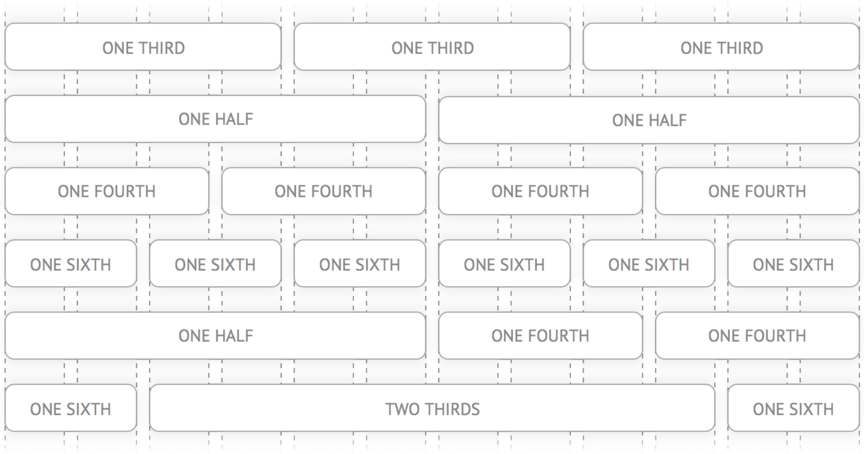
Wrapping Up

Woohoo! Three faces down and three to go. At this point, you have everything you need to build the other three. Give it a shot! When you're done, take a look at the code examples for the answers.

CHAPTER 2

Crafting Twelve-Column Layouts

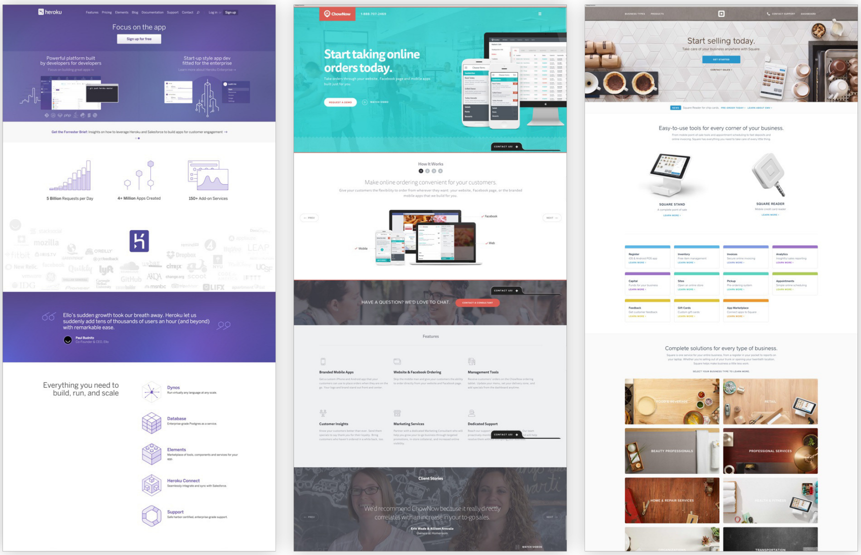
In a twelve-column layout, the page is broken apart into twelve invisible columns. These columns have small amounts of space between them, called *gutters*. The page is divided into rows, and the containers in the rows take up a certain number of columns.



A twelve-column grid with columns and gutters

If you look for them, you'll start to see twelve-column layouts *everywhere*. Take a look at these landing pages from Heroku, ChowNow and Square. Notice how the sections are broken up into halves, thirds and fourths?

In this chapter, I'll show you how to use the `flex-grow`, `flex-shrink` and `flex-basis` properties to build twelve-column layouts, without the need for a library!

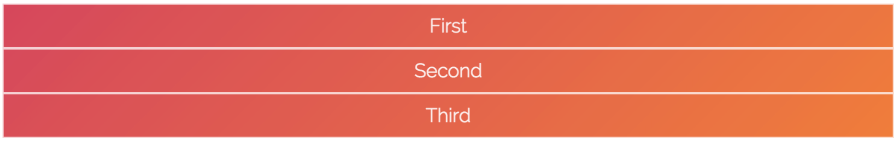

Focus on the app
Start taking online orders today.
Start selling today.
Easy-to-use tools for every corner of your business.
Complete solutions for every type of business.

Examples of twelve-column layouts from Heroku, ChowNow and Square

Setting Up the Container

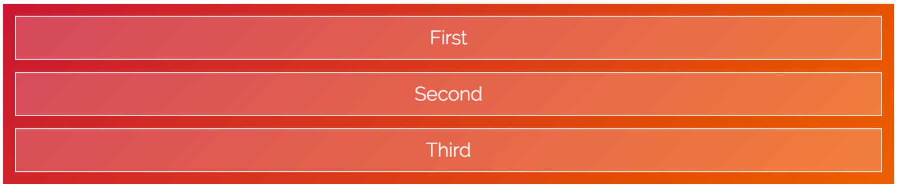
Let's say you want each of the `<div>` elements in the following HTML to take up a third of the `<section>`.

```
<section>
  <div class="column">First</div>
  <div class="column">Second</div>
  <div class="column">Third</div>
</section>
```



By default, the `<section>` element takes up 100% of the width of the screen. Start by limiting its width to 740 pixels. While you're at it, also add gutters around the columns.

```
section {
  max-width: 740px;
  margin: 0 auto;
}

.column {
  margin: 10px;
}
```

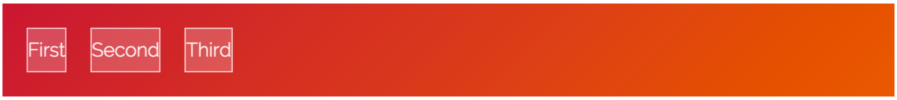


Pop open the code examples and try dragging your browser window until it's smaller than 740 pixels. Notice how the `<section>` gets smaller as the screen shrinks, but stays fixed when the screen is larger than 740 pixels?

Flexin' It Up

Make the `<section>` a flex container like you did in Chapter 1.

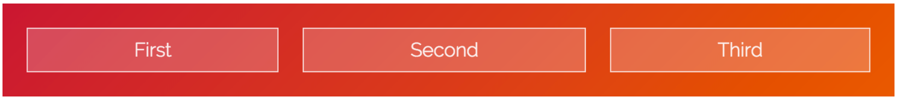
```
section {
  max-width: 740px;
  margin: 0 auto;
  display: flex;
}
```



By default, flexbox sets the widths of the columns to the size of their content. You can change this behavior by using the `flex-grow` and `flex-shrink` properties.

The `flex-grow` property tells flexbox how to grow the item to take up additional space, if necessary. `flex-shrink` tells flexbox how to shrink when necessary. Since we want the columns to behave the same while growing and shrinking, set both of these properties to `1`.

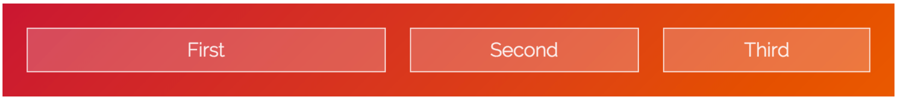
```
.column {
  margin: 10px;
  flex-grow: 1;
  flex-shrink: 1;
}
```



Woohoo! The flexbox container now fills up three columns. The values for `flex-grow` and `flex-shrink` are *proportional*, meaning they change relative to other items in the flex container. Flexbox adds the values for the properties and then divides each column's value by that sum. So each column takes up `1 ÷ (1 + 1 + 1)`, or ⅓ of the total space.

What happens if one of the columns has a different value?

```
.column:first-of-type {
  flex-grow: 2;
  flex-shrink: 2;
}
```



The first column takes up the same amount of space as the other two. That's because the values add up to `4`, so the first column is:

```
2 ÷ (2 + 1 + 1) = ½
```

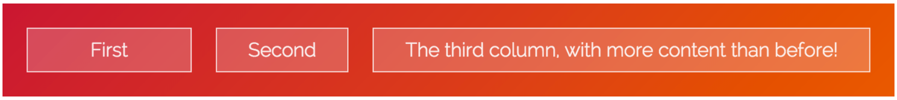
The other two are:

```
1 ÷ (2 + 1 + 1) = ¼
```

All About That Basis

If you look closely at the last example, you'll notice that the first column doesn't quite cover half of the container. If you add more content to the third column, you can really see the problem.

```
<section>
  <div class="column">First</div>
  <div class="column">Second</div>
  <div class="column">
    The third column, with more content than
    before!
  </div>
</section>
```



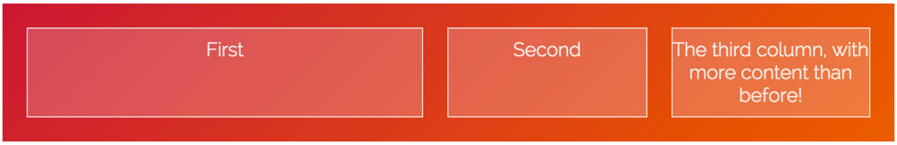
What's going on? Why is flexbox not flexing correctly?

It turns out flexbox doesn't distribute space evenly to each column. It figures out how much space each column *starts with*, specified by the `flex-basis` property. Then, the *remaining* space is distributed using the `flex-grow` and `flex-shrink` properties.

This might seem confusing, and that's because it is. The way this stuff adds up is [really damn complicated](), but don't worry, you don't need to understand the nuances to use flexbox.

Since we don't care about how much space the content originally takes up, set `flex-basis` to `0`.

```
.column {
  margin: 10px;
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0;
}

.column:first-of-type {
  flex-grow: 2;
  flex-shrink: 2;
  flex-basis: 0;
}
```



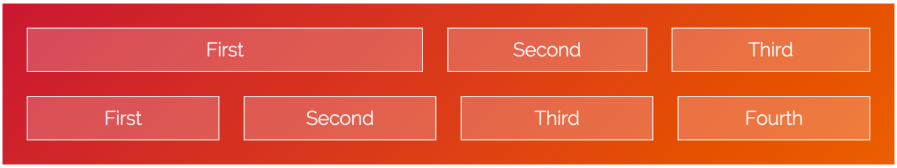
Tah-dah! It works! Well, kind of—there's one last thing to fix.

More Flex Basis

If you add another section below the first, you can see the problem.

```html
<section>
  <div class="column">First</div>
  <div class="column">Second</div>
  <div class="column">Third</div>
</section>
<section>
  <div class="column">First</div>
  <div class="column">Second</div>
  <div class="column">Third</div>
  <div class="column">Fourth</div>
</section>
```

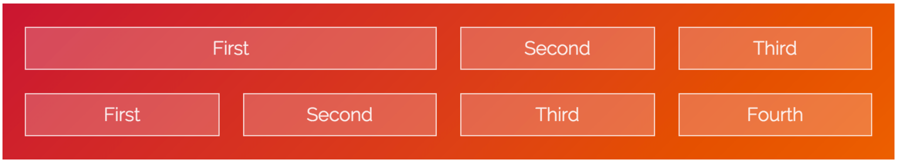
```css
.column {
  margin: 10px;
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0;
}

section:first-of-type .column:first-of-type {
  flex-grow: 2;
  flex-shrink: 2;
  flex-basis: 0;
}
```



Why don't the columns line up? It's because flexbox includes the padding, border and margin in the basis when it calculates how big the item should be.

The first and second columns in the second row have 22 pixels between them (20 pixels for the gutter and 2 pixels for the borders). We can add this missing space to the first column in the first row by setting `flex-basis` to `22px`.

```
section:first-of-type .column:first-of-type {
  flex-grow: 2;
  flex-shrink: 2;
  flex-basis: 22px;
}
```



Shorthand

Together, `flex-grow`, `flex-shrink` and `flex-basis` form the cornerstone of what makes flexbox flexible. Since these properties are so closely tied together, there's a handy shorthand property, `flex`, that lets you set all three. You can use it like this:

```
flex: <flex-grow> <flex-shrink> <flex-basis>;
```

We can rewrite our CSS to look like this:

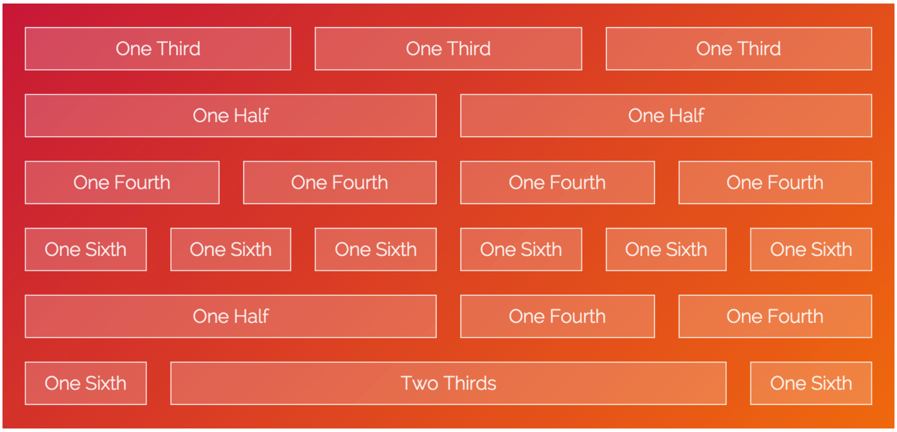
```
.column {
  flex: 1 1 0px;
}

section:first-of-type .column:first-of-type {
  flex: 2 2 22px;
}
```

Ahh, that's better. Why the `0px` in the first `flex` declaration? There's a bug in Internet Explorer 10 and 11 that ignores `flex` if the basis doesn't include a unit.

That's It!

You've covered a ton of great stuff in this chapter, including `flex-grow`, `flex-shrink` and `flex-basis`. You've also seen how these properties can be used to implement twelve-column layouts.

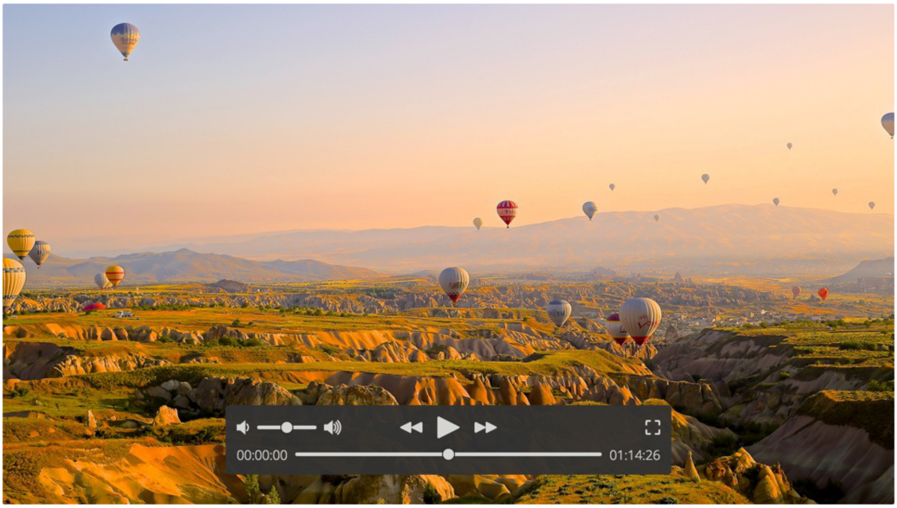
If you're looking for a challenge, try finishing off the entire grid. Here's what it looks like completed.



If you're still confused about how `flex-grow`, `flex-shrink` and `flex-basis` work, *don't worry*. These properties are the hardest thing to understand about flexbox. You'll be reviewing them again in later chapters, including the next chapter, where you'll build an awesome video player layout!

CHAPTER 3

Building a Video Player

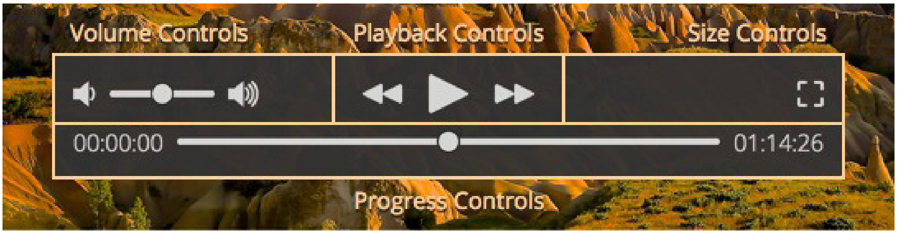


What's the best part about watching a movie? Is it the salty popcorn that coats your fingertips in hot, melted butter? How about the mountains of crunchy candy or the monolithic soda? Could it be the special effects and explosions, or the raw talent of the actors and actresses? Maybe it's the profound cinematography or the moving musical score?

Of course not! It's the playback controls for the video player, and in this chapter, you're going to learn how to make them! I'll show you how to build the killer layout you see above using flexbox!

Lights, Camera, Action!

If you look at the video player screenshot about, you'll notice that it can be cleanly divided into multiple sections.



Let's start by capturing this structure in HTML.

```
<div class="video-player">
  <img src="hot_air_balloons.jpg" alt="Video"
    width="960" height="540">

  <div class="controls-container">
    <div class="controls">
      <div class="top-controls">
        <div class="volume-controls"></div>
        <div class="playback-controls"></div>
        <div class="size-controls"></div>
      </div>
      <div class="progress-controls"></div>
    </div>
  </div>
</div>
```

Here you've created a container for the video player. Normally, inside that container you'd use a `<video>` element, but to make life easier we'll use an `<img>` element.

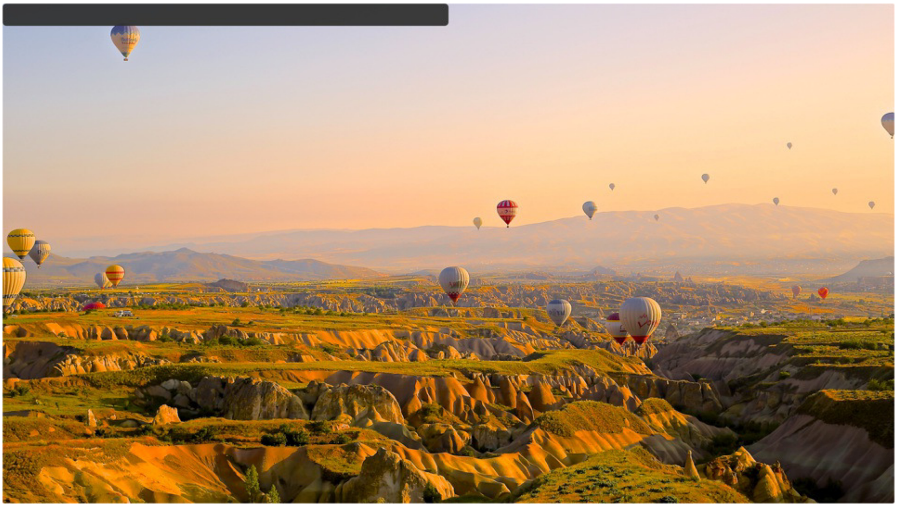
Inside the video player container is a `<div>` with a class of `controls-container`, which will be used for—you guessed it —containing the controls. The top row of the controls is split into the volume controls, the playback controls and the size controls. The bottom row is devoted to the progress controls.

The Container

The first thing you need to do is center the video player controls in the container. You can do this by absolutely positioning the controls container over the top of the video player. This allows the video player `<div>` to be determined by the size of the image inside of it. While you're at it, add some CSS to size the controls container so you can see it.

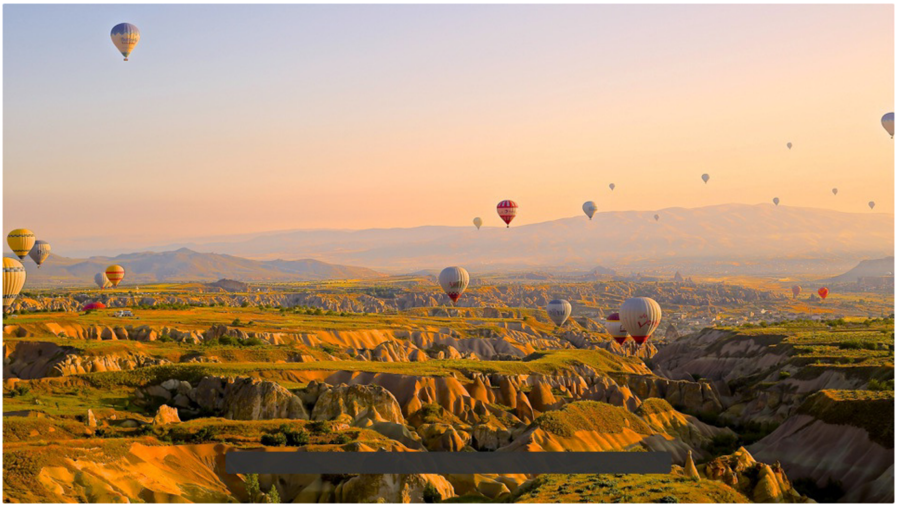
```
.video-player {
  position: relative;
}

.controls-container {
  position: absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
}
```

```
.controls {
  width: 480px;
  margin-bottom: 32px;
  padding: 12px 4px;
}
```



What we want is for the controls to be positioned in the bottom center of the controls container. You can accomplish that setting the control container's `display` property to `flex` and using `align-items` and `justify-content`.

```
.controls-container {
  ...

  display: flex;
  justify-content: center;
  align-items: flex-end;
}
```

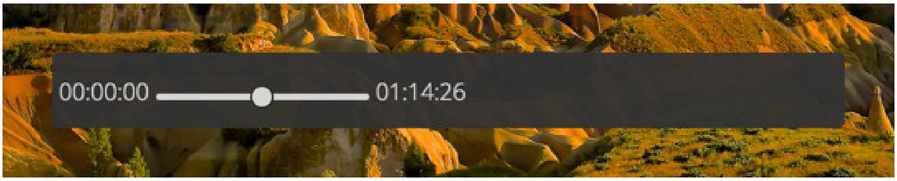


There you go! Now you have a nicely positioned `<div>` for your controls.

The Progress Controls

The next step is to build the progress controls. The HTML for these is pretty straightforward.

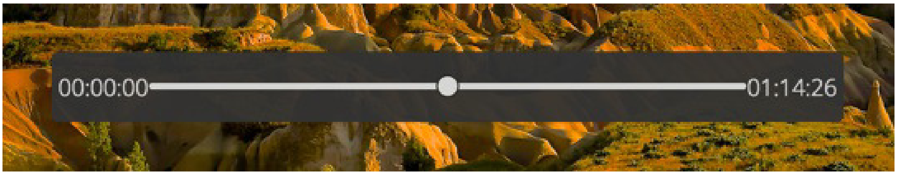
```
<div class="progress-controls">
  <span class="time-elapsed">00:00:00</span>
  <input type="range">
  <span class="time-remaining">01:14:26</span>
</div>
```



The idea here is to place the time elapsed and time remaining `<span>` elements on the left and right of the container, respectively. The `<input>` then fills up the remaining space.

```
.progress-controls {
  display: flex;
}

.time-elapsed, .time-remaining {
  flex: 0 0 auto;
}

.progress-controls input[type="range"] {
  flex: 1 1 0px;
}
```

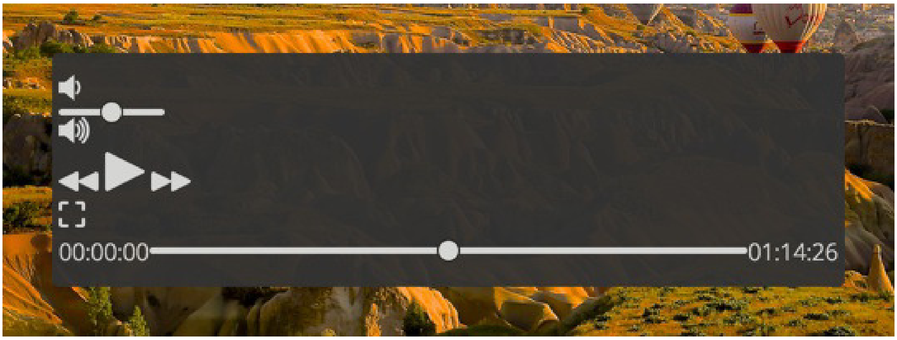
What's that `auto` value? Setting the `flex-basis` to `auto` tells flexbox to resize the container based upon the size of the content. In this case, the time elapsed and time remaining spans take up as much room as they need. Then, the progress controls container stretches to take up the rest of the space.



The Top Controls

The top controls are a little trickier than the bottom controls.

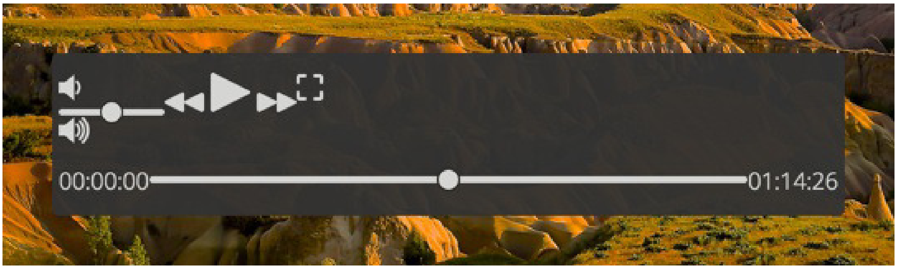
```
<div class="top-controls">
  <div class="volume-controls">
    <button>
      <img alt="Low Volume"
        src="low_volume.svg">
    </button>
    <input type="range">
    <button>
      <img alt="High Volume"
        src="high_volume.svg">
    </button>
  </div>
  <div class="playback-controls">
    <button>
      <img alt="Rewind" src="rewind.svg">
    </button>
    <button>
      <img alt="Play" src="play.svg">
    </button>
    <button>
      <img alt="Fast Forward"
        src="fast_forward.svg">
    </button>
  </div>
  <div class="size-controls">
    <button>
      <img alt="Fullscreen"
        src="fullscreen.svg">
    </button>
  </div>
</div>
```



The markup doesn't look very nice, but it'll do the job. It mainly consists of buttons containing images and `<div>` containers.

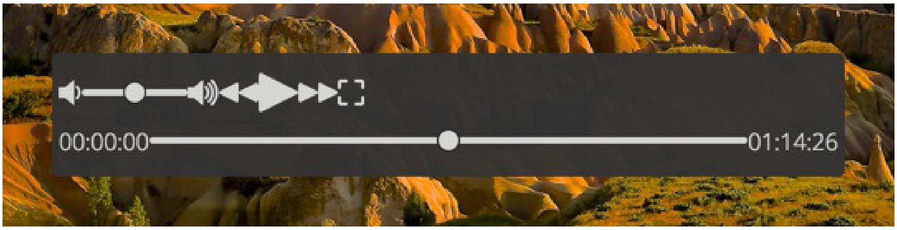
The first step in styling the top controls is to display them side by side. In order to do that, you need to set the top container's `display` to `flex`. Remember, the default value for `flex-direction` is `row`, so the container's contents will be displayed horizontally. While you're at it, add a little margin to the bottom of the top controls.

```
.top-controls {
  display: flex;
  margin-bottom: 8px;
}
```

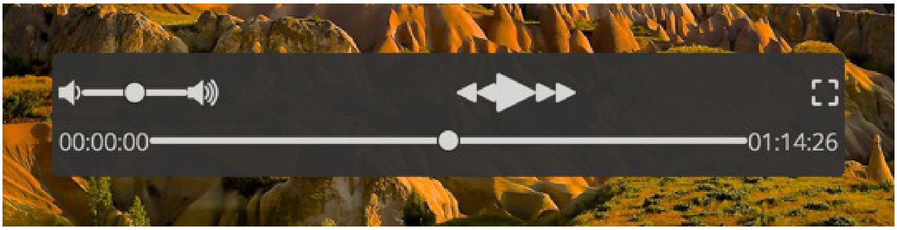


To make the volume controls, playback controls and size controls horizontal, you'll also make each a flex container. You can use `align-items` to vertically center their content.

```
.volume-controls,
.playback-controls,
.size-controls {
  display: flex;
  align-items: center;
}
```

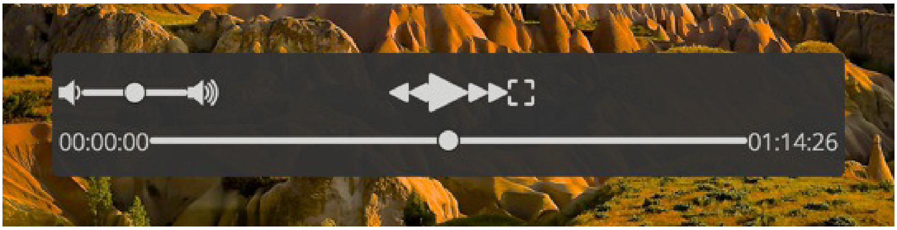


Next, you need to space them out. You may be thinking you can make the volume controls and size controls container size to their content, and have the playback controls stretch to fit the container using `flex-grow` and `flex-shrink`. However, if you try that, you'll end up with controls that look like this:



Notice how the playback controls aren't centered? Instead, you'll make the playback controls container size to its content and let the volume and size controls expand.

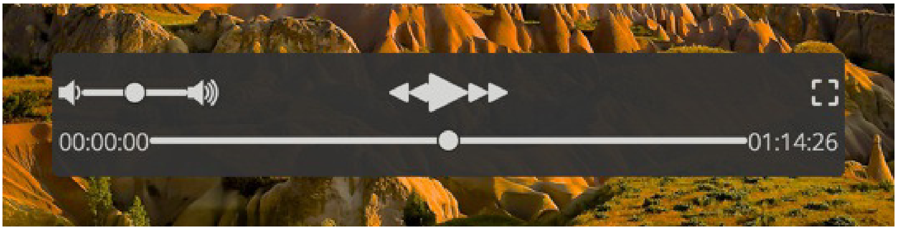
```
.playback-controls {
  flex: 0 0 auto;
}

.volume-controls, .size-controls {
  flex: 1 1 0px;
}
```



This works because the `flex-basis` of the playback controls is `auto`, so playback controls container is sized to the buttons it contains. The volume and size controls then evenly fill the remaining space.

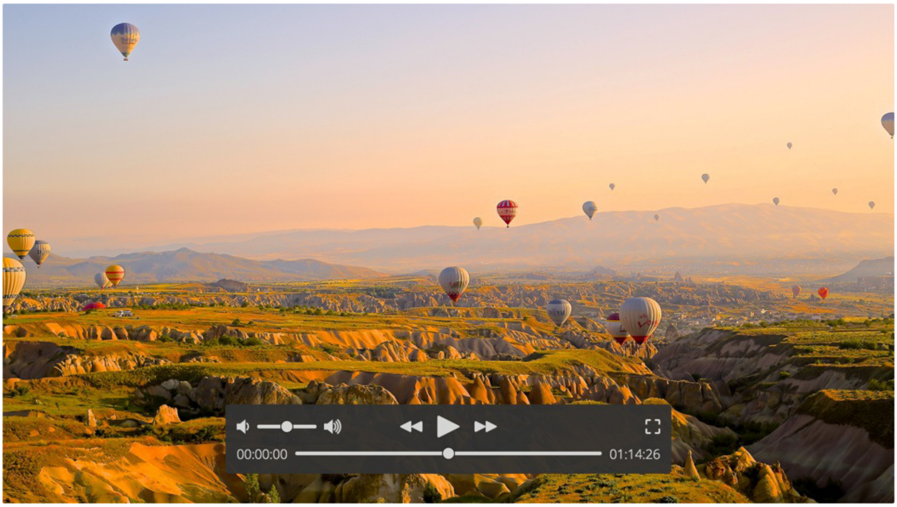
Next, align the items in the size controls container to the end.

```
.size-controls {
  justify-content: flex-end;
}
```



The very last step is to add a small margin around the buttons and time elements.

```
button, .time-elapsed, .time-remaining {
  margin: 0 8px;
}
```



That's it! Two thumbs up!

Fin

The next time you're ready to kick back and watch your favorite action flick, remember you can rebuild the playback controls using your own flexbox kung fu.

CHAPTER 4

Say Goodbye to Vendor Prefixes

Imagine you're lying on a beach. Waves slide up and down a sandy shore while the warm sun beats down on your skin. You sip a cool, refreshing drink, and sigh as gulls faintly caw in the distance.

A gentle breeze lightly brushes your fingers as they slide across your keyboard. Tap tap tap. You're writing CSS. Not just any CSS, but *pure* CSS, the *purest* you can imagine. There are no vendor prefixes or browser inconsistencies, no external libraries and no compilers. Your code just works.

In this chapter, I'll show you how this dream can become reality with a tool called Autoprefixer. I'll also walk you through the problems it solves and how to set it up.

Writing Vanilla Flexbox Sucks

With flexbox, there are two things getting in the way of coding utopia: *old versions of the syntax* and *vendor prefixes*.

Old Versions of Flexbox

The process for adding a new feature like flexbox into the CSS language is complex and lengthy. It may seem like flexbox is fairly new, but the first draft was actually done back in 2009. Since then, flexbox has gone through two major changes, leaving us with three versions of the syntax.

- The 2009 version used `display: box` and had properties that began with `box`.
- The 2012 syntax used `display: flexbox`.
- The current version uses `display: flex` and properties that begin with `flex`. It's *extremely* unlikely the syntax will change again.

These implementations are very similar, but the syntax is different and the older versions don't support all of the newer features. Unfortunately, Android 4.1 and 4.3 only support the 2009 syntax, and IE10 only supports the 2012 syntax, so if you want maximum browser support you still need to use them.

Vendor Prefixes

Have you ever seen CSS properties starting with `-webkit-`, `-moz-`, `-ms-` or `-o-`? Those thing are called *vendor prefixes*. My favorite explanation of vendor prefixes comes from Peter-Paul Koch of QuirksMode:

> *Originally, the point of vendor prefixes was to allow browser makers to start supporting experimental CSS declarations.*
>
> *Let’s say a W3C working group is discussing a grid declaration (which, incidentally, wouldn’t be such a bad*

idea). Let’s furthermore say that some people create a draft specification, but others disagree with some of the details. As we know, this process may take ages.

Let’s furthermore say that Microsoft as an experiment decides to implement the proposed grid. At this point in time, Microsoft cannot be certain that the specification will not change. Therefore, instead of adding grid to its CSS, it adds -ms-grid.

The vendor prefix kind of says “this is the Microsoft interpretation of an ongoing proposal.” Thus, if the final definition of grid is different, Microsoft can add a new CSS property grid without breaking pages that depend on -ms-grid.

Putting Them Together

Let's take an example of some CSS from Chapter 1.

```
.fourth-face {
  display: flex;
  justify-content: space-between;
}
```

```css
.fourth-face .column {
  display: flex;
  flex-direction: column;
  justify-content: space-between;
}
```

When you add the vendor prefixes and old versions of the syntax, it looks like this:

```css
.fourth-face {
  display: -webkit-box;
  display: -webkit-flex;
  display: -ms-flexbox;
  display: flex;
  -webkit-box-pack: justify;
  -webkit-justify-content: space-between;
  -ms-flex-pack: justify;
  justify-content: space-between;
}
```

```
.fourth-face .column {
  display: -webkit-box;
  display: -webkit-flex;
  display: -ms-flexbox;
  display: flex;
  -webkit-box-orient: vertical;
  -webkit-box-direction: normal;
  -webkit-flex-direction: column;
  -ms-flex-direction: column;
  flex-direction: column;
  -webkit-box-pack: justify;
  -webkit-justify-content: space-between;
  -ms-flex-pack: justify;
  justify-content: space-between;
}
```

Yikes! There are a ton of problems with this code:

- **You're repeating yourself.** Generally, developers shouldn't duplicate code when possible.
- **Too much to remember.** It's a lot of extra work to remember all of those vendor prefixes. They don't always match the names of the regular properties.
- **It's easy to forget a prefix.** If you do, you won't notice the problem unless you're specifically testing in that browser.

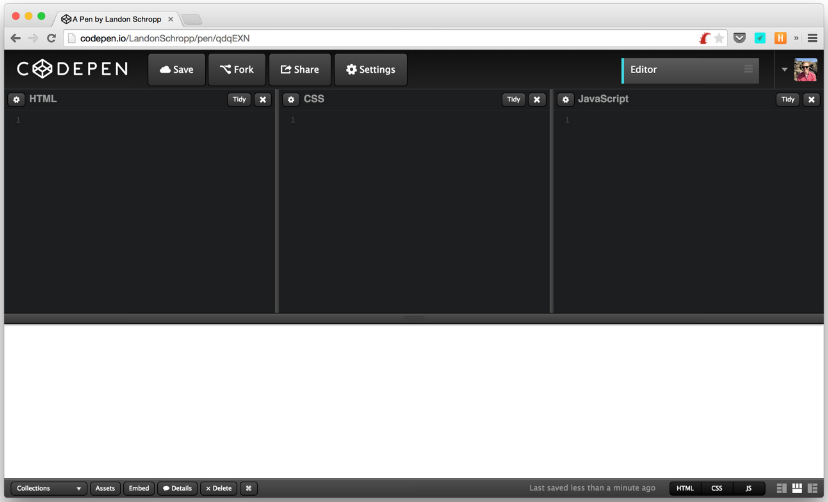
- **The code is difficult to maintain.** When you make a change, you have to do it in several places.

Autoprefixer to the Rescue!

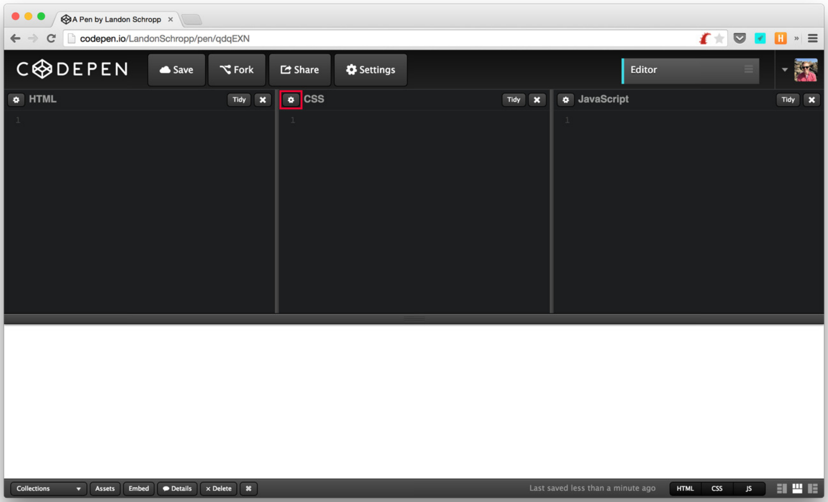
Autoprefixer is a tool that automatically adds vendor prefixes to your CSS. It also translates properties to older versions and even *removes* any unnecessary prefixes. You can configure it to target specific browsers. Best of all, it works like magic—once it's turned on, you can forget it's there.

CodePen

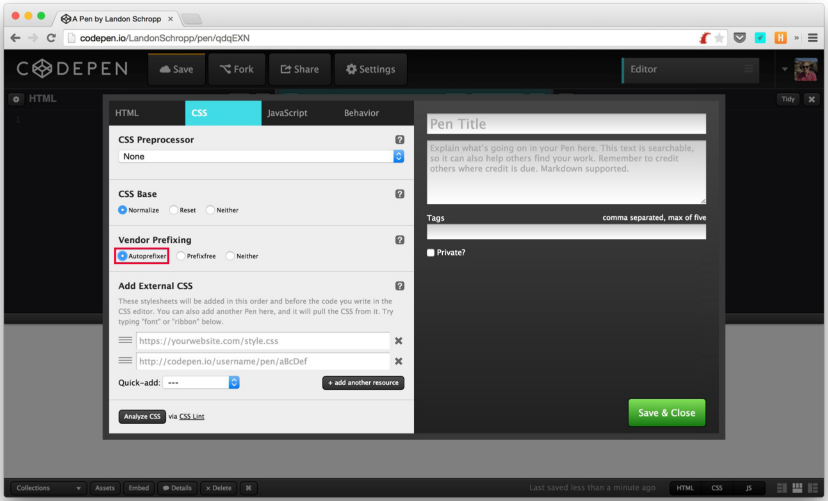
The easiest way to try out AutoPrefixer is with CodePen. CodePen is an online code editor that lets you jump straight into coding. To give it a shot, head over to CodePen and click on the “New Pen” button.



Next, click the settings icon next to CSS.



Enable Autoprefixer by clicking on the radio button next to the “Autoprefixer” label. (I'd also recommend turning on Normalize.)



CSS Preprocessor: None

CSS Base: [✓ Normalize] Reset Neither

Vendor Prefixing: [✓ Autoprefixer] Prefixfree Neither

Add External CSS

These stylesheets will be added in this order and before the code you write in the CSS editor. You can also add another Pen here, and it will pull the CSS from it. Try typing "font" or "ribbon" below.

https://yourwebsite.com/style.css

http://codepen.io/username/pen/aBcDef

Quick-add: ---

Explain what's going on in your Pen here. This text is searchable, so it can also help others find your work. Remember to credit others where credit is due. Markdown supported.

From here, you can type unprefixed CSS into the CSS area and Autoprefixer will do all the work for you!

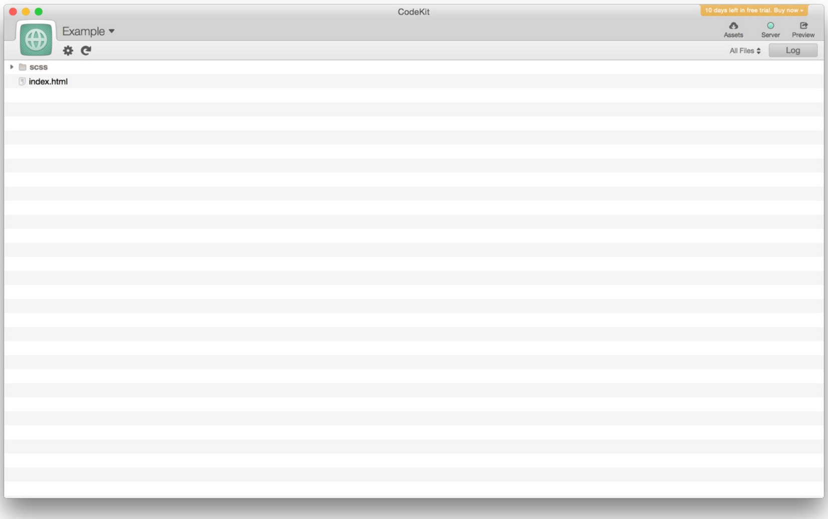
CodeKit

CodePen is great for small projects, but most web development is done locally. Setting up a computer for development can get *extremely* complicated. Fortunately, it's easy with CodeKit.

CodeKit is a program that watches and incorporates a *ton* of popular tools, such as Autoprefixer, Sass and CoffeeScript, into your project. It's $32, but it's worth the money if you're not quite ready for tools like Gulp or Grunt.

Note: Prepros is an alternative to CodeKit for OS X and Windows. The instructions for setting it up should be very similar to the CodeKit instructions.

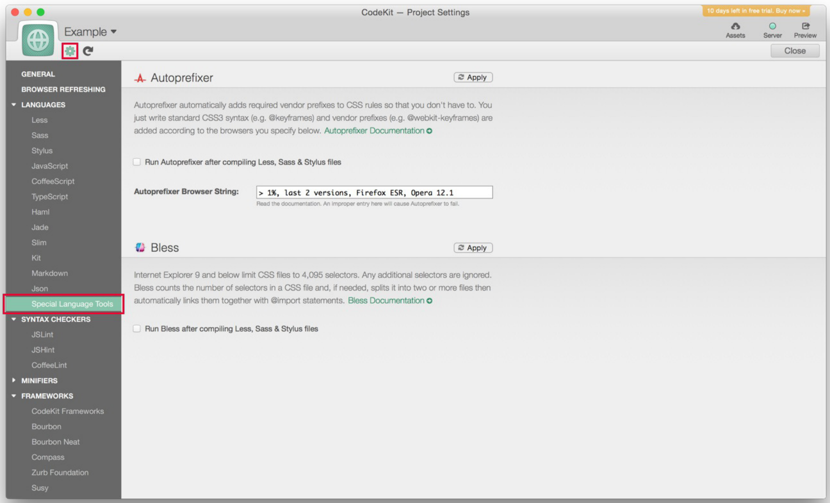
To get started, open up CodeKit and drag a folder into the main window.



Unfortunately, CodeKit doesn't allow you to run Autoprefixer on plain old CSS files. However, you can convert them to Sass files by renaming the extensions to `.scss`. You can still write

CSS in these files like you normally would. CodeKit automatically copies all of your compiled CSS files into a `css` directory, so I'd recommend keeping your source CSS files in an `scss` folder.

Click on the settings button and then under “Languages” click on “Special Language Tools.”



Autoprefixer

Autoprefixer automatically adds required vendor prefixes to CSS rules so that you don't have to. You just write standard CSS3 syntax (e.g. @keyframes) and vendor prefixes (e.g. @webkit-keyframes) are added according to the browsers you specify below. Autoprefixer Documentation

☐ Run Autoprefixer after compiling Less, Sass & Stylus files

Autoprefixer Browser String: `> 1%, last 2 versions, Firefox ESR, Opera 12.1`

Read the documentation. An improper entry here will cause Autoprefixer to fail.

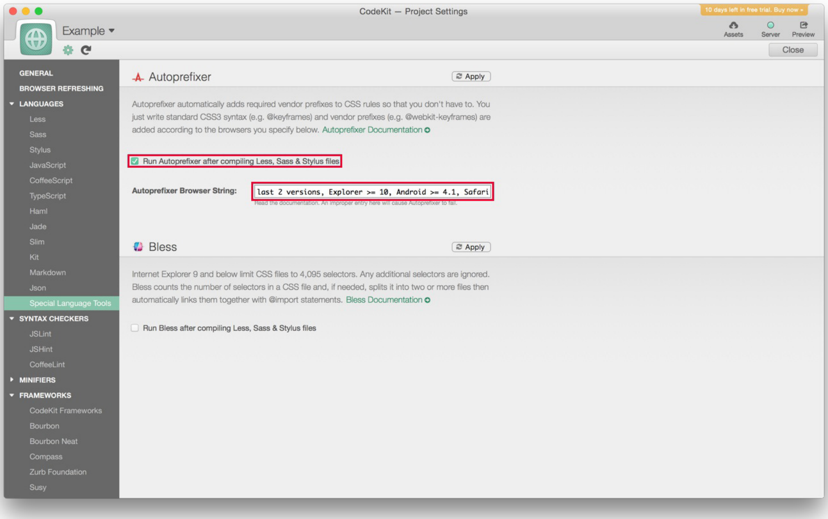
Bless

Internet Explorer 9 and below limit CSS files to 4,095 selectors. Any additional selectors are ignored. Bless counts the number of selectors in a CSS file and, if needed, splits it into two or more files then automatically links them together with @import statements. Bless Documentation

☐ Run Bless after compiling Less, Sass & Stylus files

On this page, check the box next to “Run Autoprefixer.” In order to support all of the browsers that can display flexbox, change the “Autoprefixer Browser String” to this:

```
last 2 versions, Explorer >= 10,
Android >= 4.1, Safari >= 7, iOS >= 7
```



Autoprefixer

Autoprefixer automatically adds required vendor prefixes to CSS rules so that you don't have to. You just write standard CSS3 syntax (e.g. @keyframes) and vendor prefixes (e.g. @webkit-keyframes) are added according to the browsers you specify below. Autoprefixer Documentation

[✓] Run Autoprefixer after compiling Less, Sass & Stylus files

Autoprefixer Browser String: `last 2 versions, Explorer >= 10, Android >= 4.1, Safari`

Read the documentation. An improper entry here will cause Autoprefixer to fail.

Bless

Internet Explorer 9 and below limit CSS files to 4,095 selectors. Any additional selectors are ignored. Bless counts the number of selectors in a CSS file and, if needed, splits it into two or more files then automatically links them together with @import statements. Bless Documentation

[ ] Run Bless after compiling Less, Sass & Stylus files

That's it! Now, every time you make a change to one of your files, CodeKit will automatically compile it using Autoprefixer.

Gulp, Grunt and Other Frameworks

Gulp and Grunt are tools that are make it easy to set up and build your projects. With them, you can set up powerful build systems that compile your files, run static analysis on your code and even host local servers.

Configuring these tools is a little too in depth for this lesson. However, if you're curious, I'd highly recommend digging into one of them. The gulp-autoprefixer and grunt-autoprefixer packages both include examples in their readme files. If you'd like to see my personal gulpfile, check out this Gist.

Many web application frameworks support Autoprefixer, including Ruby on Rails, ASP.NET, Express and CakePHP. If you're building simple sites, you can use static site generators like Middleman or Jekyll.

Chances are good there's a way to incorporate Autoprefixer into your favorite framework!

What If You Can't Use Autoprefixer?

If you can't use Autoprefixer in your project, you have a couple options.

The first is to use a library called -prefix-free. This library does the same thing as Autoprefixer, but in the browser. The downside is that it takes extra time and processing power to prefix the CSS, which can make your site feel a little slow, especially on mobile browsers.

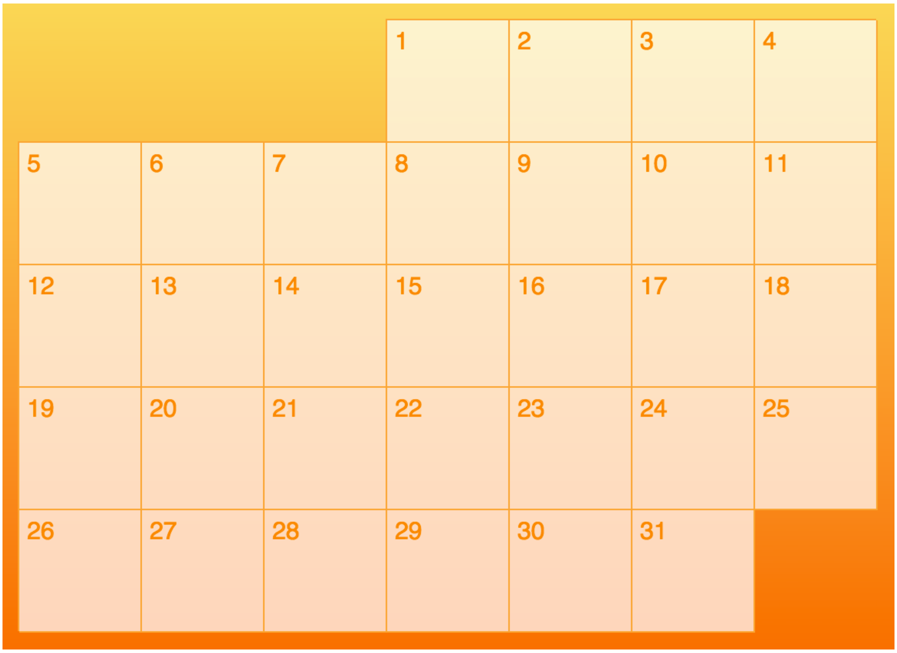
The other option is a tool called Pleeease. Pleeease lets you paste in CSS. It then uses Autoprefixer to print out prefixed styles you can copy into your stylesheets.

Don't Worry, Be Happy

That's it! You've learned how to set up your environment for full CSS awesomeness using tools like including CodePen, CodeKit, Gulp and Grunt. You never have to worry about vendor prefixes again!

CHAPTER 5

Breaking Free from
Twelve-Column Layouts



You're a rebel without a cause. You live life on the edge. When others go right, you veer left. You pave your own way, blaze your own trail and march to the beat of your own drum. You have your cake and eat it too.

So it shouldn't come as a surprise that the twelve-column layouts we covered in Chapter 2 aren't enough for you. Others might be content, but not you. You need *more*. You need to craft layouts without constraints. You need *freedom*.

Well, in this chapter you're going to get it. I'll show you how you can use flexbox to build layouts that were once impossible. When you're done, you'll be able to smash through the boundaries of twelve columns layouts. You'll also learn to tip everything on its side and build the same layouts vertically.

N-Column Layouts

Twelve-column layouts are great, but there are some things you can't do with them. Take a calendar for example. Since weeks in calendars tend to be broken up into seven columns, no matter how you arrange it your layout isn't going to center in a twelve-column layout.

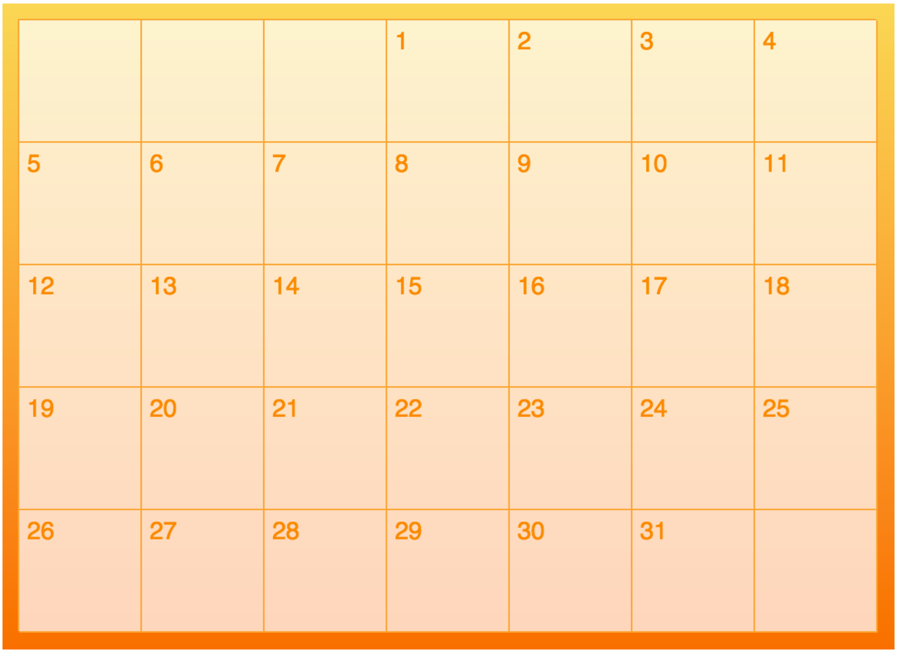
With flexbox, the calendar layout is a piece of cake! All you need to do is set the `flex-grow` and `flex-shrink` properties of each day to `1`.

```
<div class="month">
  <div class="week">
    <div class="day"></div>
    <div class="day"></div>
    <div class="day"></div>
    <div class="day">1</div>
    <div class="day">2</div>
    <div class="day">3</div>
    <div class="day">4</div>
  </div>
```

```html
<div class="week">
  <div class="day">5</div>
  <div class="day">6</div>
  <div class="day">7</div>
  <div class="day">8</div>
  <div class="day">9</div>
  <div class="day">10</div>
  <div class="day">11</div>
</div>
<div class="week">
  <div class="day">12</div>
  <div class="day">13</div>
  <div class="day">14</div>
  <div class="day">15</div>
  <div class="day">16</div>
  <div class="day">17</div>
  <div class="day">18</div>
</div>
<div class="week">
  <div class="day">19</div>
  <div class="day">20</div>
  <div class="day">21</div>
  <div class="day">22</div>
  <div class="day">23</div>
  <div class="day">24</div>
  <div class="day">25</div>
</div>
```

```
  <div class="week">
    <div class="day">26</div>
    <div class="day">27</div>
    <div class="day">28</div>
    <div class="day">29</div>
    <div class="day">30</div>
    <div class="day">31</div>
    <div class="day"></div>
  </div>
</div>
```

```
.week {
  display: flex;
}

.day {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0;
}
```

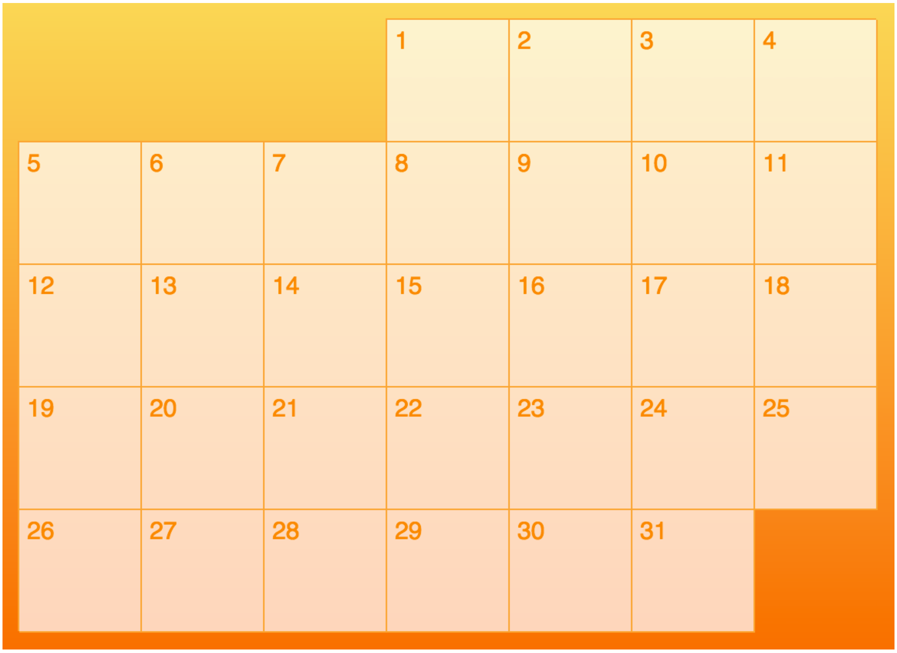


			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Wouldn't it be nice if you didn't have to include empty days? With flexbox, that's easy!

For the first day, set the left margin to `100%` × $^{3}/_{7}$, which is `42.857%`. Do the same on the last day by setting the right margin to `100%` × $^{1}/_{7}$, or `14.285%`. Finally, remove the empty days from the HTML.

```
.week:first-of-type .day:first-of-type {
  margin-left: 42.857%;
}

.week:last-of-type .day:last-of-type {
  margin-right: 14.285%;
}
```



The Golden Ratio

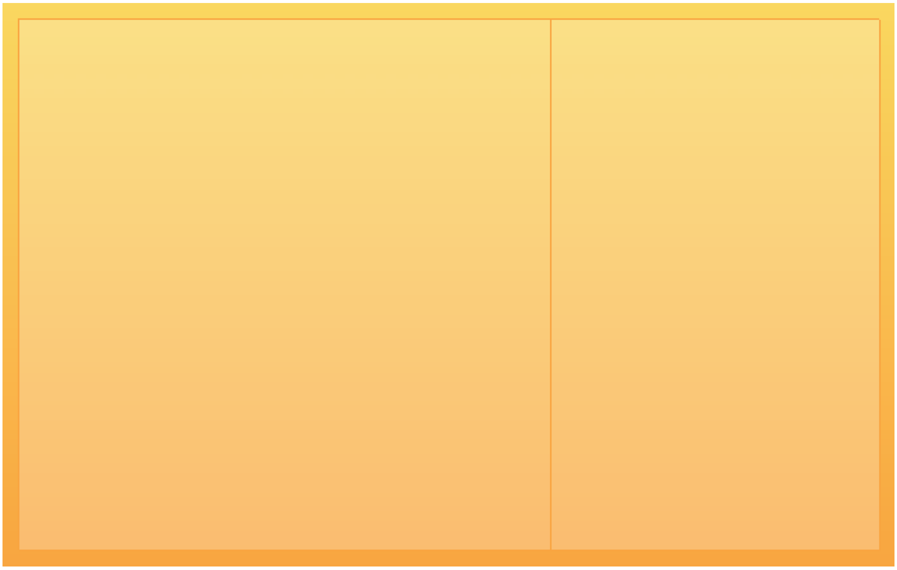
Sometimes, you want to build layouts that aren't divisible by whole numbers. A good example is layouts based on the golden ratio, a popular constant found in art, mathematics, architecture, biology and numerous other places.

Flexbox can handle golden ratio layouts with ease. Simply set `flex-grow` and `flex-shrink` to `1` for one of the columns and `1.61803` (the golden ratio) for the other.

```
<section>
  <div class="phi-column"></div>
  <div class="column"></div>
</section>
```

```
section {
  width: 560px;
  height: 346px;
  display: flex;
}

.phi-column {
  flex-grow: 1.61803;
  flex-shrink: 1.61803;
  flex-basis: 0;
}
```

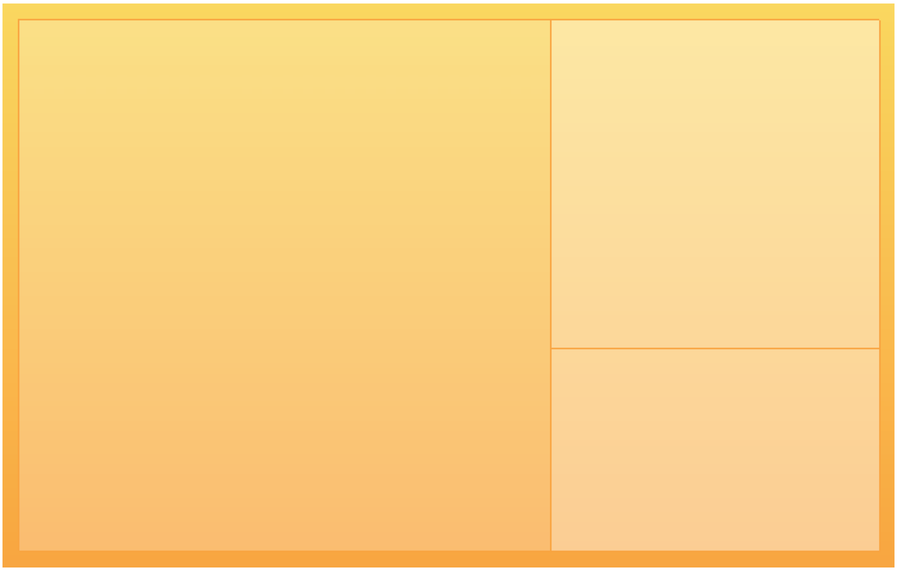
```
.column {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0;
}
```



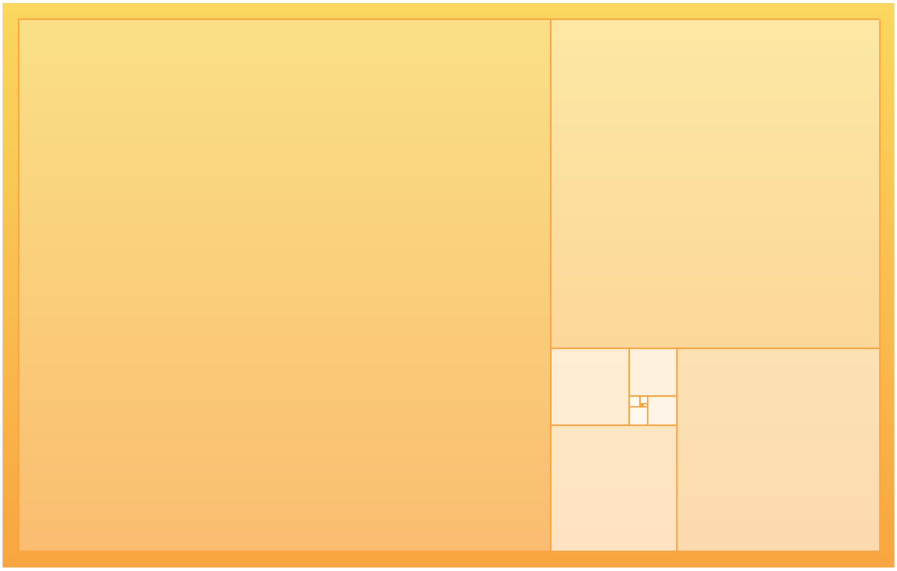
Because flexbox allows flex containers to be nested, it's easy to add containers for rows as well.

```
<section>
  <div class="phi-column"></div>
  <div class="column">
    <div class="phi-row"></div>
    <div class="row"></div>
  </div>
</section>
```

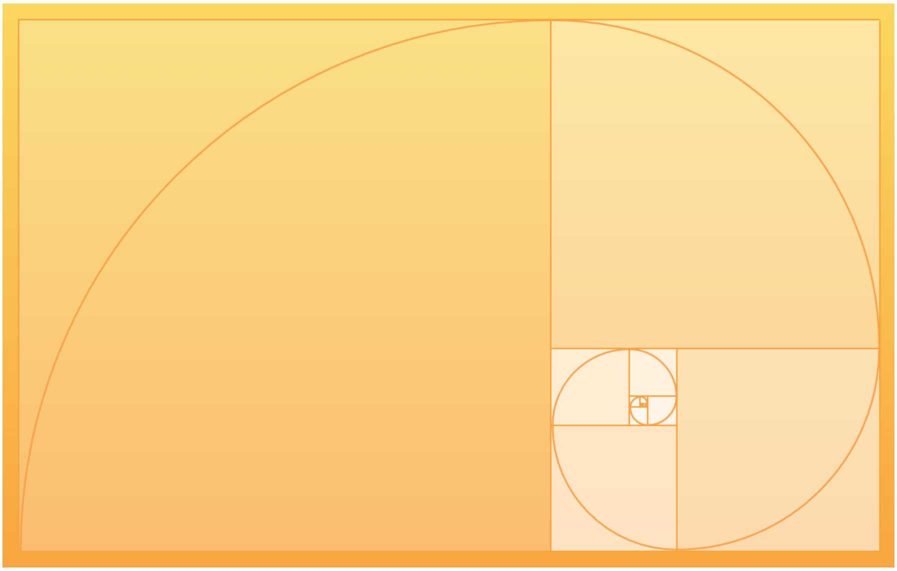
```
.phi-column, .phi-row {
  flex-grow: 1.61803;
  flex-shrink: 1.61803;
  flex-basis: 0;
}

.column, .row {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0;
  display: flex;
}

.column {
  flex-direction: column;
}
```



By continuing this pattern of nesting rows and columns, you'll end up with a golden rectangle.



If you overlay this rectangle with the golden spiral, you'll really see the golden ratio in action!



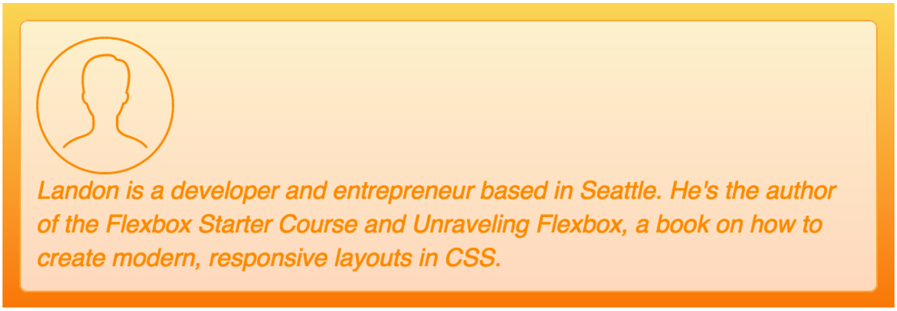
Fixed-Sized Items

Let's say you want to build a short bio for yourself, but you don't know the width of the container.

When the page resizes, ideally the image will stay the same width and the text will fill up the rest of the space. Can you do it with flexbox? Absolutely!

Let's start with the markup.

```
<section class="bio">
  <figure>
    <img width="90" height="90"
      src="profile.svg" alt="Profile">
  </figure>
  <p>
    Landon is a developer and entrepreneur
    based in Seattle. He's the author of the
    Flexbox Starter Course and Unraveling
    Flexbox, a book on how to create modern,
    responsive layouts in CSS.
  </p>
</section>
```


Landon is a developer and entrepreneur based in Seattle. He's the author of the Flexbox Starter Course and Unraveling Flexbox, a book on how to create modern, responsive layouts in CSS.

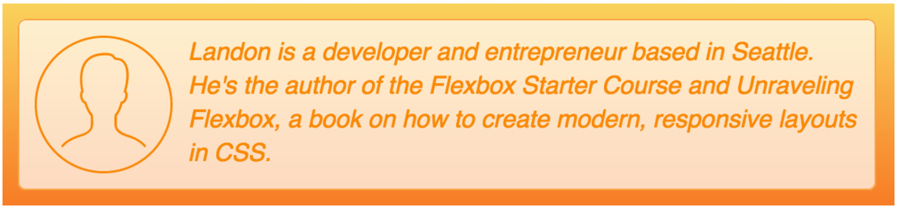
Remember the `flex-basis` property? It determines how much space a flex item takes up *before* the remaining space is distributed using `flex-grow` and `flex-shrink`. You can accomplish your goal by setting `flex-basis` to the size you want and setting `flex-grow` and `flex-shrink` to `0`.

```
section {
  display: flex;
}

p {
  flex: 1 1 0px;
  margin-left: 10px;
}

figure {
  flex: 0 0 90px;
}

p {
  flex: 1 1 0px;
}
```

Landon is a developer and entrepreneur based in Seattle. He's the author of the Flexbox Starter Course and Unraveling Flexbox, a book on how to create modern, responsive layouts in CSS.

What if you don't know the size of the image ahead of time? No problem! Use the `auto` value for the `flex-basis` property to tell flexbox to size the figure to the size of its content.

```
figure {
  flex: 0 0 auto;
  padding-right: 10px;
}
```



Landon is a developer and entrepreneur based in Seattle. He's the author of the Flexbox Starter Course and Unraveling Flexbox, a book on how to create modern, responsive layouts in CSS.

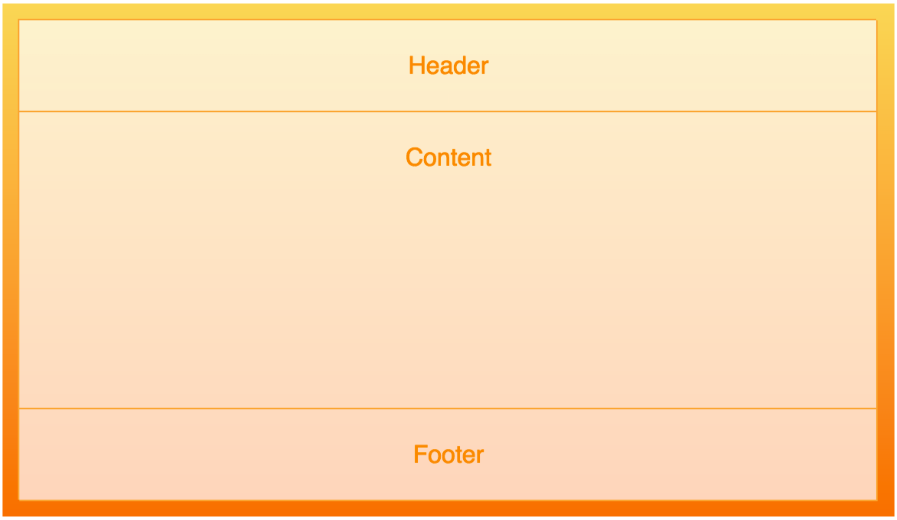
Vertical Layouts

In the past, vertical layouts weren't really feasible with CSS. Flexbox changes all of that. *Everything you've learned so far about flexbox can be done with vertical layouts.*

Let's say you wanted to build a full page layout with a fixed header and footer, and content in the middle. That's easy! You can apply the same trick you used in the bio layout to keep the header and footer the same size. Then, you'll set `flex-grow` and `flex-shrink` to `1` for the content.

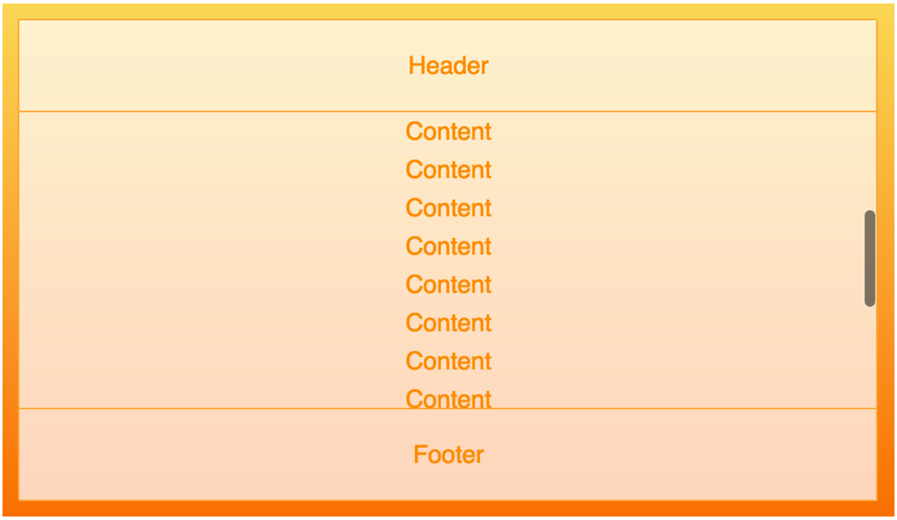
```html
<main>
  <header>Header</header>
  <section>Content</section>
  <footer>Footer</footer>
</main>
```

```css
html, body {
  height: 100%;
}

main {
  display: flex;
  flex-direction: column;
  height: 100%;
}

header, footer {
  flex: 0 0 100px;
}

section {
  flex: 1 1 0px;
}
```



What if the content in the `<section>` is larger than the space on the screen? Let's make it scroll!

```
section {
  flex: 1 1 0px;
  overflow: auto;
  -webkit-overflow-scrolling: touch;
}
```

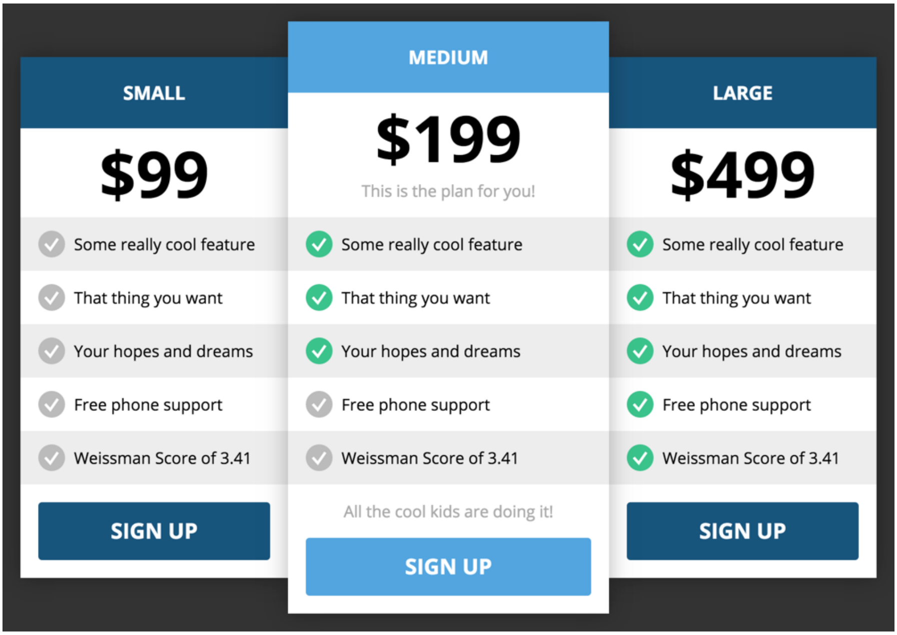


All Done

In this chapter, you learned that flexbox can do much more than twelve-column layouts. You built items that stay the same size while their siblings grow and shrink, and you've even done vertical layouts. The next time you need to build a tricky layout, don't hesitate to look for a flexbox solution!

CHAPTER 6

Perfect Pricing



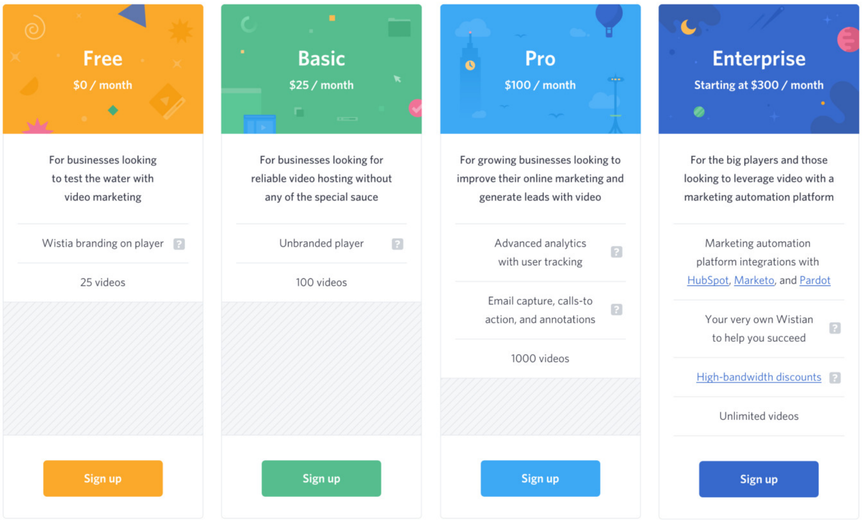
SMALL	MEDIUM	LARGE
$99	$199 This is the plan for you!	$499
✓ Some really cool feature	✅ Some really cool feature	✅ Some really cool feature
✓ That thing you want	✅ That thing you want	✅ That thing you want
✓ Your hopes and dreams	✅ Your hopes and dreams	✅ Your hopes and dreams
✓ Free phone support	✓ Free phone support	✅ Free phone support
✓ Weissman Score of 3.41	✓ Weissman Score of 3.41	✅ Weissman Score of 3.41
SIGN UP	All the cool kids are doing it! SIGN UP	SIGN UP

Cha-ching! If there's one place on a website that matters more than any other, it's the place where people spend their hard-earned cash. It's where someone goes from being interested in a product to trusting a company with their money.

Pricing layouts tend to be a combination of a few key elements:

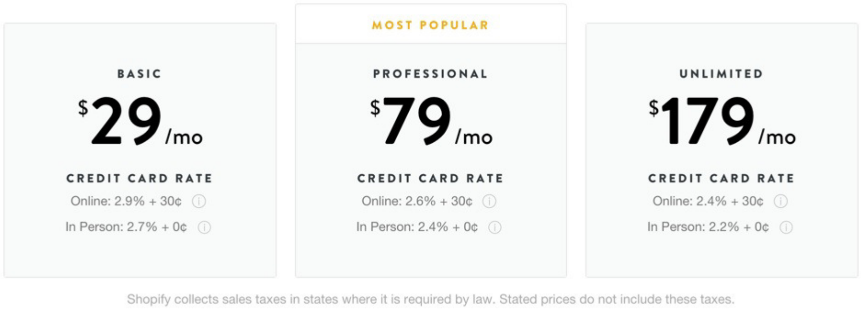
- Three or four plans, arranged in columns
- The price in big, bold letters
- A list of features and benefits
- Call-to-action buttons

Take a look at these examples from Wistia, Shopify and Slack.



Free	Basic	Pro	Enterprise
$0 / month	$25 / month	$100 / month	Starting at $300 / month
For businesses looking to test the water with video marketing	For businesses looking for reliable video hosting without any of the special sauce	For growing businesses looking to improve their online marketing and generate leads with video	For the big players and those looking to leverage video with a marketing automation platform
Wistia branding on player	Unbranded player	Advanced analytics with user tracking	Marketing automation platform integrations with HubSpot, Marketo, and Pardot
25 videos	100 videos	Email capture, calls-to action, and annotations	Your very own Wistian to help you succeed
		1000 videos	High-bandwidth discounts
			Unlimited videos
Sign up	Sign up	Sign up	Sign up

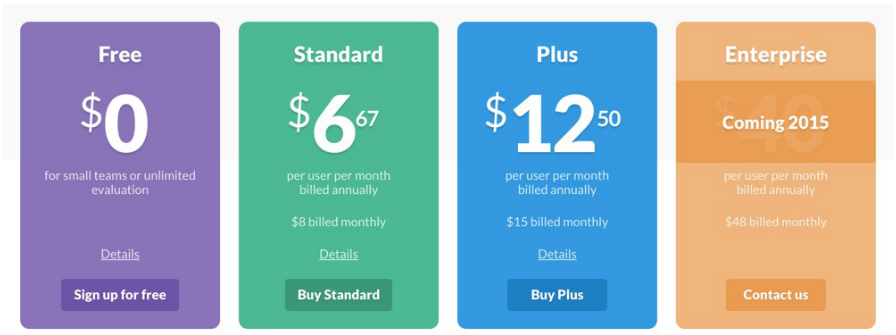
Wistia's pricing page



Basic	Professional (Most Popular)	Unlimited
$29 /mo	$79 /mo	$179 /mo
Credit card rate	Credit card rate	Credit card rate
Online: 2.9% + 30¢	Online: 2.6% + 30¢	Online: 2.4% + 30¢
In Person: 2.7% + 0¢	In Person: 2.4% + 0¢	In Person: 2.2% + 0¢

Shopify collects sales taxes in states where it is required by law. Stated prices do not include these taxes.

Shopify's pricing page



Slack's pricing page

In this chapter, I'll show you how to build a killer pricing page using flexbox. If you develop marketing sites for business, you'll use what you learn from the chapter *all the time*. Let's jump in.

The Columns

The first step is, as always, the markup.

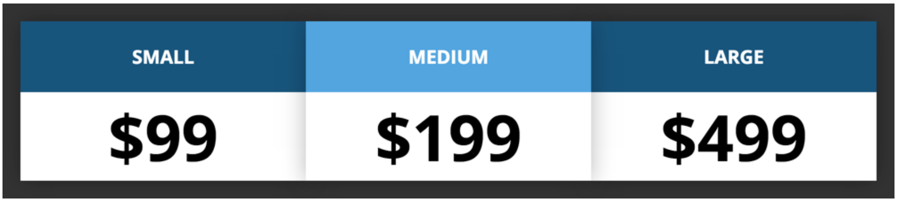
```
<section class="pricing">
  <div class="plan">
    <h3>Small</h3>
    <div class="price">$99</div>
  </div>
  <div class="plan">
    <h3>Medium</h3>
    <div class="price">$199</div>
  </div>
  <div class="plan">
    <h3>Large</h3>
    <div class="price">$499</div>
  </div>
</section>
```

Note: Like the other examples, I've included a few basic styles to speed things up. Don't forget you can always check the source code for the examples to see all of the styles!

Here, you have added a pricing `<section>` with three plans contained inside of it. Each plan has a name and a price.

The next step is to make the pricing `<section>` a flex container and turn the plans into columns.

```
.pricing {
  display: flex;
  width: 960px;
}

.plan {
  flex: 1 1 0px;
}
```

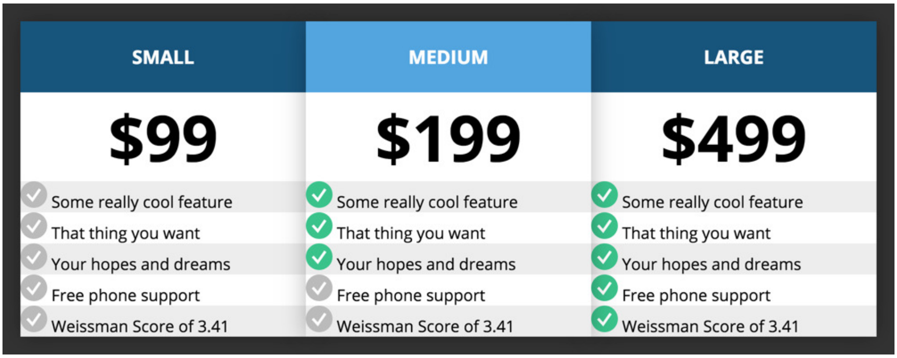


Notice how often you're using the column layouts you learned in Chapter 2? That's because they're *everywhere*! You can also see that I've added a few default styles for you.

Features

Next up is the list of features in each plan. Here's the HTML for the middle plan. The first and third plans are the same, except images are toggled differently.

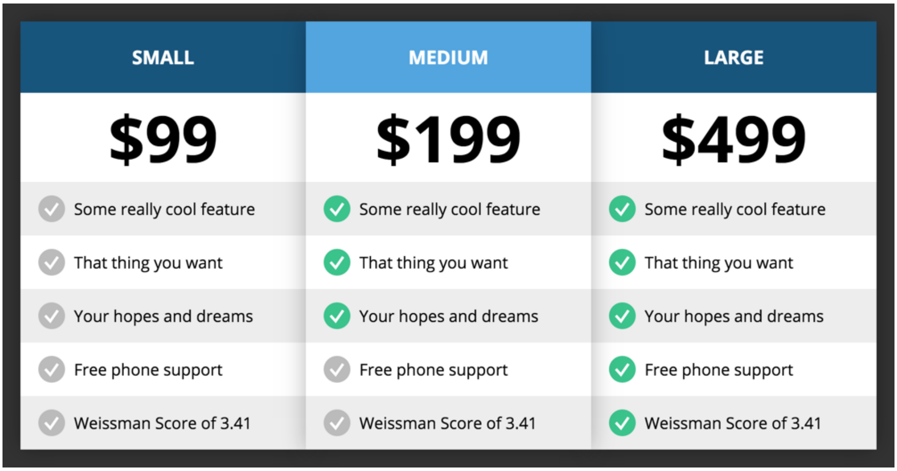
```html
<div class="plan">
  <h3>Medium</h3>
  <div class="price">$199</div>

  <ul>
    <li>
      <img src="green_check.svg"
        alt="Included">
      Some really cool feature
    </li>
    <li>
      <img src="green_check.svg"
        alt="Included">
    That thing you want
    </li>
    <li>
      <img src="green_check.svg"
        alt="Included">
      Your hopes and dreams
    </li>
    <li>
      <img src="gray_check.svg"
        alt="Not Included">
      Free phone support
    </li>
    <li>
      <img src="gray_check.svg"
        alt="Not Included">
      Weissman Score of 3.41
    </li>
  </ul>
</div>
```



SMALL	MEDIUM	LARGE
$99	$199	$499
Some really cool feature	Some really cool feature	Some really cool feature
That thing you want	That thing you want	That thing you want
Your hopes and dreams	Your hopes and dreams	Your hopes and dreams
Free phone support	Free phone support	Free phone support
Weissman Score of 3.41	Weissman Score of 3.41	Weissman Score of 3.41

Next, style the list items by making the `<li>` a flex container and setting the margin, padding and line height.

```
li {
  display: flex;
  line-height: 30px;
  padding: 15px 20px;
}

li img {
  margin-right: 10px;
}
```



SMALL	MEDIUM	LARGE
$99	$199	$499
Some really cool feature	Some really cool feature	Some really cool feature
That thing you want	That thing you want	That thing you want
Your hopes and dreams	Your hopes and dreams	Your hopes and dreams
Free phone support	Free phone support	Free phone support
Weissman Score of 3.41	Weissman Score of 3.41	Weissman Score of 3.41

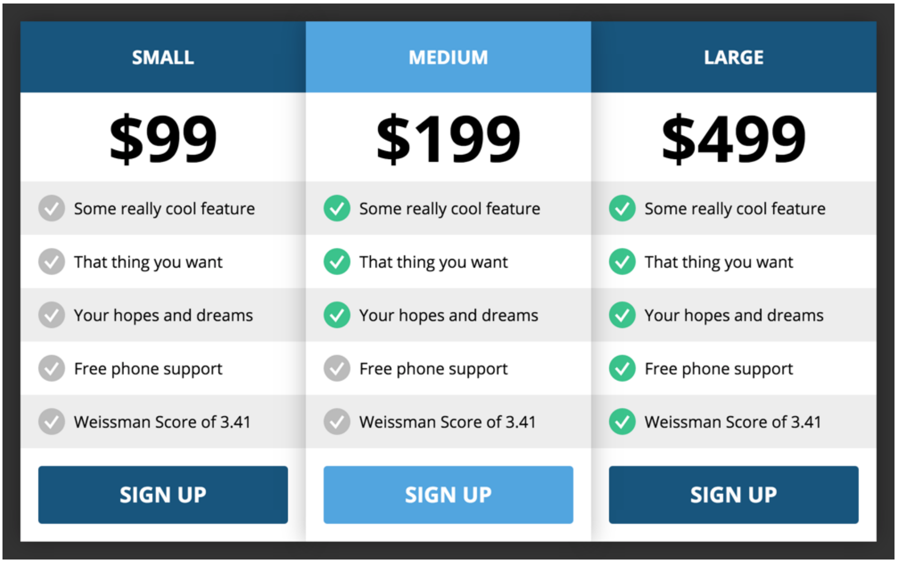
Call to Action Buttons

No pricing form is complete without a call to action. Let's add a button to the bottom of each plan so a person can select it.

```
<section class="pricing">
  <div class="plan">
    ...
    <button>Sign Up</button>
  </div>
  <div class="plan">
    ...
    <button>Sign Up</button>
  </div>
  <div class="plan">
    ...
    <button>Sign Up</button>
  </div>
</section>
```

In this case, we want to treat the button as a block element that takes up the full width of its container. Since the button has a margin of `20px`, we'll use `calc` in the `width` to get it the right size.

```
button {
  display: block;
  width: calc(100% - 40px);
  margin: 20px;
}
```



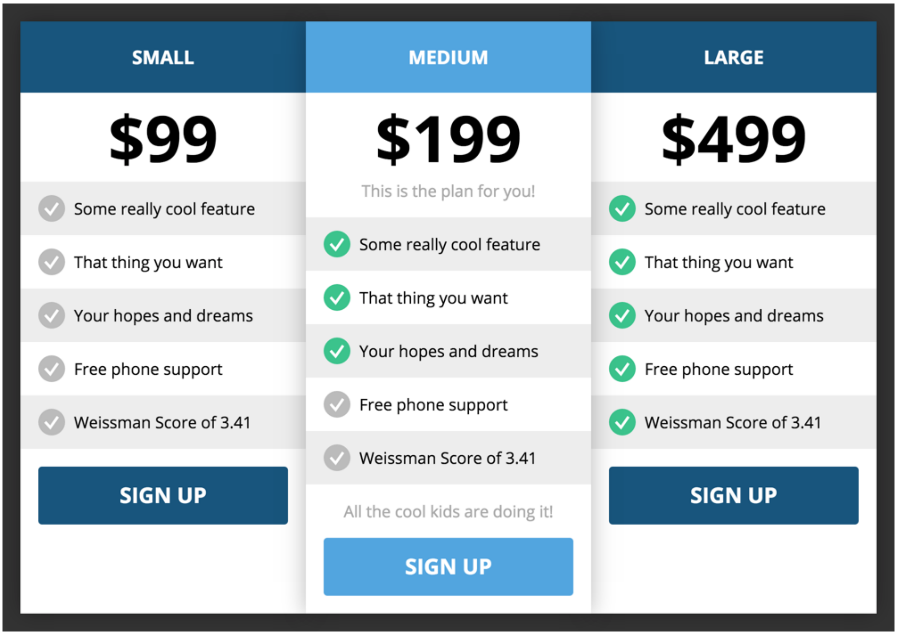
SMALL	MEDIUM	LARGE
$99	$199	$499
✓ Some really cool feature	✅ Some really cool feature	✅ Some really cool feature
✓ That thing you want	✅ That thing you want	✅ That thing you want
✓ Your hopes and dreams	✅ Your hopes and dreams	✅ Your hopes and dreams
✓ Free phone support	✓ Free phone support	✅ Free phone support
✓ Weissman Score of 3.41	✓ Weissman Score of 3.41	✅ Weissman Score of 3.41
SIGN UP	SIGN UP	SIGN UP

Making It Stand out

As it is, the pricing layout looks really good. However, there's one more thing that could make it better: often, pricing pages will highlight the most profitable plan on their page, driving more people to click it. In our page, let's emphasize the medium plan by making it stick out above the other two.

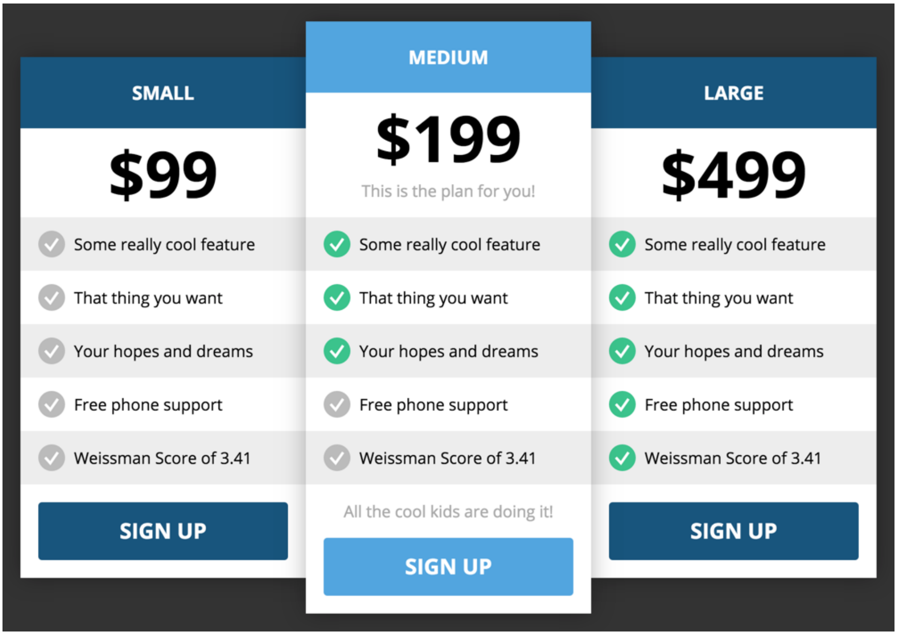
First, add some HTML to the middle column to make it a little taller.

```
<div class="plan">
  <h3>Medium</h3>
  <div class="price">$199</div>
  <p class="info">
    This is the plan for you!
  </p>
  ...
  <p class="info">
    All the cool kids are doing it!
  </p>
  <button>Sign Up</button>
</div>
```



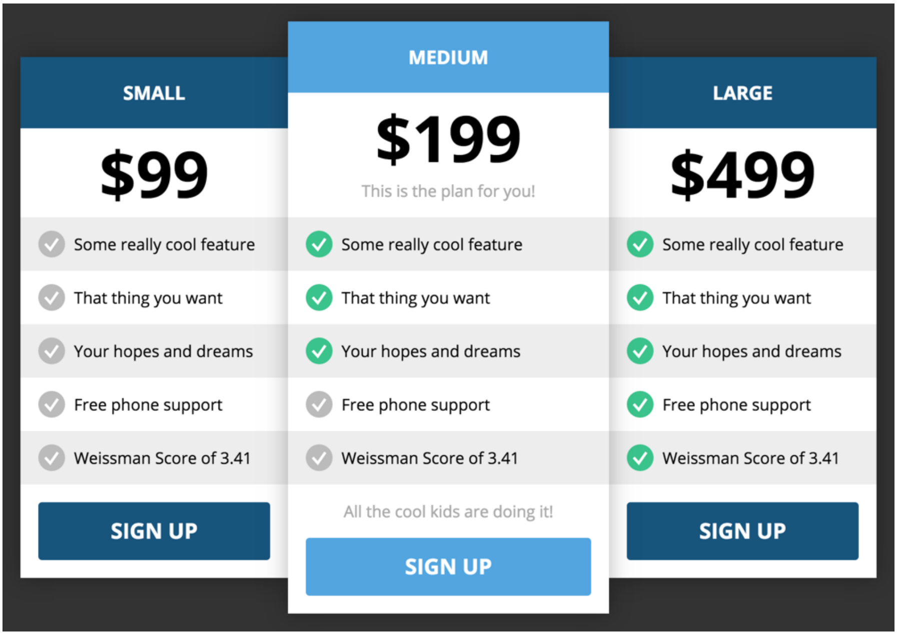
Next, add a top and bottom margin to the first and third plans.

```
.plan:nth-of-type(1), .plan:nth-of-type(3) {
  margin-top: 40px;
  margin-bottom: 40px;
}
```



Now that the middle plan is taller than the other two, it looks a little funny at the same width. You can fix that by overriding the `flex-basis` property to make it a little wider.

```
.plan:nth-of-type(2) {
  flex-basis: 60px;
}
```



Money in the Bank

You now know how to make a killer pricing layout that's sure to convert. You've seen how the previous techniques you learned can be rehashed into something completely different. In the next chapter, you'll take an even more radical departure into flexbox forms!

CHAPTER 7

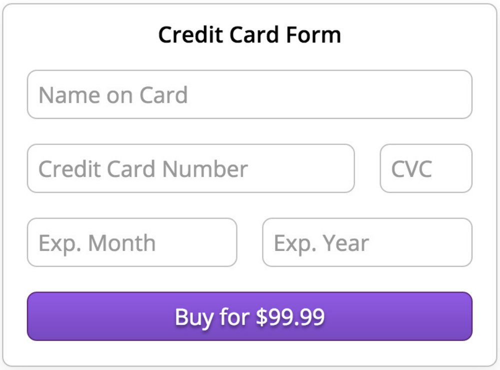
Flexbox Forms

It's undeniable: flexbox is great for full-page layouts. But did you know that's not all you can do with it? Unlike twelve-column grid systems, flexbox is perfect for building small layouts as well. Flexbox works well with these layouts because it's easy to make them fill the space containing them, so you can reuse them almost anywhere!

In this chapter, I'll show you how to build three form layouts that fit this pattern. You'll learn how to create a credit card form, multi-buttons, and attached buttons, all with flexbox!

Credit Card Form

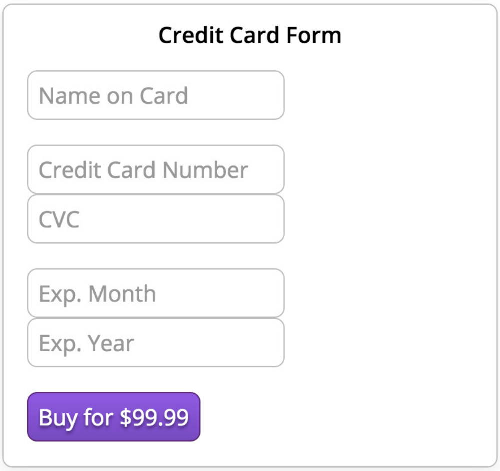
If you spend enough time building web applications, you'll eventually need to code a credit card form. Flexbox makes that easy!



Let's start with the markup.

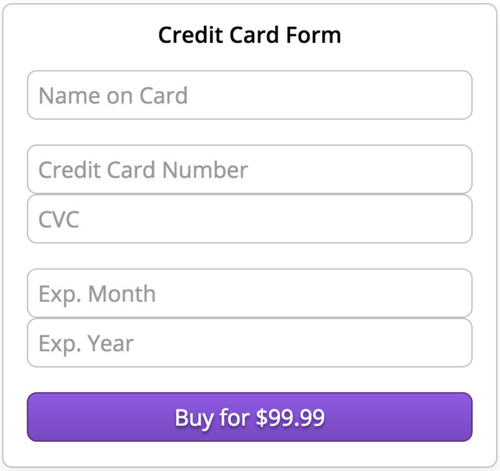
Note: In this example, we're using `<div class="fieldset">` *instead of* `<fieldset>`*. Why? There's a bug in Chrome, Safari and Firefox that prevents* `<fieldset>` *elements from being flex containers. Boo! Oh well, at least it's easy to work around.*

```
<form>
  <h2>Credit Card Form</h2>

  <div class="fieldset">
    <input id="name" type="text"
      placeholder="Name on Card">
  </div>

  <div class="fieldset">
    <div class="credit-card-number-container">
      <input id="credit-card-number"
        type="number"
        placeholder="Credit Card Number">
    </div>
    <div>
      <input id="cvc" type="number"
        placeholder="CVC">
    </div>
  </div>
  <div class="fieldset">
    <div>
      <input type="text"
        placeholder="Exp. Month">
    </div>
    <div>
      <input type="text"
        placeholder="Exp. Year">
    </div>
  </div>

  <button type="submit">
    Buy for $99.99
  </button>
</form>
```



Step one is to set `display` for the inputs and buttons to `block` to make them fill up the full width of their containers. I like to do this as a default in almost all of my projects. It makes these elements *much* easier to work with.

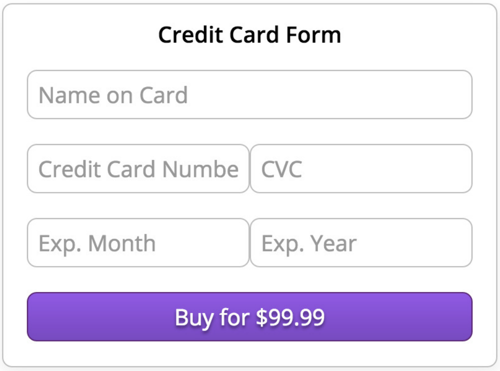
```
label, input, button {
  display: block;
  width: 100%;
}
```



The next step is to make the inputs inside the fieldsets share a single line. You can accomplish this with the `flex` property.

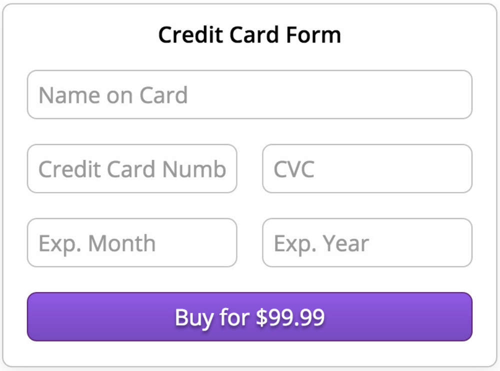
```
.fieldset {
  display: flex;
}

.fieldset > * {
  flex: 1 1 0px;
}
```

This sets `flex-grow` and `flex-shrink` properties for the children of the fieldsets to `1`, which distributes the space evenly.



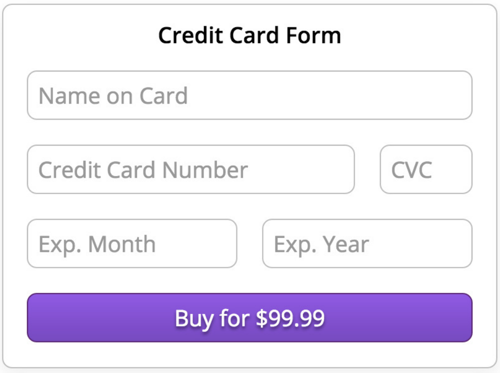
The next step is to add some space between the inputs. The trick is to set the left and right margins of the fieldset's children, and then to remove them for the first and last children.

```
.fieldset > * {
  margin-left: 10px;
  margin-right: 10px;
}

.fieldset > :first-child {
  margin-left: 0;
}

.fieldset > :last-child {
  margin-right: 0;
}
```



The credit card number input is a little trickier. We want it to be three times the width of the CVC container, which seems easy. However, think back to Chapter 2. Remember, you have to set the `flex-basis` of the credit card number container to compensate for the space between the inputs.

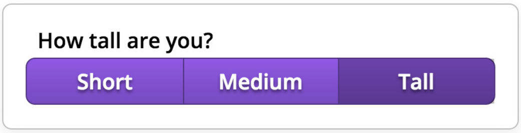
```
.credit-card-number-container {
  flex: 3 3 40px;
}
```



You're open for business!

Multi-Buttons

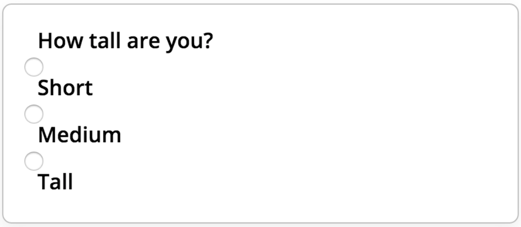
One of my favorite form controls is the multi-button. Unlike a normal button, it's split into parts. When you click on a part, it stays selected. These controls are nice for letting people select from a short list of simple options.



Your first instinct might be to build your multi-button using `<button>` elements, and use JavaScript to handle the selection. However, you can do it without JavaScript—there's another control that already has the behavior you need. What is it? A radio button!

Radio buttons? What? Well, if you select one, the browser deselects the others. They work well with form submissions and they're accessible out of the box.

```
<form>
  <fieldset>
    <label>
      How tall are you?
    </label>
    <div class="multi-button">
      <input type="radio" name="height"
        id="short" value="short">
      <label for="short">Short</label>
      <input type="radio" name="height"
        id="medium" value="medium">
      <label for="medium">Medium</label>
      <input type="radio" name="height"
        id="tall" value="tall">
      <label for="tall">Tall</label>
    </div>
  </fieldset>
</form>
```


How tall are you?

Short

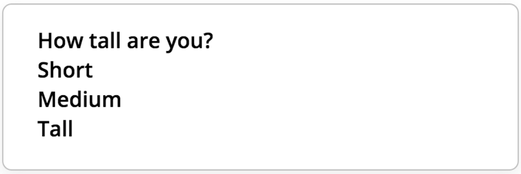
Medium

Tall

So how do you make radio buttons look like a multi-button? The trick is to use *labels*. One of the coolest things about labels is if you click on them, it selects their corresponding inputs. You can take advantage of this by making the label the "button" part of the multi-button.

The first step is to hide those inputs.

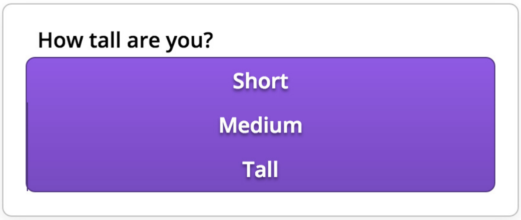
```
input {
  display: none;
}
```


How tall are you?
Short
Medium
Tall

Next, make the multi-button container and labels look like a button.

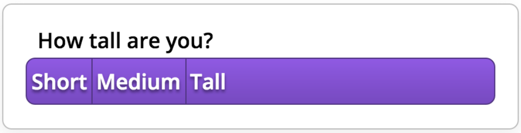
```
.multi-button {
  border: 1px solid #513681;
  background: linear-gradient(to bottom,
    #905ae3, #784bc0);
  color: white;
  text-shadow: 0 2px 2px rgba(0, 0, 0, 0.5);
  border-radius: 8px;
}

.multi-button label {
  text-align: center;
  margin-bottom: 0;
  line-height: 30px;
  padding: 4px;
  user-select: none;
}
```

```
.multi-button label:not(:first-of-type) {
  box-shadow: inset 1px 0 #513681;
}
```



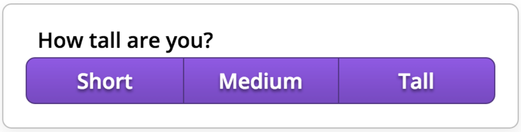
Move the labels onto the same line by making `<div class="multi-button">` a flex container.

```
.multi-button {
  ...
  display: flex;
}
```



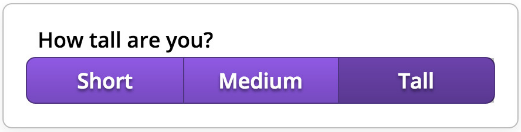
It's starting to look like a multi-button! What we want is for each label to be the same width. You can accomplish this with the `flex` property.

```
.multi-button label {
  ...
  flex: 1 1 0px;
}
```



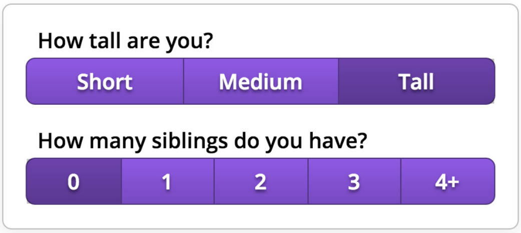
Finally, in order to to show which button is selected, style the checked input's label. It's easy with the adjacent sibling selector!

```
input:checked + label {
  background-color: rgba(0, 0, 0, 0.25);
}
```



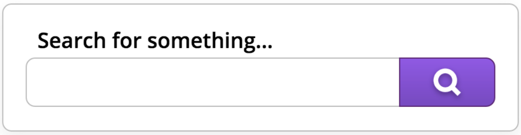
And that's all there is to a multi-button! Plus, what you've built is flexible—it's easy to add as many or as few options as you want. Check out this example with five options:

```html
<fieldset>
  <label>
    How many siblings do you have?
  </label>
  <div class="multi-button">
    <input type="radio" name="siblings"
      id="no-siblings" value="0">
    <label for="no-siblings">0</label>
    <input type="radio" name="siblings"
      id="one-sibling" value="1">
    <label for="one-sibling">1</label>
    <input type="radio" name="siblings"
      id="two-siblings" value="2">
    <label for="two-siblings">2</label>
    <input type="radio" name="siblings"
      id="three-siblings" value="3">
    <label for="three-siblings">3</label>
    <input type="radio" name="siblings"
      id="four-siblings" value="4+">
    <label for="four-siblings">4+</label>
  </div>
</fieldset>
```



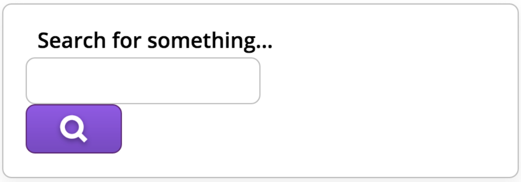
Attached Buttons

An attached button is simply a button stuck to something else, such as an input. They're handy for things like search fields.



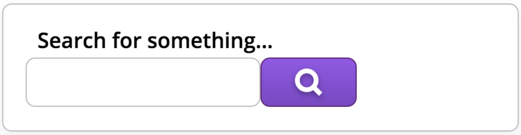
First up is—you guessed it—the HTML!

```
<form>
  <fieldset>
    <label>
      Search for something…
    </label>
    <div class="search-container">
      <input type="text">
      <button>
        <img src="search.svg" alt="Search">
      </button>
    </div>
  </fieldset>
</form>
```



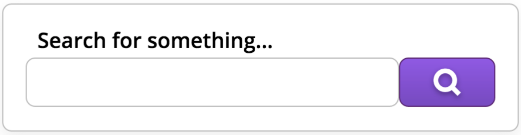
Like the multi-button, you need to make `<div class="search-container">` a flex container.

```
.search-container {
  display: flex;
}
```



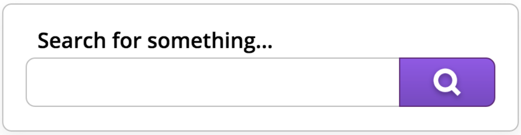
In an attached button, the button is a fixed size and the input stretches to fill the remaining space. Once again, the `flex` property comes to the rescue! For the input, you'll set `flex-grow` and `flex-shrink` to `1` and `flex-basis` to `0px`. This will force the input to grow to take up as much space as it can. With the button, you'll do the opposite: set `flex-grow` and `flex-shrink` to `0` and `flex-basis` to `auto`.

```
input {
  flex: 1 1 0px;
}

button {
  flex: 0 0 auto;
}
```



Lookin' sharp! The last thing to do is style the borders so they're attached!

```
input {
  flex: 1 1 0px;

  border-top-right-radius: 0;
  border-bottom-right-radius: 0;
  border-right: none;
}

button {
  flex: 0 0 auto;

  border-top-left-radius: 0;
  border-bottom-left-radius: 0;
}
```



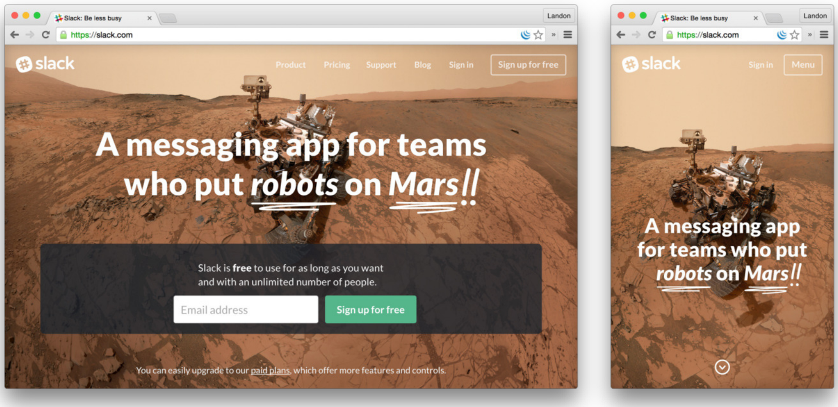
Final Thoughts

Flexbox can be used to build all sorts of little controls like this. If you start thinking of HTML and CSS in this manner, you'll learn to build small components that can be used everywhere in your application. When you get to that point, your code will be lighter, more maintainable and a joy to work with!

CHAPTER 8

Responsive Design

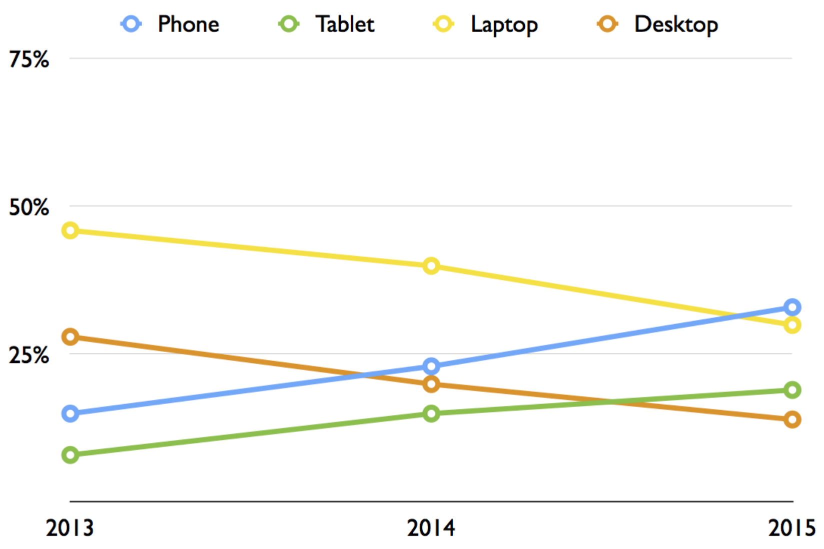
If there's one topic in CSS that's buzzing, it's responsive design. Responsive design is a technique for building layouts that adapt to multiple devices and screen sizes. It's about respecting your users' choices for how they view your site. Take a look at this example from [Slack's landing page](). See how the layout adapts to the size of the window?



Slack is **free** to use for as long as you want and with an unlimited number of people.

You can easily upgrade to our paid plans, which offer more features and controls.

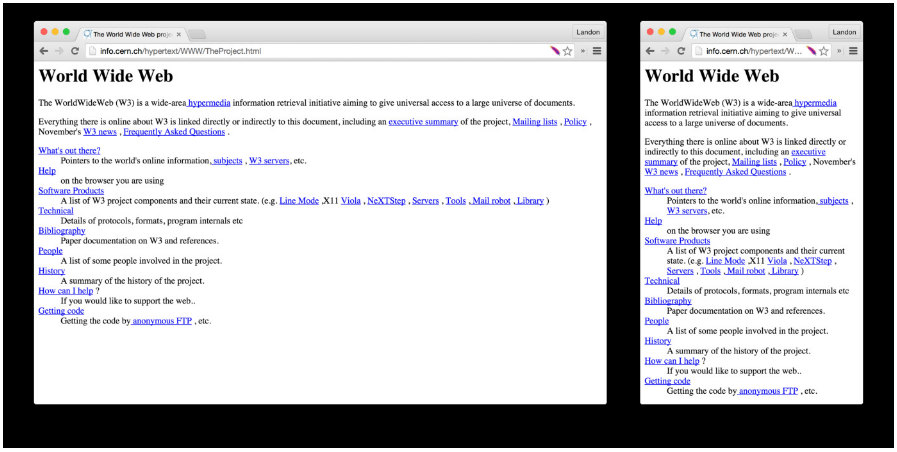
These days, more people are using phones than desktop computers to access the web. If you're not building responsive layouts, you're excluding those users. This graph from [John Gruber of Daring Fireball]() says it all.



In this chapter, I'm going to show you how to combine flexbox and responsive design to build easy, adaptable layouts.

Solutions without Media Queries

The web is responsive by default. Take a look at the first website on a laptop and on your phone. Notice how the content fills up the space? There's no fancy CSS making that happen.


The World Wide Web proje × Landon

info.cern.ch/hypertext/WWW/TheProject.html

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

The World Wide Web proje × Landon

info.cern.ch/hypertext/W...

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

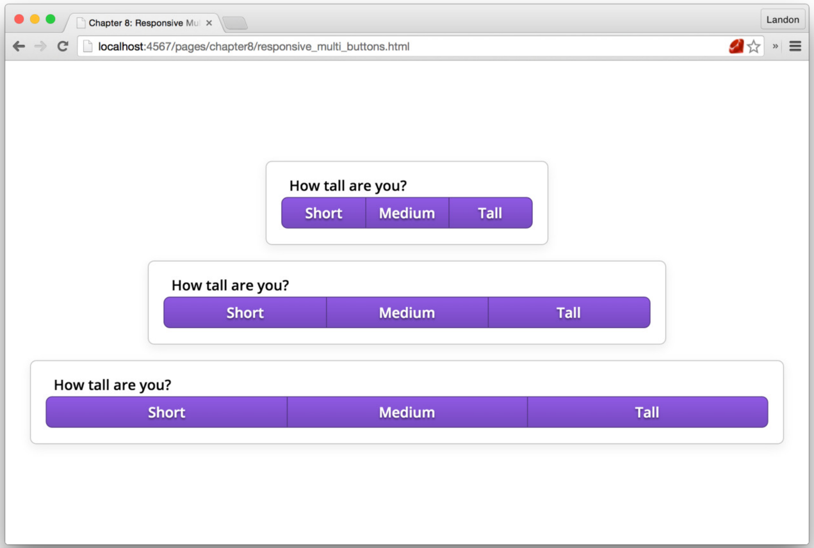
It's *the developers* who make the web unresponsive. We set fixed widths and heights for elements. We design sites for desktops, and ignore other screen sizes. We forget some people don't use a mouse when browsing our sites. These are habits we have to break to build responsive layouts.

Mobile-First

In almost every case, a website's mobile layout is simpler than its desktop layout. Mobile layouts are usually single columns that fill the full width of their containers. Because of this, it's much easier to start by building the mobile version of your site, and then add the layouts for larger screens in media queries. This is known as the *mobile-first* approach.

Fill Up the Provided Space

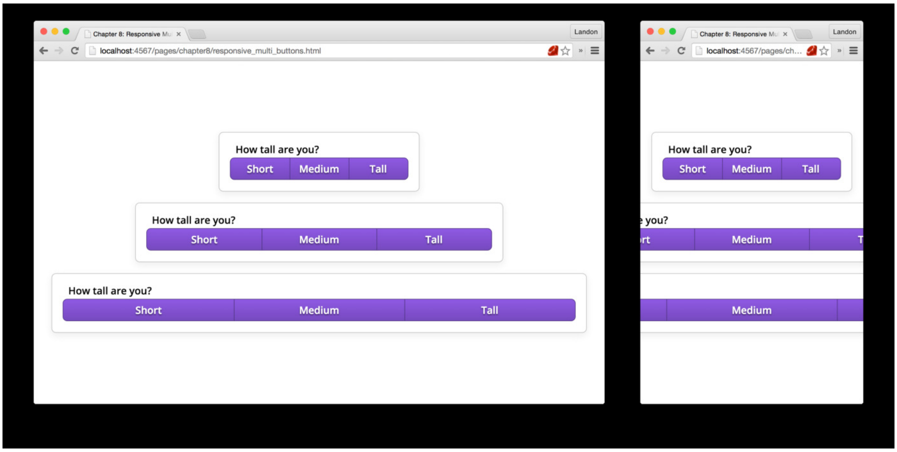
The first rule of writing responsive flexbox controls is to make your controls fill up the width they're given. Take the multi-button from the previous layout for example. Notice how the control adapts to the size of the container?



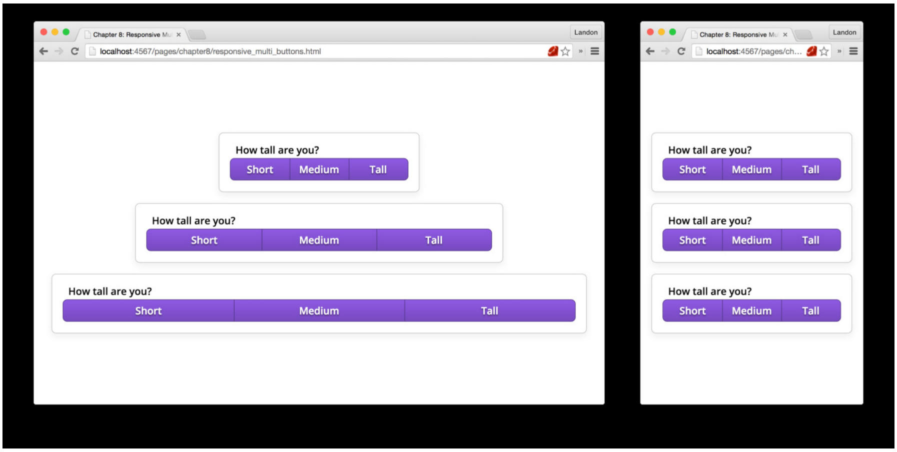
The trick is to let parent elements worry about containing their children. In this case, the form is responsible for defining the width and the multi-button is responsible for filling it. The advantage of this approach is the child element can be reused without having to change its CSS.

Maximum and Minimum Sizes

At certain screen sizes, some controls look a little crummy. Look at the multi-button on a small screen.



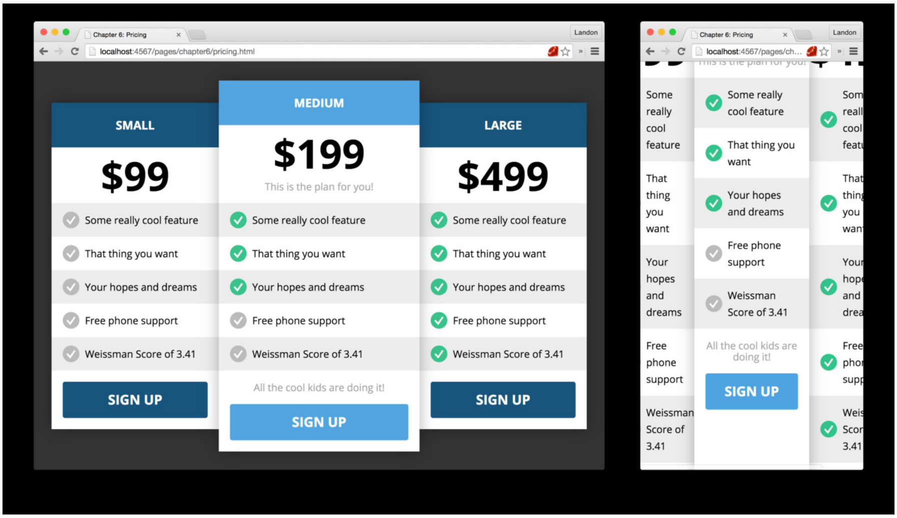
You *could* set the width of the container, but then you'd be breaking the previous guideline. Instead, try setting the `max-width` of your control. That allows the control to grow until it runs out of room.



If you have a control that's too big for its layout, such as a large table, set its `min-width` and make its parent horizontally scroll with `overflow-x`.

Breakpoints

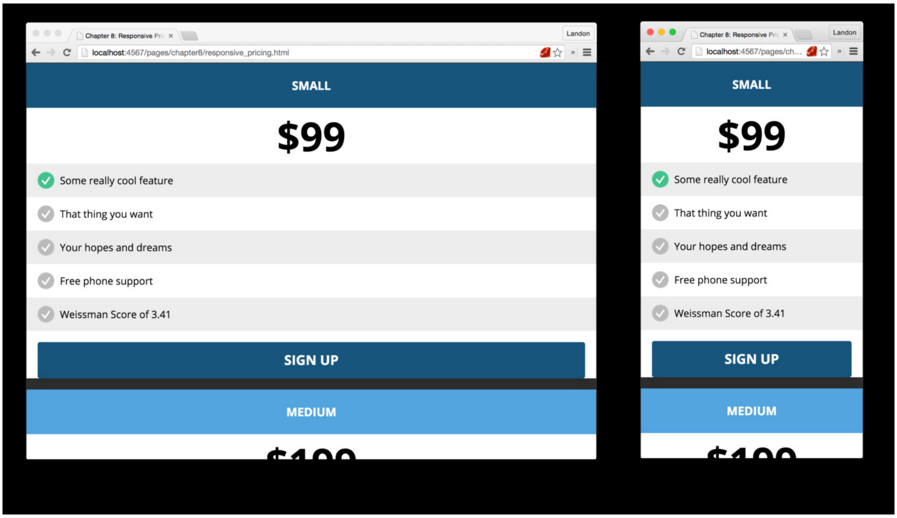
The meat and potatoes of a responsive layout is using media queries to style controls differently on mobile and desktop. The screen sizes where your layout changes are known as *breakpoints*. Take our pricing layout from Chapter 6 for example. On small screens, it looks broken.



However, with a few additions, it's easy to make it shine. Let's start by commenting out the column styles so the pricing tiers stack.

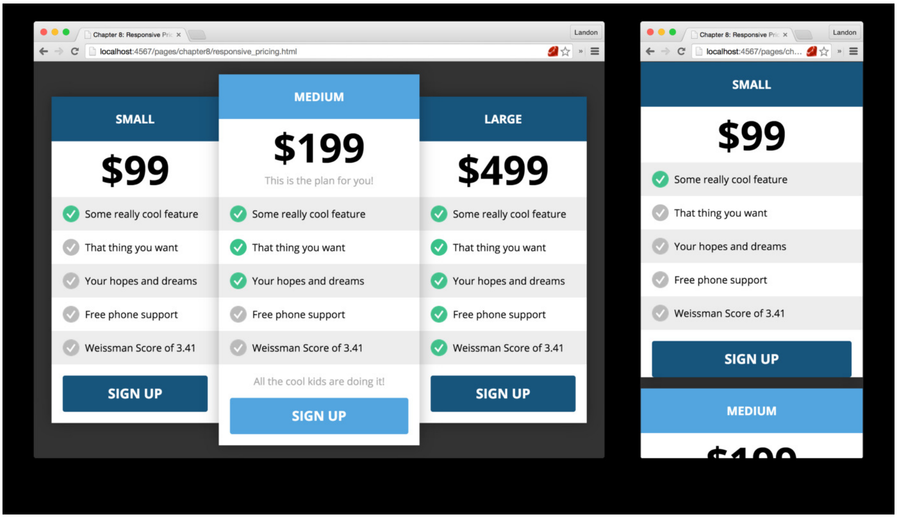
```
li {
  display: flex;
  line-height: 30px;
  padding: 15px 20px;
}


li img {
  margin-right: 10px;
}
```

```css
button {
  display: block;
  width: calc(100% - 40px);
  margin: 20px;
}

/*
.pricing {
  display: flex;
  width: 960px;
}

.plan {
  flex: 1 1 0px;
}

.plan:nth-of-type(1), .plan:nth-of-type(3) {
  margin-top: 40px;
  margin-bottom: 40px;
}

.plan:nth-of-type(2) {
  flex-basis: 60px;
}
*/
```



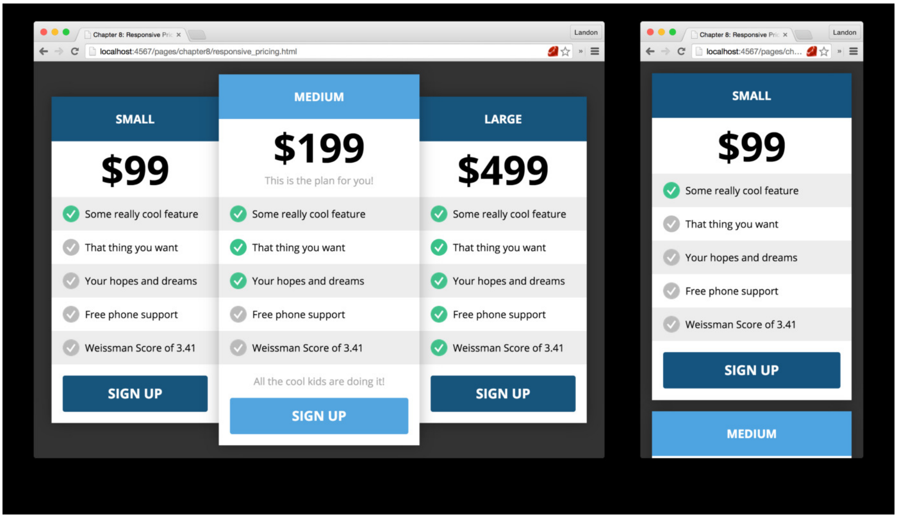
Next, add back in the CSS to turn the columns into three tiers, but only when the screen is large enough to comfortably fit them.

```
@media (min-width: 640px) {
  .pricing {
    display: flex;
    width: 960px;
  }

  .plan {
    flex: 1 1 0px;
  }
  .plan:nth-of-type(1), .plan:nth-of-type(3) {
    margin-top: 40px;
    margin-bottom: 40px;
  }

  .plan:nth-of-type(2) {
    flex-basis: 60px;
  }
}
```



Finally, add a little space around the plans on mobile, and prevent their margins from collapsing.

```
.plan {
  margin: 20px;

  /* prevent the margins from collapsing */
  padding-bottom: 1px;
}

@media (min-width: 640px) {
  .plan {
    margin: 0;
  }
}
```


Chapter 8: Responsive Pri
localhost:4567/pages/chapter8/responsive_pricing.html
SMALL
$99
MEDIUM
$199
This is the plan for you!
LARGE
$499
Some really cool feature
That thing you want
Your hopes and dreams
Free phone support
Weissman Score of 3.41
All the cool kids are doing it!
SIGN UP
localhost:4567/pages/ch...

Do you see how the mobile-first paradigm plays out here? The mobile layout is simpler, so that's where you start. When the width of the window hits 640 pixels or more, your layout shifts and uses columns instead.

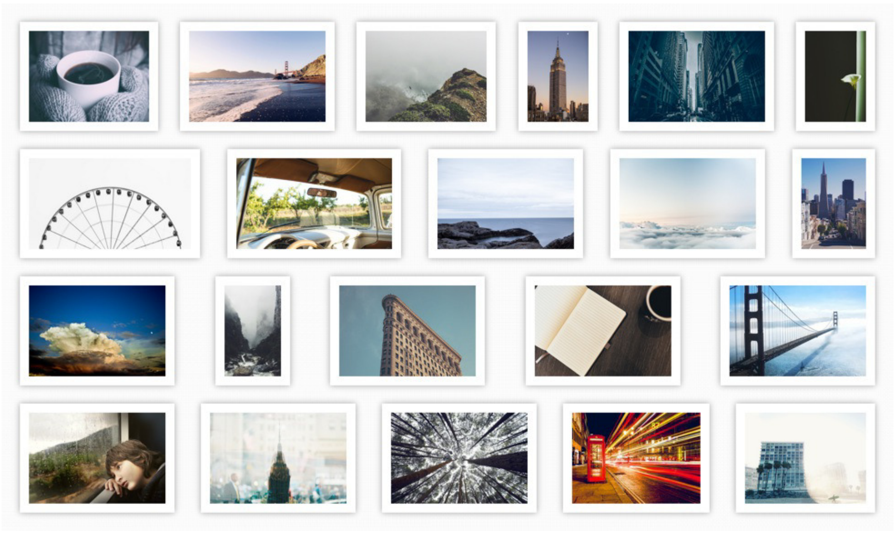
That's All

In this chapter, you've seen the most important responsive techniques and how they work with flexbox. You've also seen how several of the previous layouts in this book can be made responsive.

If you're looking for a good challenge, try making *all* of the layouts we've covered so far responsive. It's good practice!

CHAPTER 9

Wrapping Like a Boss



Everyone love photos! From selfie sticks to DSLRs, people constantly find new ways to capture moments in their lives. Between smartphones, Instagram and Facebook, most people are carrying around thousands of photos in their pocket.

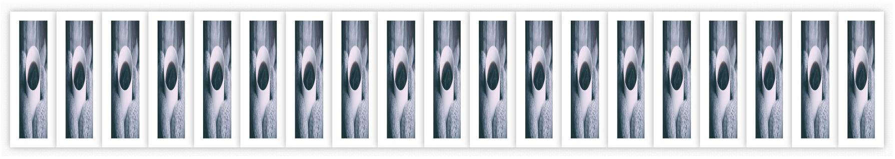
Wouldn't it be nice to have a way to show off these photos and beef up your flexbox skills at the same time? In this chapter, you'll learn how flexbox can span multiple lines with the `flex-wrap` property, and you'll use this property to build a sweet photo layout.

A Simple Example

Let's start with an easy layout. To keep things simple, you'll use the same photo multiple times. Here's the markup and the basic CSS.

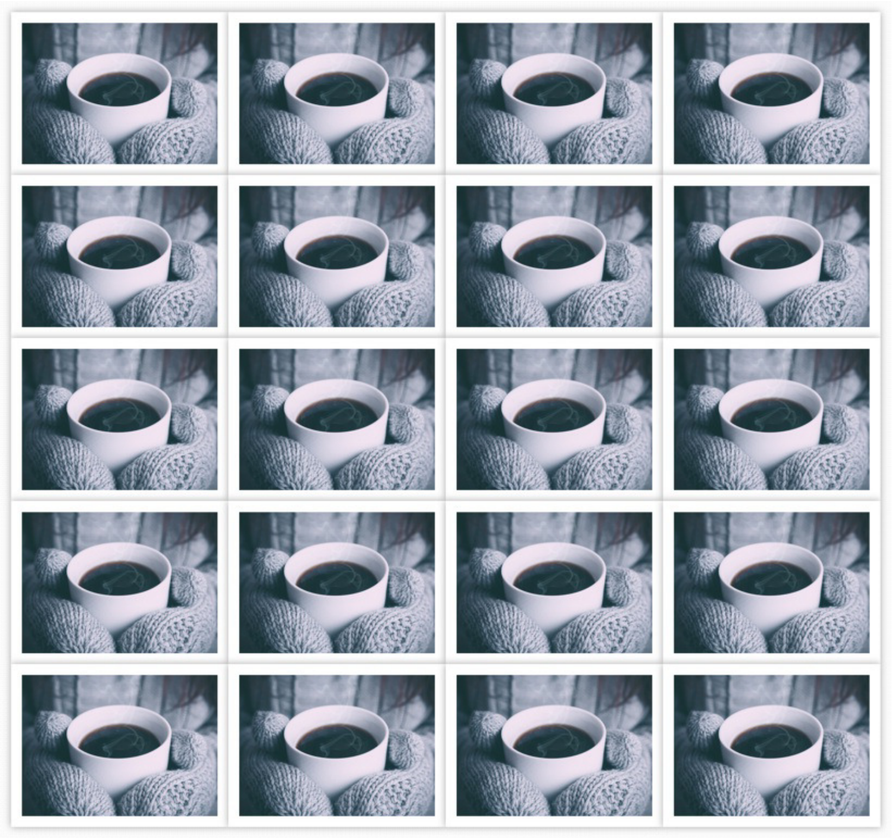
```
<section>
  <img src="/photo1.jpg" alt="Photo">
  <img src="/photo1.jpg" alt="Photo">
  <img src="/photo1.jpg" alt="Photo">
  ...
  <img src="/photo1.jpg" alt="Photo">
</section>
```

```
section {
  display: flex;
}
```

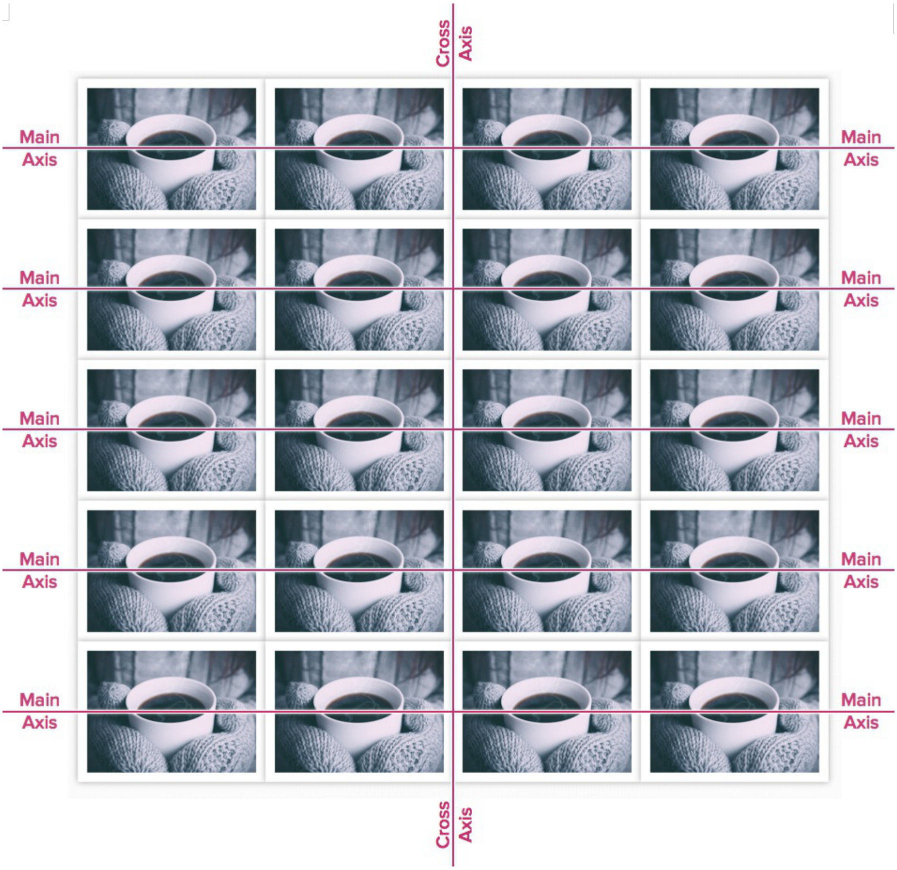


Flexbox tries to cram as many items as it can onto one line, because the default value of `flex-shrink` is `1`. What we want is for the items to drop to the next line when the first line is full. That's where the `flex-wrap` property comes in!

```
section {
  display: flex;
  flex-wrap: wrap;
}
```

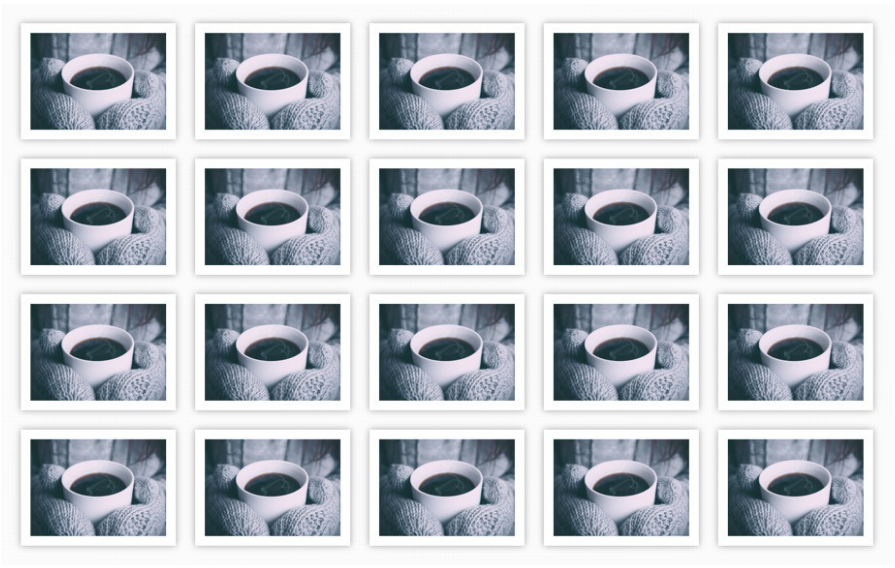


That's better! Flexbox ran out of room on the first line, so it wrapped the items to the next line. The main axis and cross axis now look like this:



The images are a little big. Let's shrink them down and add some space between them.

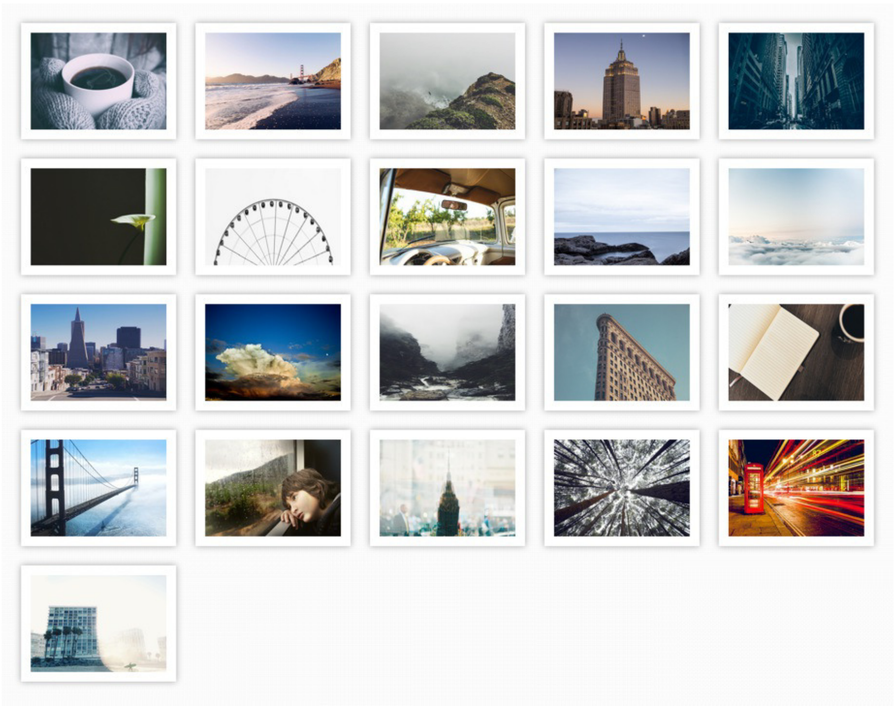
```
img {
  width: 160px;
  height: 120px;
  margin: 10px;
}
```



Different Shapes and Sizes

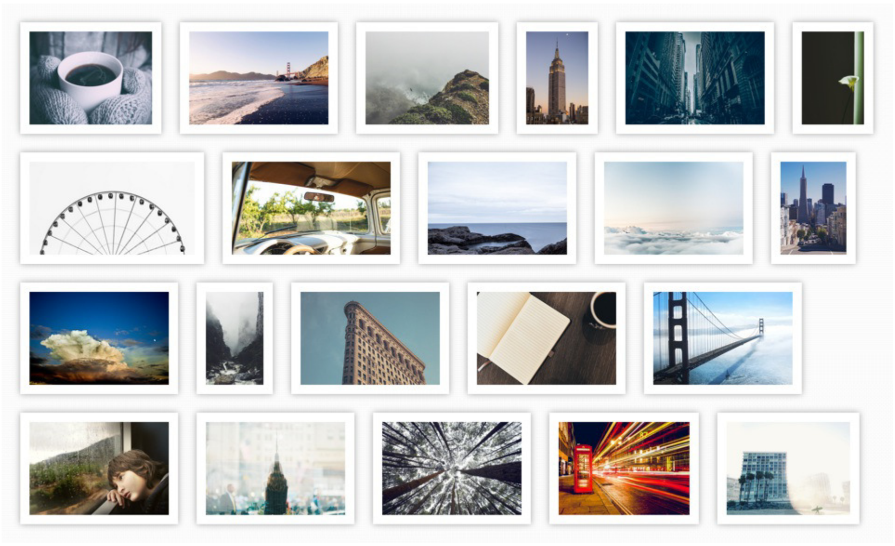
Your photo library would be a little boring if it only showed one image. Photo libraries are full of photos that are different sizes and shapes. Go ahead and update your HTML to include multiple image and sizes.

```
<section>
  <img src="/photo1.jpg" alt="Photo">
  <img src="/photo2.jpg" alt="Photo">
  <img src="/photo3.jpg" alt="Photo">
  ...
  <img src="/photo21.jpg" alt="Photo">
</section>
```



Yikes! The portrait photos, such as the building, are way too scrunched. What you want is for the photos to retain their aspect ratio, but stay the same height.

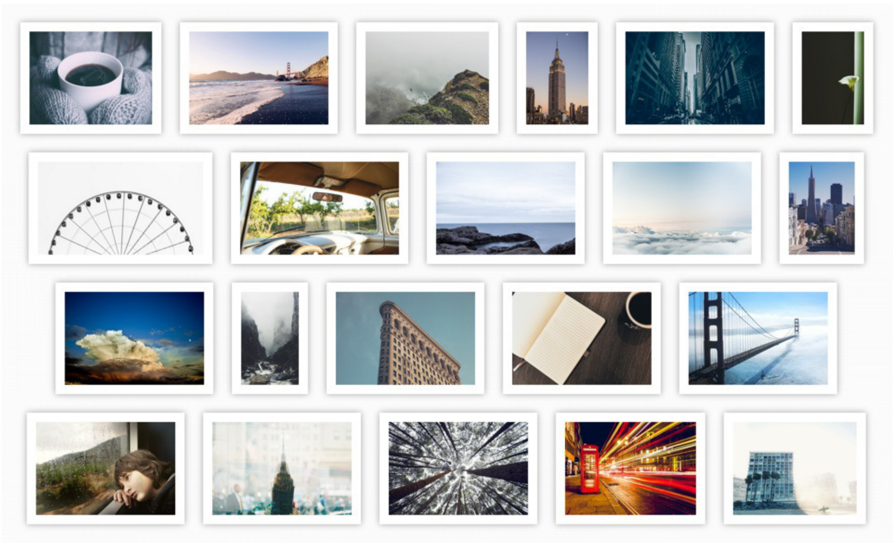
```
img {
  width: auto;
  height: 120px;
  margin: 10px;
}
```



Now the right side of the library looks a little wonky. Wouldn't it be great if we could arrange the images to look a little nicer? We can with the `justify-content` property! Remember, `justify-content` tells flexbox how to distribute space between items along the main axis. What's really cool is it even works when the content is wrapped!

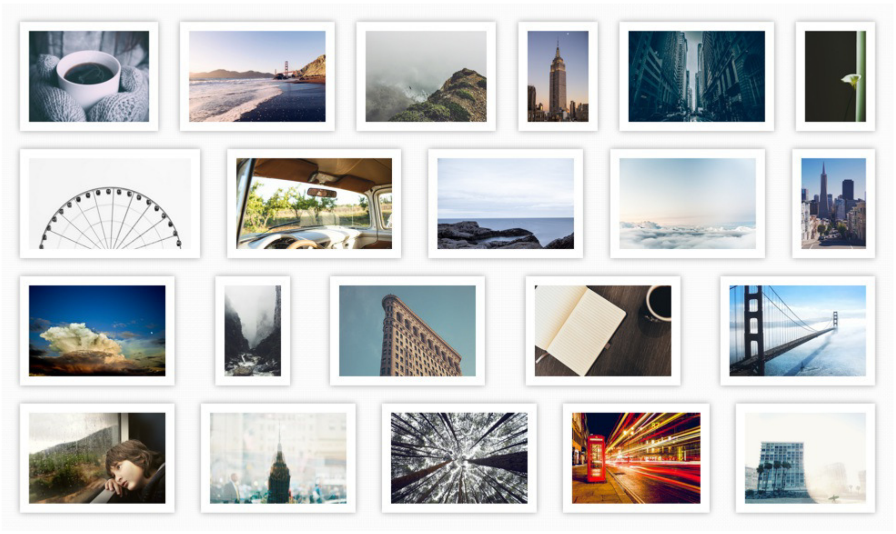
Let's try centering the content.

```
section {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
}
```



Nice! My favorite value of `justify-content` is `space-between`, which aligns the left and right sides of the images.

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



Doesn't that look better?

Align Content

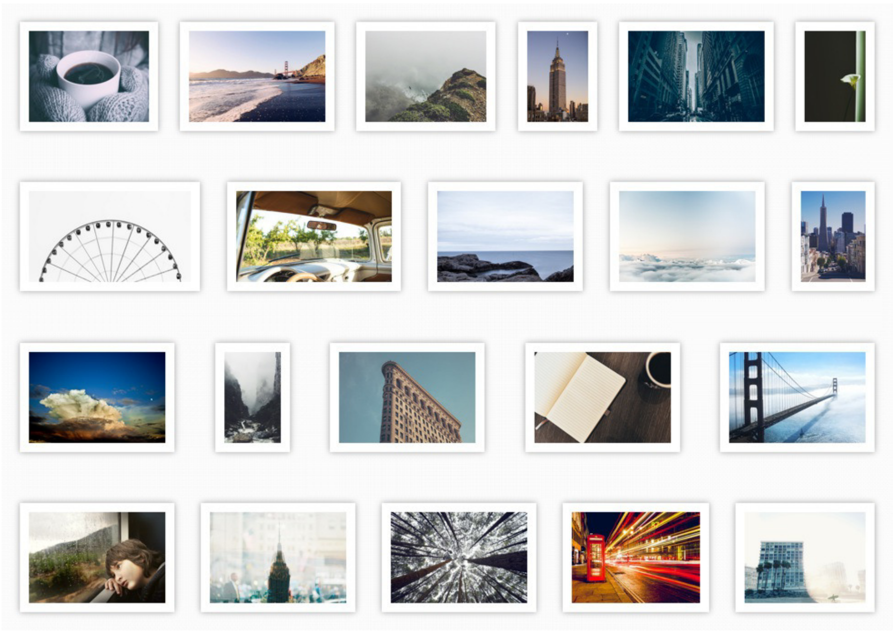
By default, when a flex container has a fixed height, flexbox stretches the items to fit. We can change that if we want to. The `align-content` property determines how space is distributed between lines along the cross axis. This property takes six values:

- `flex-start`: Align the content to the start of the container.
- `flex-end`: Align the content to the end of the container.
- `center`: Center the content in the container.

- `stretch`: Stretch the content to fit in the container.
- `space-between`: Evenly distribute the space between the lines.
- `space-around`: Evenly distribute the space around the lines.

Let's try `space-between`.

```
html, body, section {
  height: 100%;
}

section {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
  align-content: space-between;
}
```



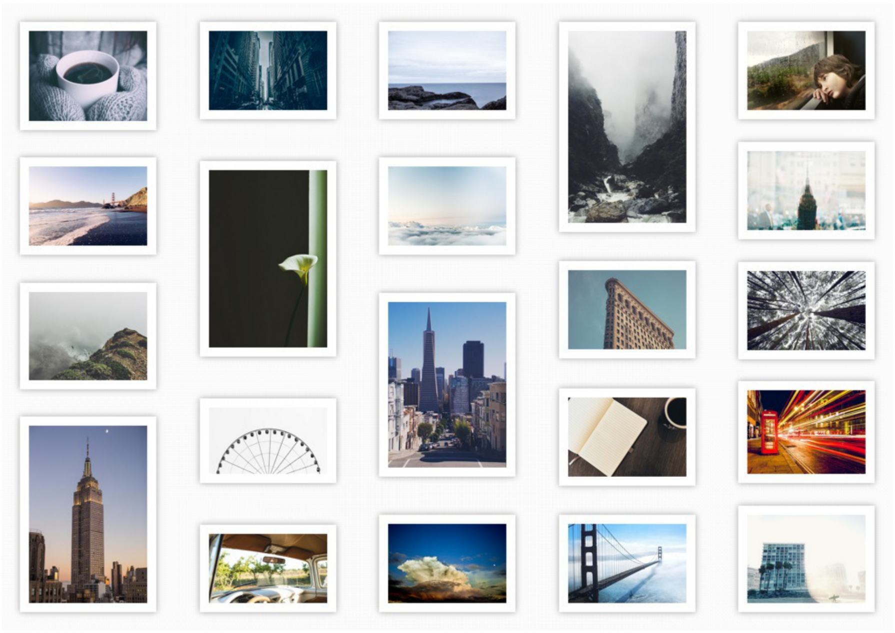
Direction Matters Too

Everything we've done works with column layouts! All we need to do is set `flex-direction` to `column`. This switches the main axis with the cross axis and changes the direction of the properties.

Let's give it a shot.

```
html, body, section {
  height: 100%;
}
```

```
section {
  display: flex;
  flex-direction: column;
  flex-wrap: wrap;
  justify-content: space-between;
  align-content: space-between;
}

img {
  width: 150px;
  height: auto;
  margin: 10px;
}
```



Pretty cool, right?

Wrapping Up

In this chapter, you've learned how to use the `flex-wrap` property to build an awesome photo album. You've played with `align-content` to arrange the items, and you've even tried out wrapping with columns.

For a little extra practice, try building a layout that pulls down random images and still looks good. You can use a service like PugMe to grab images.

CHAPTER 10

Progressive Enhancement

Of all the things web developers like to complain about, one thing sticks out the most: old browsers. Developers *hate* supporting older browsers that don't include new technologies like flexbox. If you haven't already, you will eventually experience this pain. So what can you do?

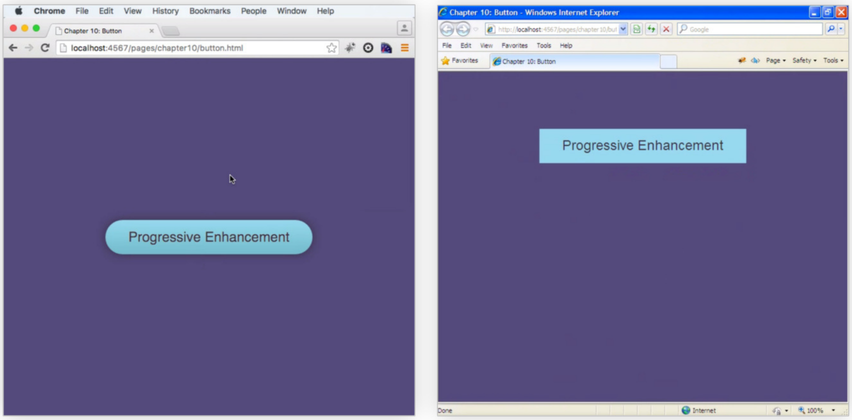
In this chapter, I'll teach you about *progressive enhancement*, a technique you can use to gain all of the flexbox goodness in newer browsers while still providing an acceptable experience in browsers that don't support it.

A Progressive Agenda

The main idea behind progressive enhancement is to build a basic site that functions across all major browsers. It might not be pretty, but it should be usable. For browsers that support newer features, you *progressively* add features to *enhance* the experience. One of the brilliant things about CSS is browsers ignore properties they don't recognize, so you can feel free to embrace the new while falling back to the old.

Let's say you're writing a site and that needs to support Internet Explorer 8. You'd like for your buttons to have rounded corners, a nice gradient and box shadows, but IE8 doesn't support any of these properties. Does this mean you can't use them? Of course not! You simply make a basic button for IE8. It's still a perfectly usable button that does all the things a button should. Then, for browsers that support the modern features, you add the extra functionality.

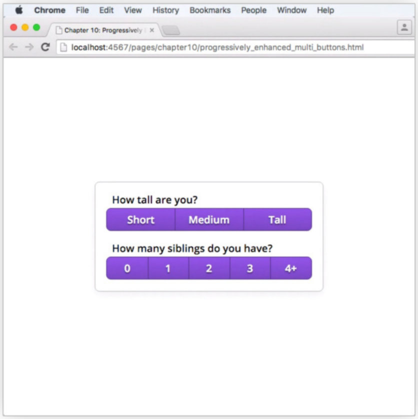
```
body {
  text-align: center;
  font-size: 24px;
  background-color: #524a79;

  /* modern properties */
  display: flex;
  align-items: center;
  justify-content: center;
}

button {
  margin: 100px 0;
  padding: 20px 40px;
  line-height: 20px;
  border: none;
  background-color: #97d8ec;
  color: #3e3245;

  /* modern properties */
  border-radius: 30px;
  background-color: #97d8ec;
  background: linear-gradient(to bottom, #97d8ec,
    #74bbca);
  text-shadow: 0 0 1px rgba(255, 255, 255, 0.5);
  box-shadow: 0 0 20px #3e3245;
}
```

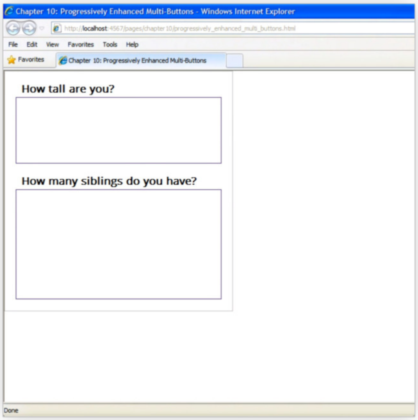

Chapter 10: Button
localhost:4567/pages/chapter10/button.html
Progressive Enhancement
Chapter 10: Button - Windows Internet Explorer
Progressive Enhancement

Progressive enhancement with flexbox follows the same idea. Your goal with the basic layout is to get information across, not to make it perfect.

Modernizr

There are some layouts that depend on modern features. Take the multi-button example from chapter 7. In an older browser, it looks like this:

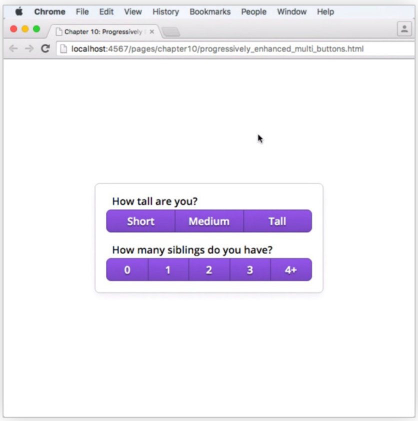

Chrome File Edit View History Bookmarks People Window Help
Chapter 10: Progressively
localhost:4567/pages/chapter10/progressively_enhanced_multi_buttons.html
How tall are you?
Short
Medium
Tall
How many siblings do you have?
0
1
2
3
4+

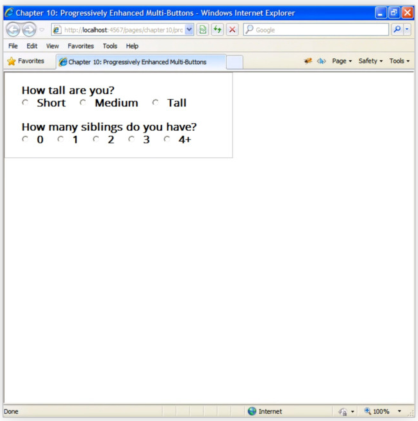

Chapter 10: Progressively Enhanced Multi-Buttons - Windows Internet Explorer
http://localhost:4567/pages/chapter10/progressively_enhanced_multi_buttons.html
File Edit View Favorites Tools Help
Favorites
Chapter 10: Progressively Enhanced Multi-Buttons
How tall are you?
How many siblings do you have?
Done

So what can you do? One option is to use a tool like Modernizr. Modernizr is a library that appends classes to the `<html>` element to tell you which features are supported by that browser. If you include this library, you can rewrite the multi-button CSS like this:

```
.flexbox label {
  ...
}

.flexbox input {
  ...
}
```

```
.flexbox .multi-button {
  ...
}

.flexbox .multi-button label {
  ...
}

.flexbox .multi-button
label:not(:first-of-type) {
  ...
}

.flexbox input:checked + label {
  ...
}
```


Chrome File Edit View History Bookmarks People Window Help
Chapter 10: Progressively
localhost:4567/pages/chapter10/progressively_enhanced_multi_buttons.html
How tall are you?
Short
Medium
Tall
How many siblings do you have?
0
1
2
3
4+


Chapter 10: Progressively Enhanced Multi-Buttons - Windows Internet Explorer
File Edit View Favorites Tools Help
Favorites
Chapter 10: Progressively Enhanced Multi-Buttons
How tall are you?
Short
Medium
Tall
How many siblings do you have?
0
1
2
3
4+
Done
Internet
100%

Isn't that great? On browsers that support flexbox, the `flexbox` class is added to the `<html>` element and you provide a nice multi-button. On browsers that don't, there's no `flexbox` class so you fall back to radio buttons.

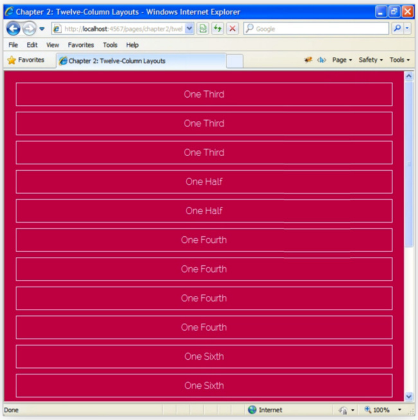
Last Resort

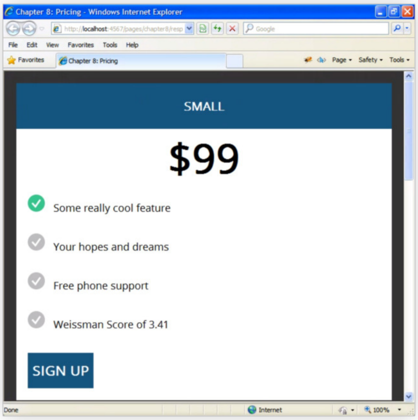
Sometimes, your layout completely depends on a browser supporting a certain feature. One last resort is to display a message letting users know they're using an older browser that your site doesn't work on. If you're using Modernizr, you can do this with a little JavaScript.

```
$(function() {
  if ($("html").hasClass("no-flexbox")) {
    alert("...");
  }
});
```

Generally speaking, this is a *bad idea*. **Only do it if your site absolutely doesn't work without flexbox.** Also, be careful with how your site interacts with non-sighted users. Sometimes, screen readers disable features that aren't needed by their users.

Examples





If you look at the twelve-column layouts from chapter 4, they're *already* progressively enhanced. On older browsers, they simply become one-column layouts. The content is still readable, which is appropriate for older browsers.

The responsive pricing layout also works in older browsers. One of the cool things about the mobile-first philosophy is the mobile layouts tend to be simpler, single-column layouts, which play well with older browsers. This lends itself well to progressive enhancement.

Going Forward

Hopefully the problem of older browsers lacking modern features will soon be a thing of the past. Microsoft's newest browser, Edge, is *evergreen*, meaning it's always kept up to date. Chrome and Firefox are also evergreen, and Safari users do a good job of upgrading. Soon, we'll be living in a world where we can use the web's latest features without worrying too much about breaking older browsers. Woohoo!

Until then, progressive enhancement is a powerful technique you can use in your CSS. Done right, it'll make people with newer browsers happy and let people with older browsers still use your site. Have fun!

CHAPTER 11

Ordering

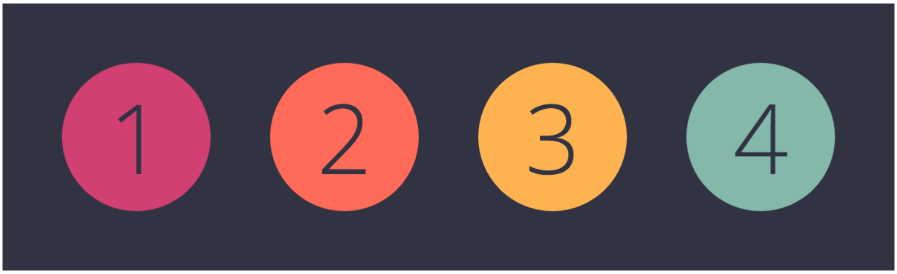
So far, you've learned almost all of the flexbox properties and how to use them. However, you're still missing one piece of the puzzle: ordering. It's not the most glamorous flexbox property, but it's damn handy when you need it.

In this chapter, I'm going to tell you why you should care about ordering and show you *two* different ways you apply it to your layouts.

Flexbox to the Rescue

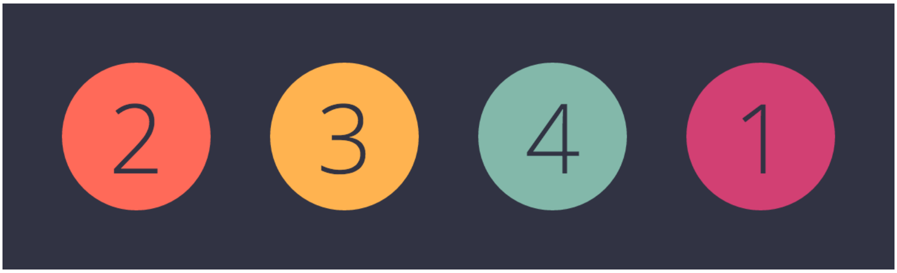
The most common way to reorder elements in a flex container is by using the `order` property. You set `order` for any of the children in a flex container to an integer and the items will be sorted by those values. For example, let's say you have a list of items.

```
<ol class="numbers">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
</ol>
```



The *value* of `order` for an element doesn't matter by itself. What is important is the *value relative to the other flex items* . Let's say you set the `order` property for the first list item to `1`.

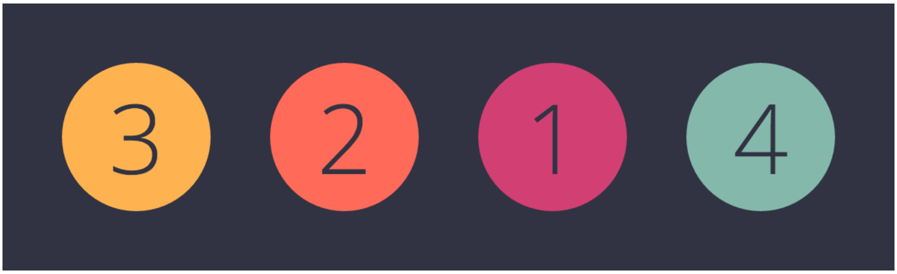
```
li:nth-of-type(1) {
  order: 1;
}
```



Why did the first `<li>` move to the back? The default value of `order` is `0`, so the second, third and fourth `<li>`s were all lower than `1`. If the values for `order` match, the ordering falls back to the original order in the HTML.

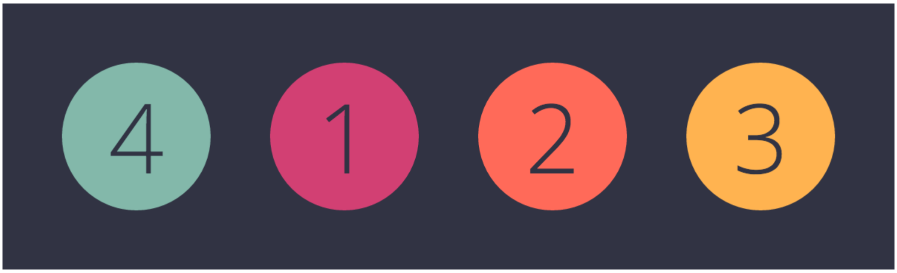
It doesn't matter how far apart the values are.

```
li:nth-of-type(1) {
  order: 3000;
}

li:nth-of-type(2) {
  order: 2000;
}

li:nth-of-type(3) {
  order: 1000;
}

li:nth-of-type(4) {
  order: 4000;
}
```



Finally, order accepts negative numbers. Since the default value of `order` is `0`, this is a handy way to move an item to the front.

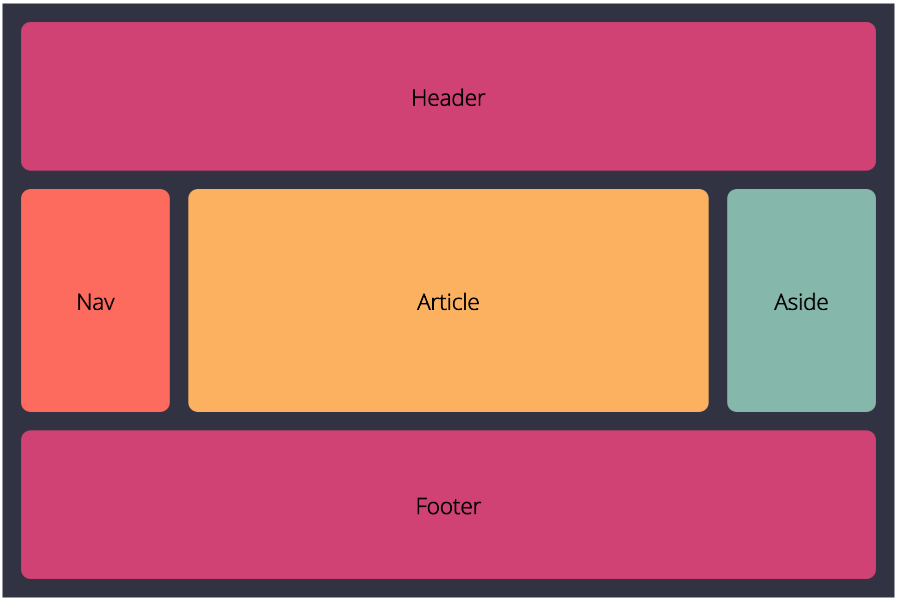
```
li:nth-of-type(4) {
  order: -1;
}
```



Why Bother?

The biggest reason to reorder elements is accessibility. The order property doesn't affect non-visual media, such as screen readers. This means you can write your HTML in a way that makes sense logically, and then change how it's displayed visually. For example, take a look at this Holy Grail layout example from the W3 flexbox specification.

```
<header>...</header>
<main>
   <article>...</article>
   <nav>...</nav>
   <aside>...</aside>
</main>
<footer>...</footer>
```



In the past, it hasn't been possible to build this layout without rearranging your HTML. With `order`, it's easy. Assuming the `<main>` element is a flex container, you can achieve the correct order like this:

```
article {
  order: 2;
}

nav {
  order: 1;
}

aside {
  order: 3;
}
```

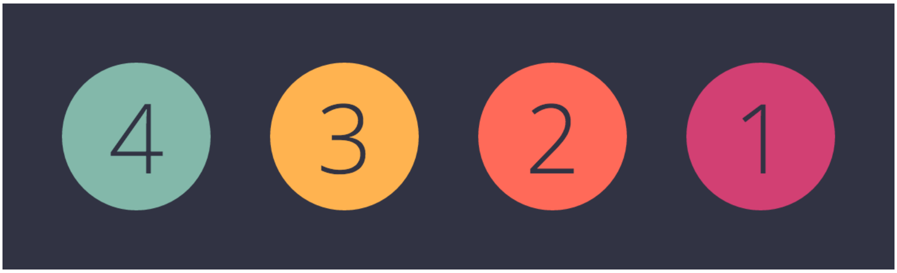
The other reason for reordering elements is responsive design. You can use the `order` property to rearrange elements as the viewport changes size.

The flex-direction Property

The `flex-direction` property isn't quite as flexible as `order`, but it can be used to reverse the order of elements in a container. We've seen two of the values for this property: `row` and `column`. `flex-direction` also accepts `row-reverse` and `column-reverse`.

Let's look at the same example from above.

```
ol {
  flex-direction: row-reverse;
}
```



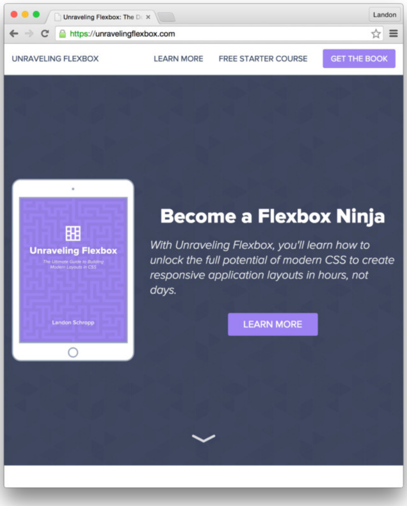
Order Up

You've now learned all of the flexbox properties! You can use the `order` property for fine-grained arrangement or `flex-direction` to reverse all of the items in a flex container. The next time you run into a tricky problem that requires rearranging elements, you'll know exactly what to do!

CHAPTER 12

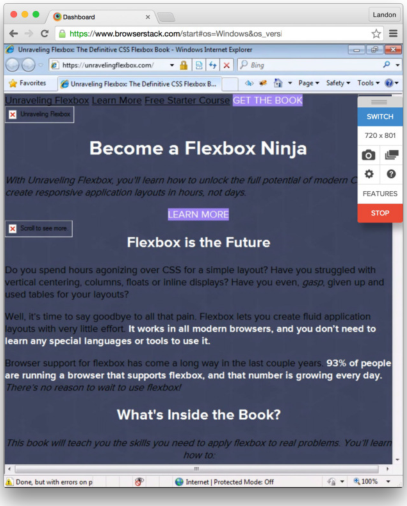
Cross-Browser Testing

You've spent weeks meticulously crafting your layout. It's perfect—a real masterpiece. The night before you're going to release your design to the world, you decide to try it out in Internet Explorer, just to make sure everything looks okay.



Become a Flexbox Ninja

With Unraveling Flexbox, you'll learn how to unlock the full potential of modern CSS to create responsive application layouts in hours, not days.


Dashboard
Landon
https://www.browserstack.com/start#os=Windows&os_versi
Unraveling Flexbox: The Definitive CSS Flexbox Book - Windows Internet Explorer
https://unravelingflexbox.com/
Bing
Favorites
Unraveling Flexbox: The Definitive CSS Flexbox B...
Page
Safety
Tools
Unraveling Flexbox Learn More Free Starter Course GET THE BOOK
Unraveling Flexbox
SWITCH
720 x 801
FEATURES
STOP
LEARN MORE
Scroll to see more.
Done, but with errors on p
Internet | Protected Mode: Off
100%

Become a Flexbox Ninja

With Unraveling Flexbox, you'll learn how to unlock the full potential of modern C create responsive application layouts in hours, not days.

Flexbox is the Future

Do you spend hours agonizing over CSS for a simple layout? Have you struggled with vertical centering, columns, floats or inline displays? Have you even, *gasp*, given up and used tables for your layouts?

Well, it's time to say goodbye to all that pain. Flexbox lets you create fluid application layouts with very little effort. **It works in all modern browsers, and you don't need to learn any special languages or tools to use it.**

Browser support for flexbox has come a long way in the last couple years. **93% of people are running a browser that supports flexbox, and that number is growing every day.** *There's no reason to wait to use flexbox!*

What's Inside the Book?

This book will teach you the skills you need to apply flexbox to real problems. You'll learn how to:

It's ruined! Everything you worked so hard to create is completely broken in IE! What can you do?

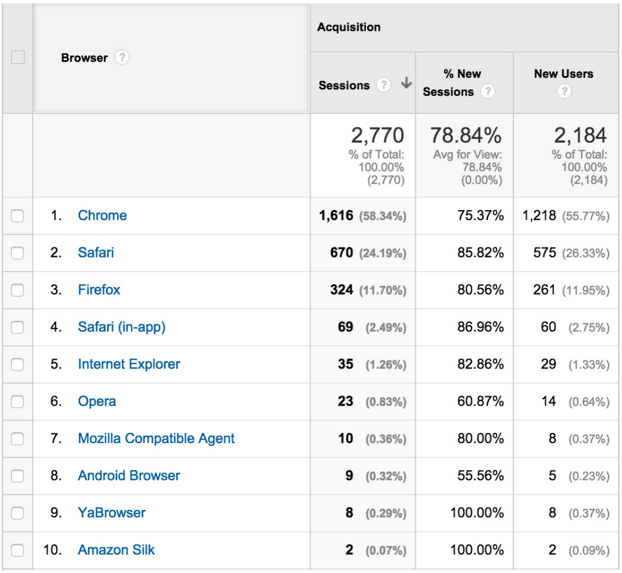
Don't worry, this happens *all the time*. Browsers have rendering quirks that break certain layouts. Some are worse than others *(cough IE cough)*. Unless you're actively testing in different browsers, chances are good your CSS will fail somewhere.

In this chapter, I'll show you how to decide which browsers to support and walk you through how to test your code in each one. I'll also show you where to find help if you run into a bug.

Deciding What to Support

Before you can test, you need to decide which browsers you'll support and which ones you'll ignore. In an ideal world, you'd be able to support everything. However, doing this is *extremely* time consuming. In my experience, supporting IE 7 and 8 *doubles* the amount of time you spend writing CSS, and IE 6 *quadruples* it.

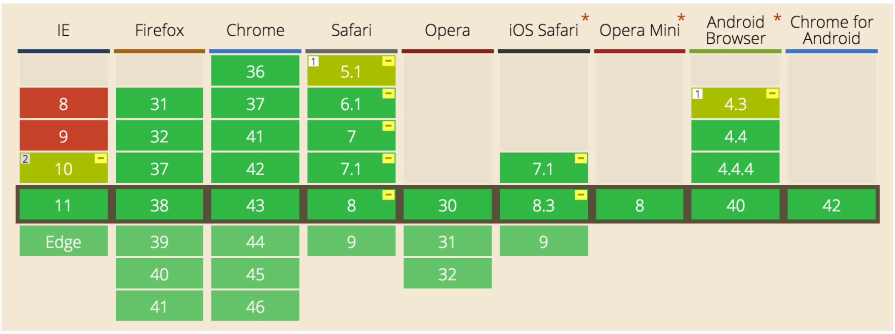
In most cases, you'll want to look at your target audience and decide what's reasonable. You can find this information in Google Analytics.



Browser	Sessions	% New Sessions	New Users
	2,770 % of Total: 100.00% (2,770)	78.84% Avg for View: 78.84% (0.00%)	2,184 % of Total: 100.00% (2,184)
1. Chrome	1,616 (58.34%)	75.37%	1,218 (55.77%)
2. Safari	670 (24.19%)	85.82%	575 (26.33%)
3. Firefox	324 (11.70%)	80.56%	261 (11.95%)
4. Safari (in-app)	69 (2.49%)	86.96%	60 (2.75%)
5. Internet Explorer	35 (1.26%)	82.86%	29 (1.33%)
6. Opera	23 (0.83%)	60.87%	14 (0.64%)
7. Mozilla Compatible Agent	10 (0.36%)	80.00%	8 (0.37%)
8. Android Browser	9 (0.32%)	55.56%	5 (0.23%)
9. YaBrowser	8 (0.29%)	100.00%	8 (0.37%)
10. Amazon Silk	2 (0.07%)	100.00%	2 (0.09%)

Google Analytics browser stats

If you need a particular technology, your browser support will be determined by that technology. For flexbox, here's the list of supported browsers from Can I Use.



IE	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android * Browser	Chrome for Android
		36	5.1					
8	31	37	6.1				4.3	
9	32	41	7				4.4	
10	37	42	7.1		7.1		4.4.4	
11	38	43	8	30	8.3	8	40	42
Edge	39	44	9	31	9			
	40	45		32				
	41	46						

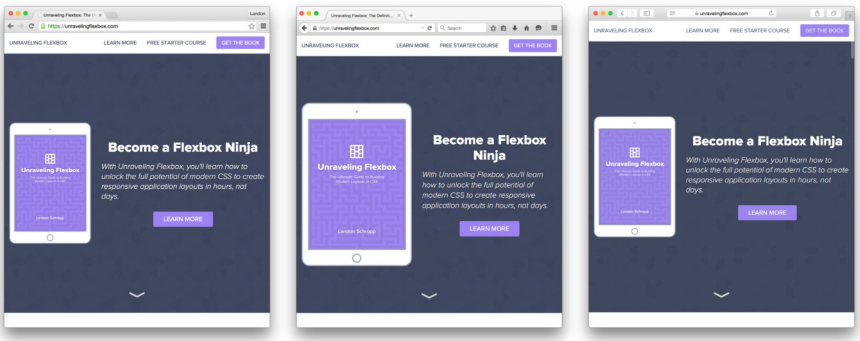
Can I User flexbox browser support

If you'd like to see the specific browser usage stats, check out the [Can I Use Usage Table](#).

Testing

Desktop Browsers

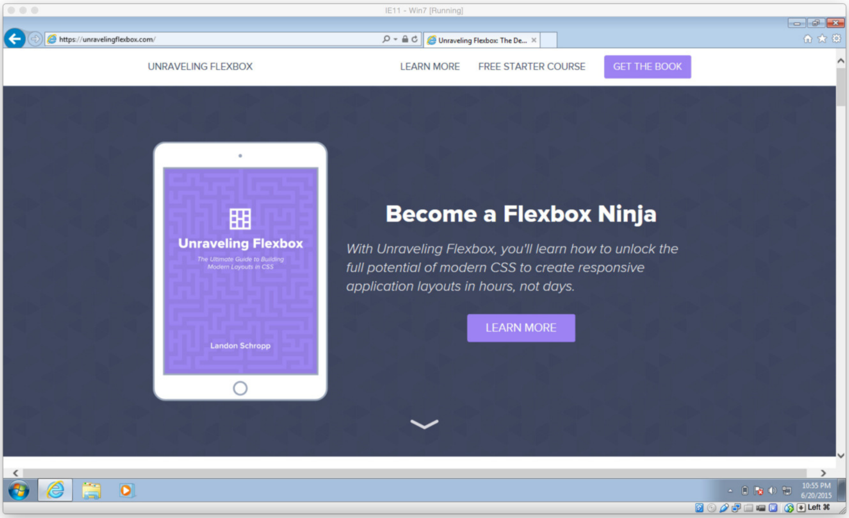
Testing desktop browsers is *much* easier than testing mobile browsers. For Chrome and Firefox, just download the browser and open up your site. Safari comes preinstalled on OS X, but isn't available for Windows.


UNRAVELING FLEXBOX
LEARN MORE
FREE STARTER COURSE
GET THE BOOK
Become a Flexbox Ninja
With Unraveling Flexbox, you'll learn how to unlock the full potential of modern CSS to create responsive application layouts in hours, not days.
LEARN MORE

Chrome, Firefox and Safari

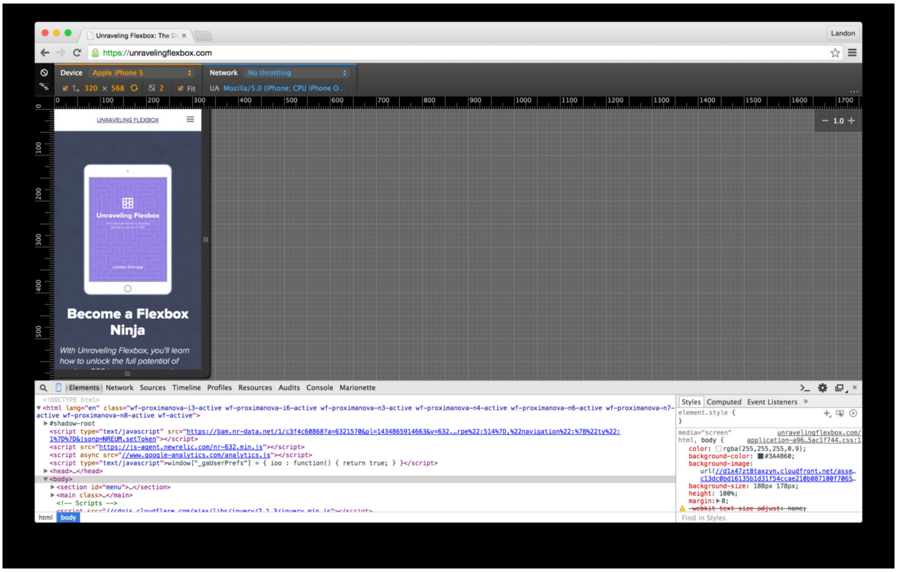
Microsoft has graciously released a collection of virtual machines to test Internet Explorer. You can download these *for free* on [Modern.IE](#). Thanks, Microsoft!

To use them, install an emulator such as VirtualBox. Then, download the virtual machines for each version of Internet Explorer you'd like to test. Open them in your emulator and you're good to go!


IE11 - Win7 [Running]
https://unravelingflexbox.com/
Unraveling Flexbox: The De...
UNRAVELING FLEXBOX
LEARN MORE
FREE STARTER COURSE
GET THE BOOK
Unraveling Flexbox
The Ultimate Guide to Building Modern Layouts in CSS
Landon Schropp
Become a Flexbox Ninja
With Unraveling Flexbox, you'll learn how to unlock the full potential of modern CSS to create responsive application layouts in hours, not days.
LEARN MORE
10:55 PM
6/20/2015

Internet Explorer running in a virtual machine

Chrome has a feature that lets you test your site for smaller screens. To use it, open up the Chrome Dev tools and click on the on the mobile icon. Here, you can adjust the screen size to see what your layout looks like on certain screens.



Unraveling Flexbox: The D
https://unravelingflexbox.com
Landon
Device Apple iPhone 5
320 × 568
Network No throttling
UA Mozilla/5.0 (iPhone; CPU iPhone O…
UNRAVELING FLEXBOX
Unraveling Flexbox
Become a Flexbox Ninja
With Unraveling Flexbox, you'll learn how to unlock the full potential of
Elements Network Sources Timeline Profiles Resources Audits Console Marionette
Styles Computed Event Listeners

```
<!DOCTYPE html>
▼<html lang="en" class="wf-proximanova-i3-active wf-proximanova-i6-active wf-proximanova-n3-active wf-proximanova-n4-active wf-proximanova-n6-active wf-proximanova-n7-active wf-proximanova-n8-active wf-active">
  ▶#shadow-root
   <script type="text/javascript" src="https://bam.nr-data.net/1/c3f4c60868?a=6321570&pl=1434865914663&v=632.…rpe%22:514%7D,%22navigation%22:%7B%22ty%22:1%7D%7D&jsonp=NREUM.setToken"></script>
   <script src="https://js-agent.newrelic.com/nr-632.min.js"></script>
   <script async src="//www.google-analytics.com/analytics.js"></script>
   <script type="text/javascript">window["_gaUserPrefs"] = { ioo : function() { return true; } }</script>
  ▶<head>…</head>
  ▼<body>
    ▶<section id="menu">…</section>
    ▶<main class>…</main>
     <!-- Scripts -->
```

```
element.style {
}
media="screen"                unravelingflexbox.com/
html, body {        application-a96…5ac1f744.css:1
    color: rgba(255,255,255,0.9);
    background-color: #3A4860;
    background-image:
        url(//d1x47zt8taxzyn.cloudfront.net/asse…
        c13dc0bd16135b1d31f54ccae210b087100f7065…
    background-size: 188px 178px;
    height: 100%;
    margin: ▶0;
  ~~-webkit-text-size-adjust: none;~~
```

Chrome's emulator

Mobile Browsers

The best way to test your code in mobile browsers is by running it on real phones. I'd recommend asking friends who are trading in their phones if you can have the old one. You can also purchase phones on CraigsList or eBay, but be careful that you don't get scammed or buy a stolen phone.

Chrome for Android uses Blink under the hood, which makes it mostly consistent with desktop Chrome. It's safe to only test the most recent version of this browser.

The Android stock browser is a different story. For flexbox, you need to test 4.1, 4.3 and 4.4 *independently*. This probably means using three separate devices, each with a different version of Android installed.

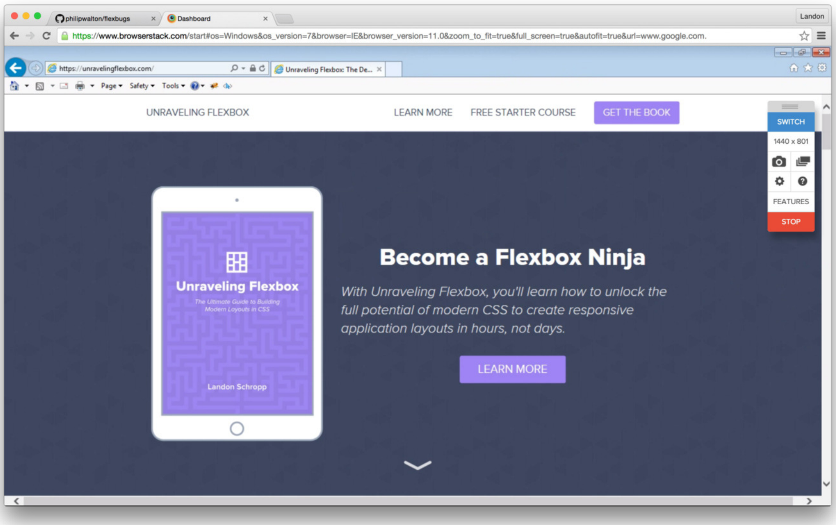
It's possible to test Android browsers using an emulator. However, it's a *huge pain* to set up, and the emulator is tediously slow. If you'd like to try, Google has published instructions for setting up an emulator. However, if you have access to an Android phone, that's a better option.

For iOS, it's possible to install an emulator on OS X, but I've seen several bugs on iPhones that couldn't be reproduced in the emulator. Your best bet is to pick up a used iPhone or iPod Touch and test on that.

BrowserStack

BrowserStack is my favorite way to test multiple browsers. It emulates desktop and mobile browsers, and makes it easy to switch between them. At $29 per month, it's somewhat expensive. However, if you're a professional web developer, it's an invaluable tool for making sure your site works perfectly across devices.

To use BrowserStack, simply choose the OS and browser you'd like to test and open your site!

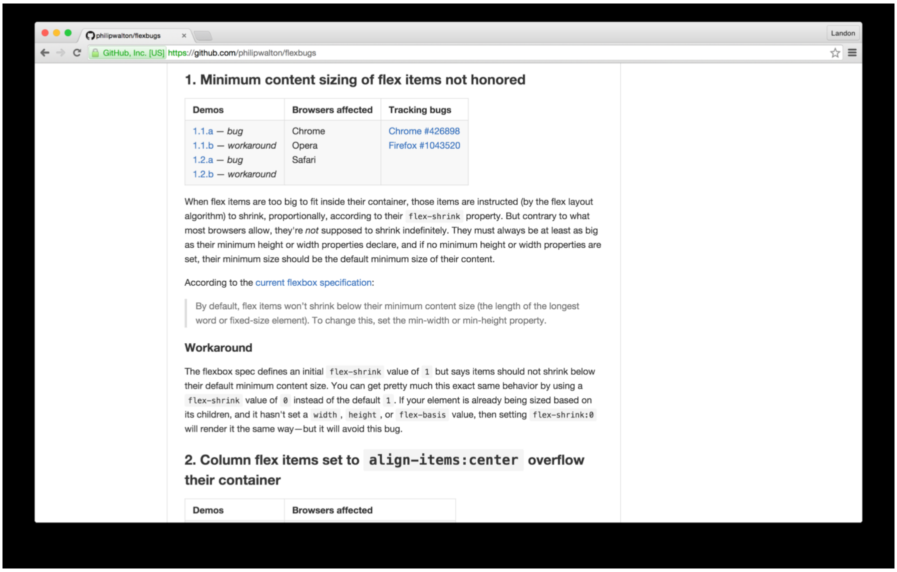

philipwalton/flexbugs
Dashboard
https://www.browserstack.com/start#os=Windows&os_version=7&browser=IE&browser_version=11.0&zoom_to_fit=true&full_screen=true&autofit=true&url=www.google.com.
https://unravelingflexbox.com/
Unraveling Flexbox: The De...
UNRAVELING FLEXBOX
LEARN MORE
FREE STARTER COURSE
GET THE BOOK
SWITCH
1440 x 801
FEATURES
STOP
Unraveling Flexbox
Landon Schropp
Become a Flexbox Ninja
With Unraveling Flexbox, you'll learn how to unlock the full potential of modern CSS to create responsive application layouts in hours, not days.
LEARN MORE

BrowserStack

What To Do When You Find a Flexbox Bug

You've tested your site across multiple browsers and found a bug. Now what? There are a few places to look for answers.

Flexbugs



1. Minimum content sizing of flex items not honored

Demos	Browsers affected	Tracking bugs
1.1.a — *bug* 1.1.b — *workaround* 1.2.a — *bug* 1.2.b — *workaround*	Chrome Opera Safari	Chrome #426898 Firefox #1043520

When flex items are too big to fit inside their container, those items are instructed (by the flex layout algorithm) to shrink, proportionally, according to their `flex-shrink` property. But contrary to what most browsers allow, they're *not* supposed to shrink indefinitely. They must always be at least as big as their minimum height or width properties declare, and if no minimum height or width properties are set, their minimum size should be the default minimum size of their content.

According to the current flexbox specification:

> By default, flex items won't shrink below their minimum content size (the length of the longest word or fixed-size element). To change this, set the min-width or min-height property.

Workaround

The flexbox spec defines an initial `flex-shrink` value of `1` but says items should not shrink below their default minimum content size. You can get pretty much this exact same behavior by using a `flex-shrink` value of `0` instead of the default `1`. If your element is already being sized based on its children, and it hasn't set a `width`, `height`, or `flex-basis` value, then setting `flex-shrink:0` will render it the same way—but it will avoid this bug.

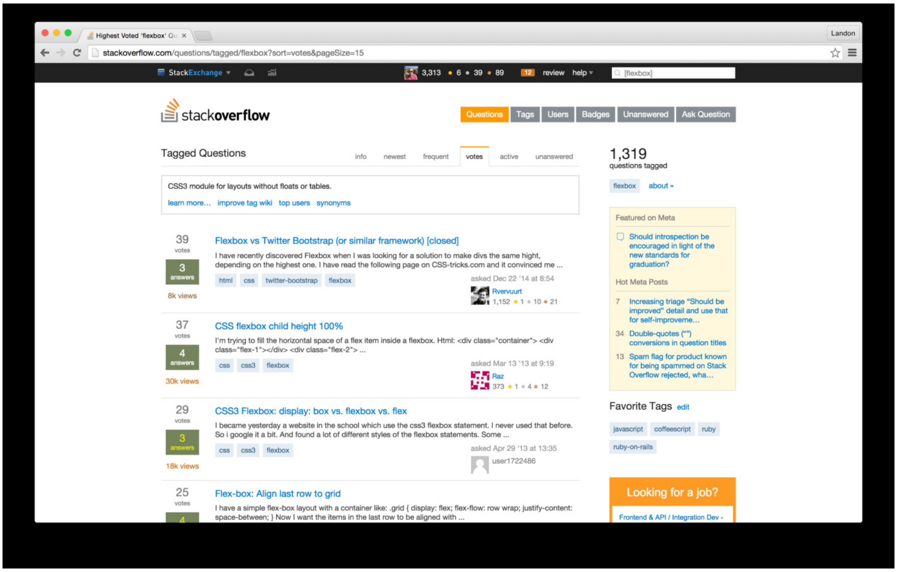
2. Column flex items set to `align-items:center` overflow their container

Demos	Browsers affected

Flexbugs

The first place you should check for flexbox bugs is Flexbugs. This fantastic resource lists many of the major flexbox bugs and workarounds for each one. The Flexbugs Issues page contains several bugs that haven't been added to the main document, so look there too.

StackOverflow



Tagged Questions

CSS3 module for layouts without floats or tables.

learn more… improve tag wiki top users synonyms

39 votes · 3 answers · 8k views

Flexbox vs Twitter Bootstrap (or similar framework) [closed]

I have recently discovered Flexbox when I was looking for a solution to make divs the same hight, depending on the highest one. I have read the following page on CSS-tricks.com and it convinced me ...

html css twitter-bootstrap flexbox

asked Dec 22 '14 at 8:54 Rvervuurt 1,152 • 1 • 10 • 21

37 votes · 4 answers · 30k views

CSS flexbox child height 100%

I'm trying to fill the horizontal space of a flex item inside a flexbox. Html: <div class="container"> <div class="flex-1"></div> <div class="flex-2"> ...

css css3 flexbox

asked Mar 13 '13 at 9:19 Raz 373 • 1 • 4 • 12

29 votes · 3 answers · 18k views

CSS3 Flexbox: display: box vs. flexbox vs. flex

I became yesterday a website in the school which use the css3 flexbox statement. I never used that before. So i google it a bit. And found a lot of different styles of the flexbox statements. Some ...

css css3 flexbox

asked Apr 29 '13 at 13:35 user1722486

25 votes · 4

Flex-box: Align last row to grid

I have a simple flex-box layout with a container like: .grid { display: flex; flex-flow: row wrap; justify-content: space-between; } Now I want the items in the last row to be aligned with ...

1,319 questions tagged

flexbox about »

Featured on Meta

Should introspection be encouraged in light of the new standards for graduation?

Hot Meta Posts

7 Increasing triage "Should be improved" detail and use that for self-improveme…

34 Double-quotes ("") conversions in question titles

13 Spam flag for product known for being spammed on Stack Overflow rejected, wha…

Favorite Tags edit

javascript coffeescript ruby ruby-on-rails

Looking for a job?

Frontend & API / Integration Dev -

StackOverflow

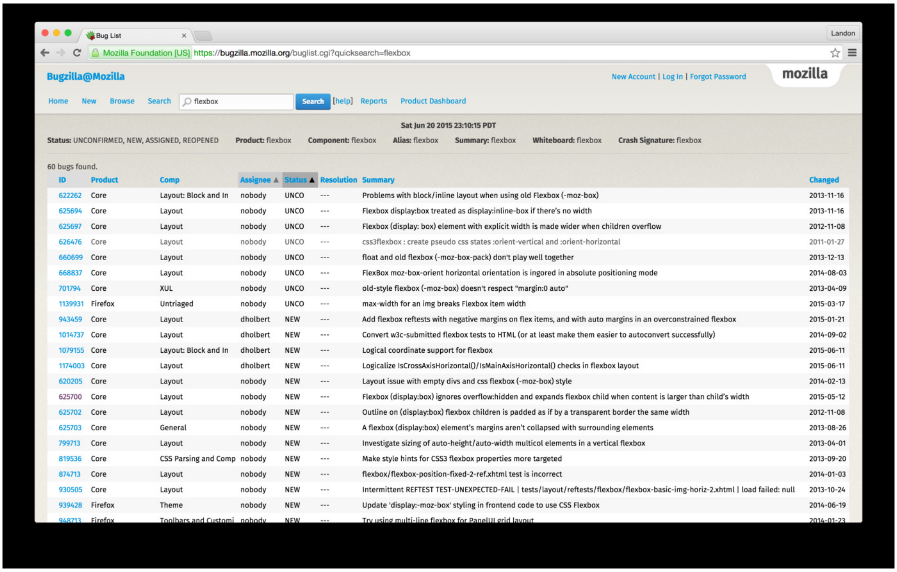
Unless you've been hiding under a rock, chances are good you've visited StackOverflow at least a few times. This question-and-answer site makes it easy to find answers about almost any development topic, including flexbox.

If you have a question you'd like to ask, there are a few important points to keep in mind.

- **Check to see if your question has already been answered.** Chances are pretty good you're not the first person to have your question. Look around and see if somebody's already asked it.

- **Try to create a *simple version* of your problem.** It's easier to solve your problem if you don't post a mountain of code with it. Plus, many times you will solve your own problem when trying to reproduce it.
- **Be specific.** Asking “Why doesn't setting display to flex for a <fieldset> work in Chrome?” is better than “Why doesn't this work?”.
- **Pay attention to spelling, grammar and formatting.** Your question is more likely to get answered if people can read it.

Browser Bug Tracker


Bug List

Mozilla Foundation [US] https://bugzilla.mozilla.org/buglist.cgi?quicksearch=flexbox

Bugzilla@Mozilla

New Account | Log In | Forgot Password

Home New Browse Search flexbox Search [help] Reports Product Dashboard

Sat Jun 20 2015 23:10:15 PDT

Status: UNCONFIRMED, NEW, ASSIGNED, REOPENED **Product:** flexbox **Component:** flexbox **Alias:** flexbox **Summary:** flexbox **Whiteboard:** flexbox **Crash Signature:** flexbox

60 bugs found.

ID	Product	Comp	Assignee	Status	Resolution	Summary	Changed
622262	Core	Layout: Block and In	nobody	UNCO	---	Problems with block/inline layout when using old Flexbox (-moz-box)	2013-11-16
625694	Core	Layout	nobody	UNCO	---	Flexbox display:box treated as display:inline-box if there's no width	2013-11-16
625697	Core	Layout	nobody	UNCO	---	Flexbox (display: box) element with explicit width is made wider when children overflow	2012-11-08
626476	Core	Layout	nobody	UNCO	---	css3flexbox : create pseudo css states :orient-vertical and :orient-horizontal	2011-01-27
660699	Core	Layout	nobody	UNCO	---	float and old flexbox (-moz-box-pack) don't play well together	2013-12-13
668837	Core	Layout	nobody	UNCO	---	FlexBox moz-box-orient horizontal orientation is ingored in absolute positioning mode	2014-08-03
701794	Core	XUL	nobody	UNCO	---	old-style flexbox (-moz-box) doesn't respect "margin:0 auto"	2013-04-09
1139931	Firefox	Untriaged	nobody	UNCO	---	max-width for an img breaks Flexbox item width	2015-03-17
943459	Core	Layout	dholbert	NEW	---	Add flexbox reftests with negative margins on flex items, and with auto margins in an overconstrained flexbox	2015-01-21
1014737	Core	Layout	dholbert	NEW	---	Convert w3c-submitted flexbox tests to HTML (or at least make them easier to autoconvert successfully)	2014-09-02
1079155	Core	Layout: Block and In	dholbert	NEW	---	Logical coordinate support for flexbox	2015-06-11
1174003	Core	Layout	dholbert	NEW	---	Logicalize IsCrossAxisHorizontal()/IsMainAxisHorizontal() checks in flexbox layout	2015-06-11
620205	Core	Layout	nobody	NEW	---	Layout issue with empty divs and css flexbox (-moz-box) style	2014-02-13
625700	Core	Layout	nobody	NEW	---	Flexbox (display:box) ignores overflow:hidden and expands flexbox child when content is larger than child's width	2015-05-12
625702	Core	Layout	nobody	NEW	---	Outline on (display:box) flexbox children is padded as if by a transparent border the same width	2012-11-08
625703	Core	General	nobody	NEW	---	A flexbox (display:box) element's margins aren't collapsed with surrounding elements	2013-08-26
799713	Core	Layout	nobody	NEW	---	Investigate sizing of auto-height/auto-width multicol elements in a vertical flexbox	2013-04-01
819536	Core	CSS Parsing and Comp	nobody	NEW	---	Make style hints for CSS3 flexbox properties more targeted	2013-09-20
874713	Core	Layout	nobody	NEW	---	flexbox/flexbox-position-fixed-2-ref.xhtml test is incorrect	2014-01-03
930505	Core	Layout	nobody	NEW	---	Intermittent REFTEST TEST-UNEXPECTED-FAIL | tests/layout/reftests/flexbox/flexbox-basic-img-horiz-2.xhtml | load failed: null	2013-10-24
939428	Firefox	Theme	nobody	NEW	---	Update 'display:-moz-box' styling in frontend code to use CSS Flexbox	2014-06-19
948713	Firefox	Toolbars and Customi	nobody	NEW	---	Try using multi-line flexbox for PanelUI grid layout	2014-01-23

Bugzilla

Every major browser has a bug tracker you can use to search for bugs.

- Microsoft Connect Feedback Center (Internet Explorer)
- Chromium Issues (Google Chrome)
- Bugzilla (Mozilla Firefox)
- Webkit Bugzilla (Apple Safari)

These tools are often incomplete and difficult to search through. However, if you take the time, it's possible to find answers you won't see anywhere else.

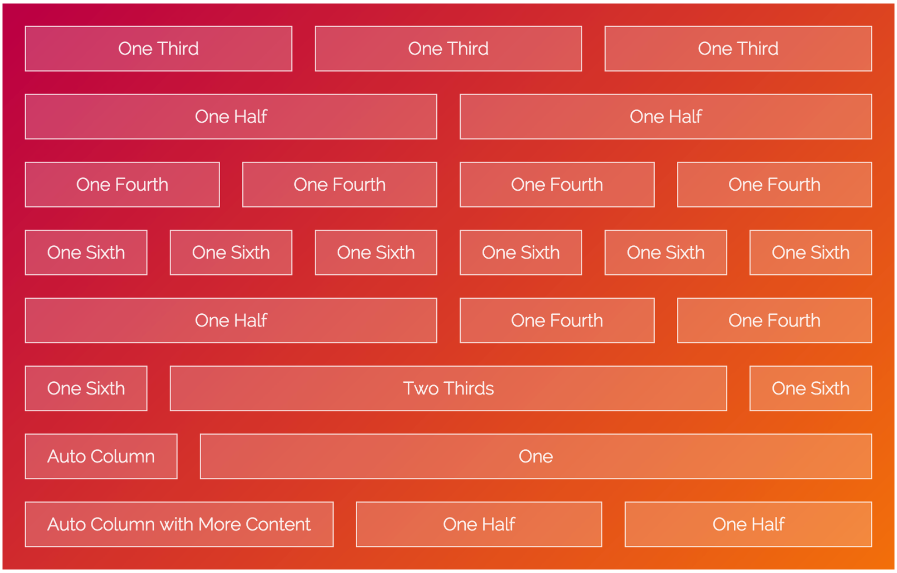
All Set

You've now learned how to determine which browsers to support for your project, how to test your site on those browsers, and where to look when you've found a problem.

If you're looking for a extra challenge, try testing some of the layouts you've previously built in older browsers. Then, see if you can use the progressive enhancement techniques from Chapter 10 to fix them!

CHAPTER 13

How to Write a Grid System



Grid systems drastically simplify and speed up web development. You append a few classes to your HTML, and suddenly you've cut out 90% of your layout CSS!

Grid systems abstract away the complexities of the underlying CSS. If you're working on a team of people who aren't as flexbox-savy as you are, you should buy them a copy of this book. ☺ Barring that, your teammates can use a grid system without having to deeply understand the technical details.

The major downside of grid systems is they pollute your DOM with stylistic classes. Many people believe that writing classes that are purely presentational is a misuse of HTML and CSS, a point I *wholeheartedly* agree with. However, I make an exception for grid systems because they're just so useful.

In this chapter, you'll learn how to build your very own flexbox grid system using Sass.

The Basics

If you've used other grid systems, such as 960gs or Bootstrap's grid, you're used to defining things in terms of a twelve-column grid. Since you're using flexbox, your grid system will cater to flexbox's proportional nature. It will consist of four major classes:

- `row-container`
- `column-container`
- `row`
- `column`

Row containers contain rows, and column containers contain columns. Easy enough to remember, right?

```
.row-container {
  display: flex;
  flex-direction: column;
}
```

```
.column-container {
  display: flex;
  flex-direction: row;
}
```

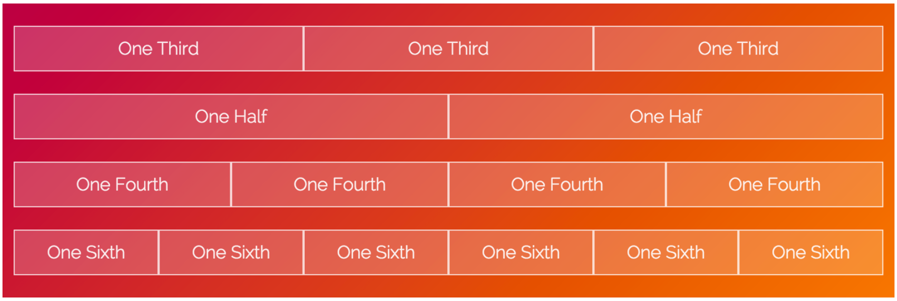
Wait, why `flex-direction` set to `column` for the `row-container`? Is that a typo? Nope! If you want `row-container` to contain rows, then the direction of the container needs to be `column`. It's a little tricky to wrap your head around, but I promise it's easier for other people using your grid system to think about it in those terms.

Next, you need to write the `row` and `column` classes. Conceptually, these are very different things. However, their implementation is *exactly the same*. The only reason to include both is because it makes it easier for others to use your grid system.

```
.row-container > .row,
.column-container > .column {
  flex: 1 1 0px;
}
```

With that, you can build equally-sized columns and rows without writing a single line of CSS!

```
<section class="column-container">
  <div class="column">One Third</div>
  <div class="column">One Third</div>
  <div class="column">One Third</div>
</section>

<section class="column-container">
  <div class="column">One Half</div>
  <div class="column">One Half</div>
</section>

...
```



Different-Sized Columns

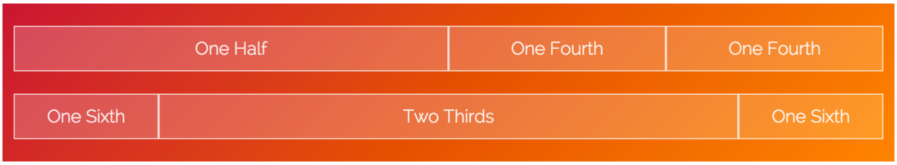
If that's all your grid system can do, it's not very useful. What you need is *different-sized* columns. You can accomplish that with some Sass magic.

Sass is a CSS preprocessor that enhances the language in several ways, including adding variables, control structures and mixins. Please be aware Sass code won't run in a regular browser; see Chapter 4 for details on how to set up CodePen or CodeKit to compile it.

```
$numbers: (one, two, three, four, five, six,
  seven, eight, nine, ten, eleven, twelve);

@for $index from 1 through length($numbers) {
  $number: nth($numbers, $index);

  .row-container > .#{ $number }-row,
  .column-container > .#{ $number }-column {
    flex: $index, $index, 0px;
  }
}
```

Just like that, you now have classes for `one-column`, `two-column`, `three-column` and so on.

```
<section class="column-container">
  <div class="two-column">One Half</div>
  <div class="column">One Fourth</div>
  <div class="column">One Fourth</div>
</section>

<section class="column-container">
  <div class="column">One Sixth</div>
  <div class="four-column">Two Thirds</div>
  <div class="column">One Sixth</div>
</section>
```

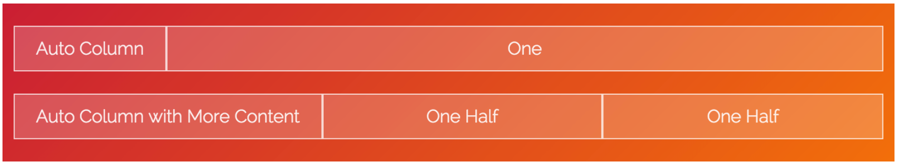


A Little Sugar on Top

There's one more *extremely* useful column to add: an `auto-column`. This column will size itself to the size of its content.

```
.row-container > .auto-row,
.column-container > .auto-column {
  flex: 0 0 auto;
}
```

```
<section class="column-container">
  <div class="auto-column gutter">
    <p>Auto Column</p>
  </div>
  <div class="column gutter">
    <p>One</p>
  </div>
</section>

<section class="column-container">
  <div class="auto-column gutter">
    <p>Auto Column with More Content</p>
  </div>
  <div class="column gutter">
    <p>One Half</p>
  </div>
  <div class="column gutter">
    <p>One Half</p>
  </div>
</section>
```



Gutters

Every major grid system supports gutters, the space between the items in the grid.

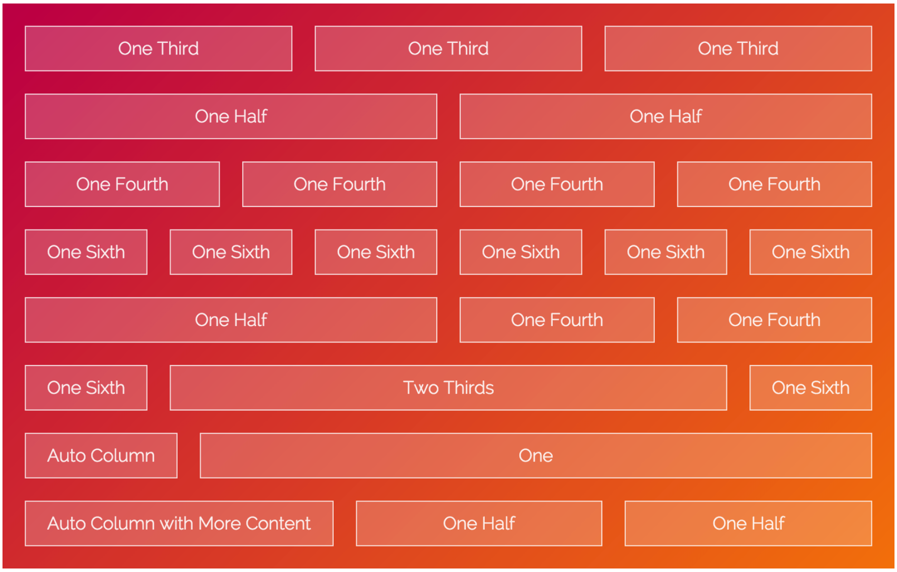
Remember back to chapter 2 when we used `flex-basis` to fix the sizes for the gutters? That's really hard to do in a reusable way for a grid system. Instead, you can apply them to the *children* of columns and rows. This is *much* simpler, but it does require a child element if you want a gutter.

```
$gutter-size: 20px !default;


.gutter > * {
  margin-left: $gutter-size / 2;
  margin-right: $gutter-size / 2;
}
```

The `!default` tells Sass that the gutter size is a default value, but can be overridden. This makes it easy to reuse the grid system across projects.

For the earlier examples, let's place the column contents into paragraphs so the margins can be applied.

```
<section class="column-container">
  <div class="column gutter">
    <p>One Third</p>
  </div>
  <div class="column gutter">
    <p>One Third</p>
  </div>
  <div class="column gutter">
    <p>One Third</p>
  </div>
</section>


...
```



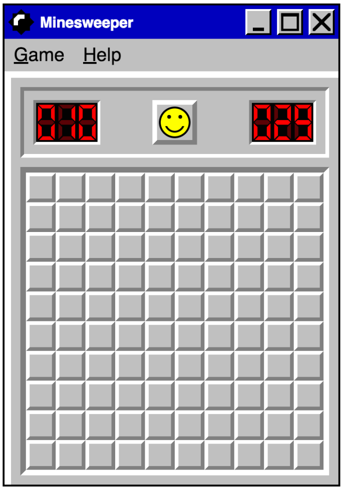
Just the Beginning

There's so much more you can do with your grid system. Play around with it and see what features you can add!

If you'd like to use a fully fleshed out flexbox grid system, try mine: Waffle. Feel free to submit pull requests!

CHAPTER 14

Minesweeper

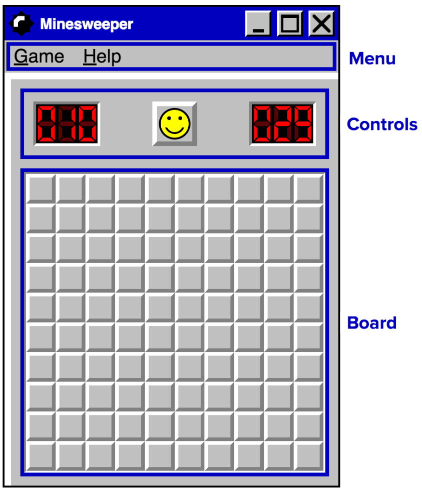


If you're like me, you've wasted hours of your life clearing squares in Minesweeper. This tricky game was first introduced in Windows 3.1, and has been a procrastination staple ever since. Well, it's time to put all of those lost hours to good use.

In this lesson, you're going to build the classic Minesweeper layout using flexbox. If you look closely at the layout, you'll notice pieces of it are similar to other layouts you've already built. Minesweeper requires most of the skills you've acquired in the course, so it's a good opportunity to try out what you've learned!

Set Up

In order to build the layout, you'll need to break it apart into three pieces: the menu bar, the game controls and the game board



It's fairly straightforward to represent this layout with HTML. You'll use a `<nav>` for the menu bar, and `<div>` containers for the controls and board. You'll place the controls and the board into a `<section>` to make it easier to add the border and padding around them.

```
<div class="minesweeper">
  <nav>
    ...
  </nav>
  <section class="game">
    <div class="controls">
      ...
    </div>
    <div class="board">
      ...
    </div>
  </section>
</div>
```

You'll also include a few styles that apply to the layout. I've left out the title bar for simplicity.

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

.minesweeper {
  font-size: 20px;
  font-family: sans-serif;
  background-color: #c0c0c0;
}
```

```
.game {
  padding: 10px;
  border-top: 7px solid white;
  border-left: 7px solid white;
}

button {
  padding: 0;
  border: none;
  background-color: transparent;
}
```

To make life a little easier, you'll use a Sass mixin to style the borders. If you're not comfortable with Sass, don't worry too much about this—it's simply a more concise way to write the `border` properties.

```
@mixin border($size, $top-left-color,
  $bottom-right-color) {

  border-top: $size solid $top-left-color;
  border-right: $size solid
    $bottom-right-color;
  border-bottom: $size solid
    $bottom-right-color;
  border-left: $size solid $top-left-color;
}
```

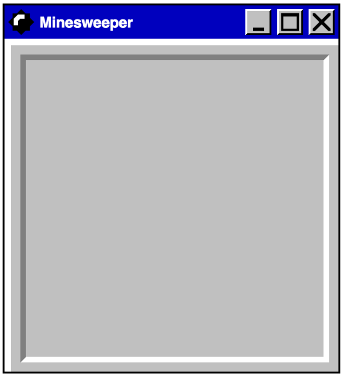
The Board

For the board, you want a ten-by-ten grid of buttons. You'll use the wrapping technique from Chapter 9 to achieve this.

```
<div class="board">
  <button class="space"></button>
  <button class="space"></button>
  <button class="space"></button>
  ...
  <button class="space"></button>
</div>
```

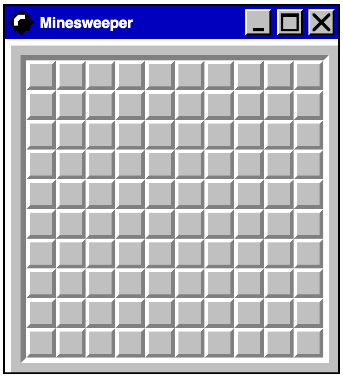
Before you can wrap the buttons, you need to set the size of the board. Each cell will be 32 pixels by 32 pixels, and the game has a 6 pixel border, so set the width and height to `10 × 32px + 2 × 6px`, which comes out to `332px`.

```
.board {
  @include border(6px, #808080, white);
  width: 332px;
  height: 332px;
}
```



To get those spaces to wrap, make the board a flex container, set its `flex-wrap` property and set the width and height of the spaces.

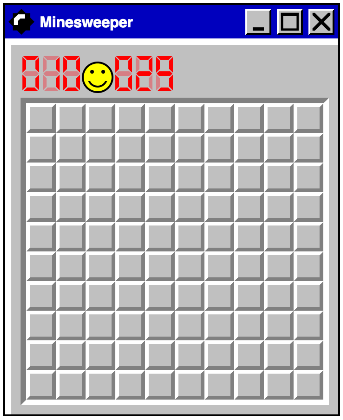
```
.board {
  @include border(6px, #808080, white);
  width: 332px;
  height: 332px;

  display: flex;
  flex-wrap: wrap;
}

.space {
  @include border(4px, white, #808080);
  width: 32px;
  height: 32px;
}
```



The Controls

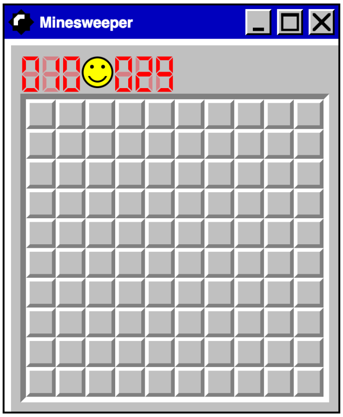
Next up are the game controls.

```
<div class="controls">
  <img class="number" src="numbers10.svg"
    alt="Score">
  <button type="button" class="reset">
    <img src="smiley.svg" alt="Smiley Face">
  </button>
  <img class="number" src="numbers29.svg"
    alt="Time">
</div>
```

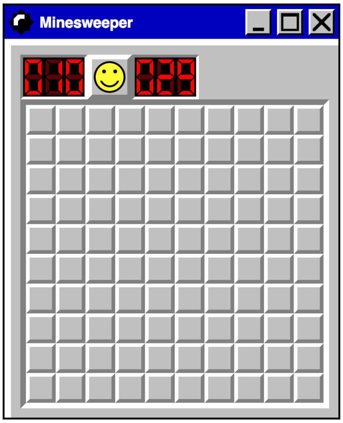


Make the `controls` a flex container, then style the number images and the reset button.

```
.controls {
  display: flex;
}
```

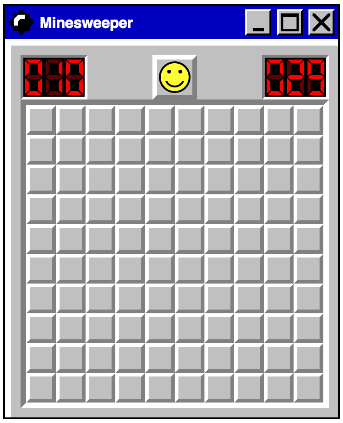


```
.number {
  @include border(3px, #808080, white);
  background-color: #030303;
}


.reset {
  @include border(5px, white, #808080);
  width: 48px;
  height: 48px;
}
```



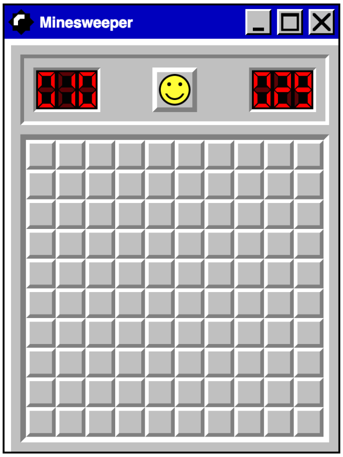
You can add space between the numbers and the reset button by using the `justify-content` property.

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



Finally, style the controls container.

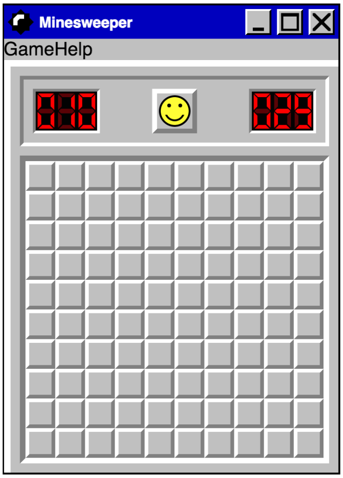
```
.controls {
  @include border(4px, #808080, white);

  display: flex;
  justify-content: space-between;

  margin-bottom: 10px;
  padding: 10px;
}
```



The Menu Bar

The last piece of the layout is the menu bar.

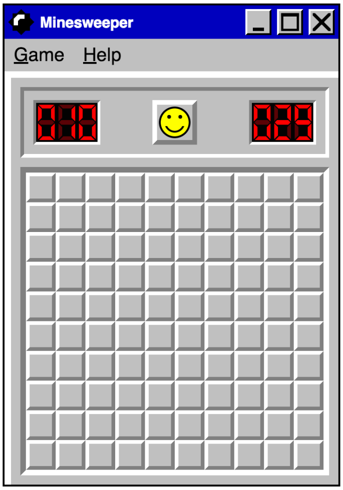
```
<nav>
  <button>Game</button>
  <button>Help</button>
</nav>
```



You *could* use flexbox to style the menu bar, but you really don't need it. The `<button>`'s default display of `inline-block` is enough to give the effect you want.

All you need to do is add a margin around the buttons. While you're at it, use the `first-letter` pseudo-element to underline the first letter of the menu items.

```
nav button {
  margin: 6px 10px;
}

nav button::first-letter {
  text-decoration: underline;
}
```



Game Over

That's it! You've taken the flexbox positioning, columns and wrapping skills you've learned from the previous lessons and used them to build something completely different. That's really what flexbox is—a collection of simple tools that can be combined to create amazing layouts.

This chapter is a good example of breaking down a hard problem into simpler pieces. Focusing on individual sections until the whole layout is complete is how most flexbox development is done in real projects.

For an extra JavaScript challenge, try turning the Minesweeper layout into a working game. Let me know if you do!

CONCLUSION

That's the end of the book! I hope you've enjoyed reading it as much as I have writing it.

You can now consider yourself a seasoned flexbox pro. You've learned all of the major flexbox properties, and you've seen flexbox used in real-world scenarios. Hopefully, you've also started to apply flexbox to your day-to-day web development.

If you'd like to learn a little more about flexbox, I'd recommend these resources:

- CSS-Tricks: A Complete Guide to Flexbox: This is a fantastic resource with explanation of all of the flexbox properties.
- Mozilla Developer Network (MDN): This is my go-to CSS reference. Type in any flexbox property and you'll see a list of all of the possible values and what they do.
- W3 Flexbox Specification: This is the canonical specification browsers use to implement flexbox.

If you want to keep up with what I'm working on, follow me on Twitter at @LandonSchropp.

Thanks again for reading, and happy flexboxing!