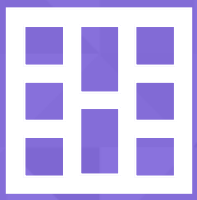


UNRAVELING FLEXBOX

The ultimate guide to building modern CSS layouts with flexbox

Landon Schropp

Sample Version 1.3

This is a sample of Unraveling Flexbox. The full version is available for purchase at unravelingflexbox.com.

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!

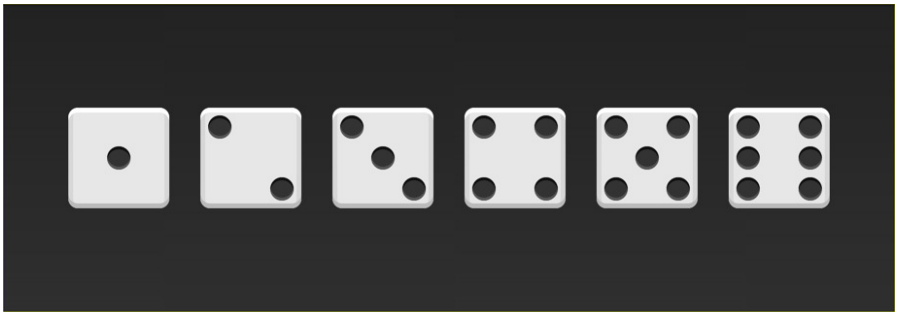
I'd also like to thank my beta readers, especially Joshua, Darrin, Andrew, Duc and Christine. You guys made a huge difference in the quality of this book, and I really appreciate it!

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!

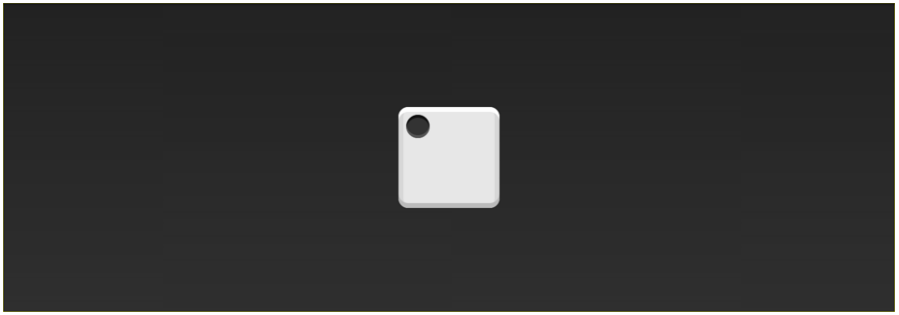
The First Face

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.

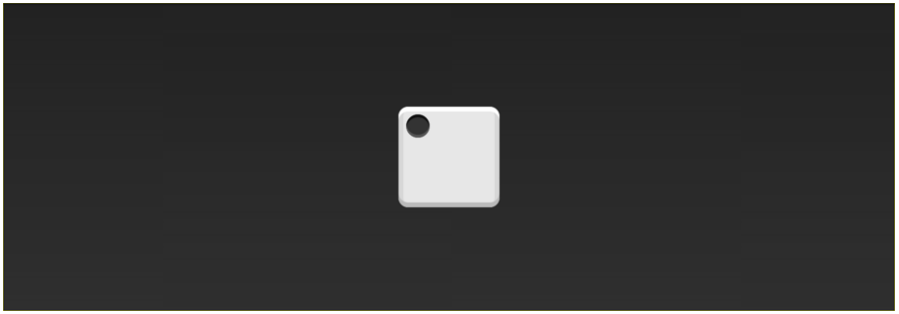
```
<div class="first-face">
  <span class="pip"></span>
</div>
```

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

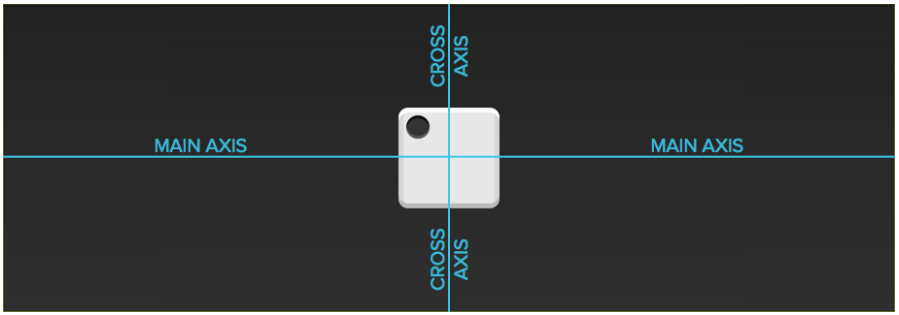


The first step is to tell the browser to make the face a flexbox container.

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



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

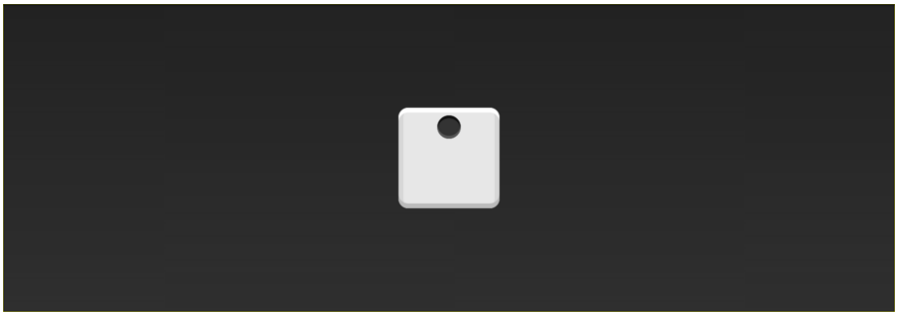


The flexbox container's main axis and cross axis

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.

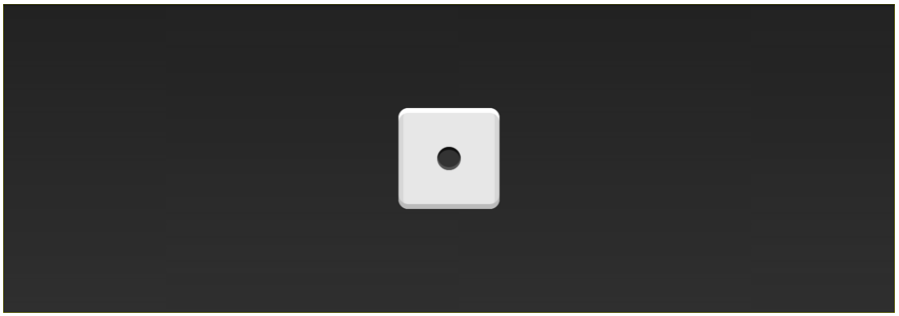
```
.first-face {
  display: flex;
  justify-content: center;
}
```



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!

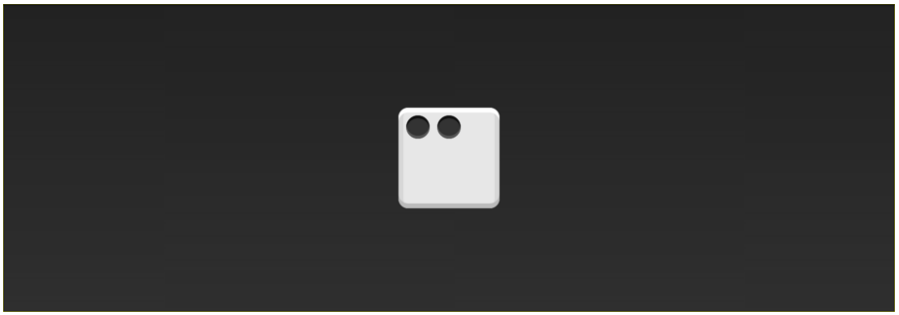
Getting Trickier

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>
```

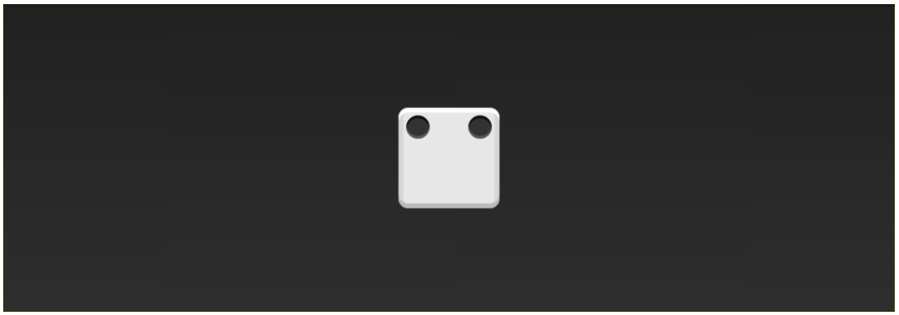
```
.second-face {
  display: flex;
}
```



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`.

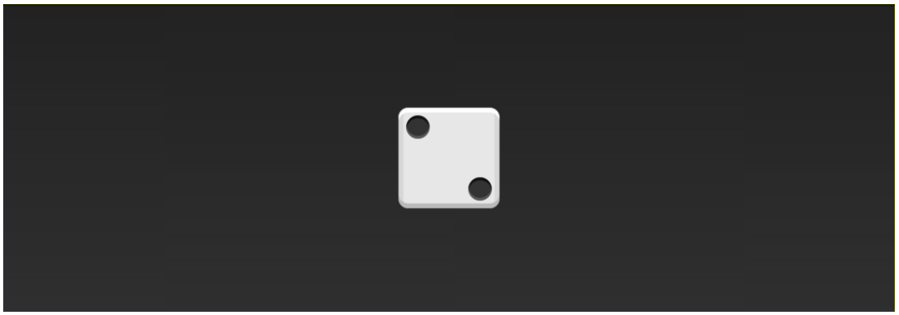
The `space-between` value evenly fills the space between flex items. Since there are only two pips, this pushes them away from each other.

```
.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!

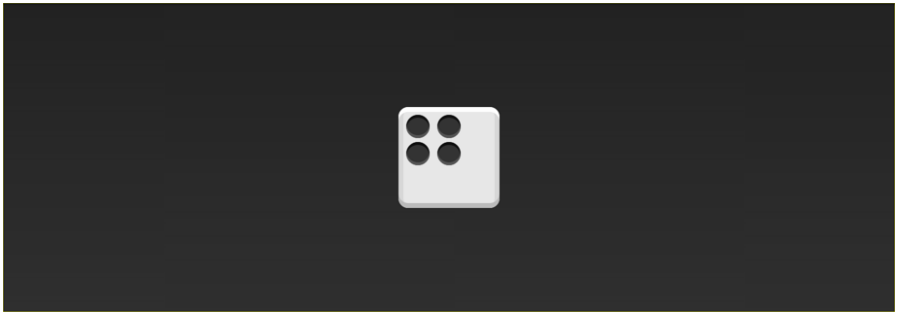
Horizontal and Vertical Nesting

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.

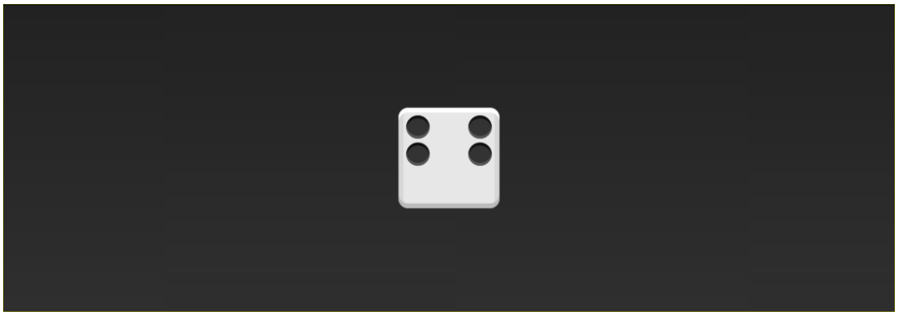
Unlike before, the markup will now include columns.

```
<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>
```



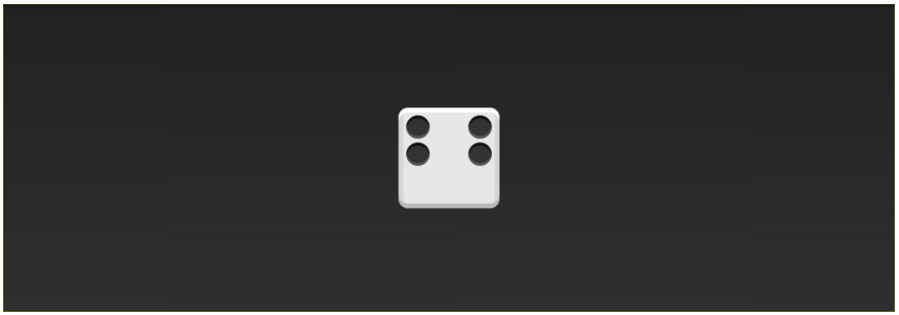
Since you want the two columns to be on opposite sides, go ahead and use `justify-content: space-between` like you did before.

```
.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.

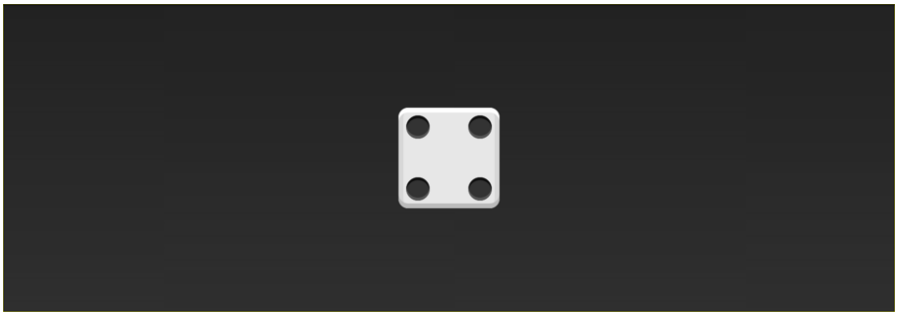
```
.fourth-face {
  display: flex;
  justify-content: space-between;
}

.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.

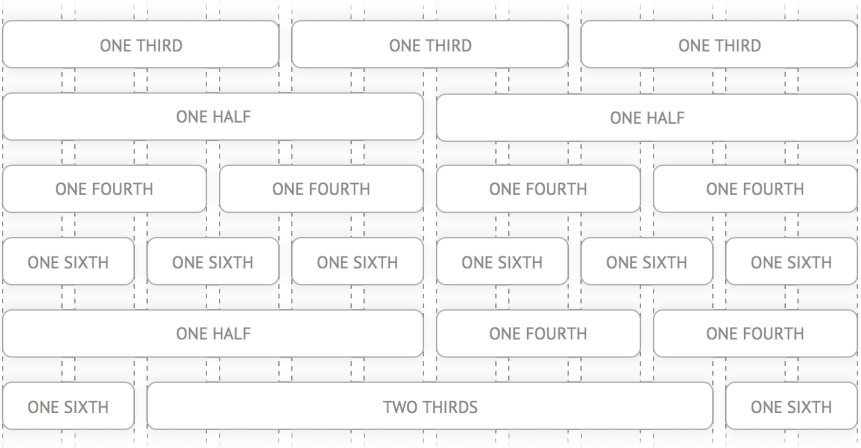
Wrapping Up

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

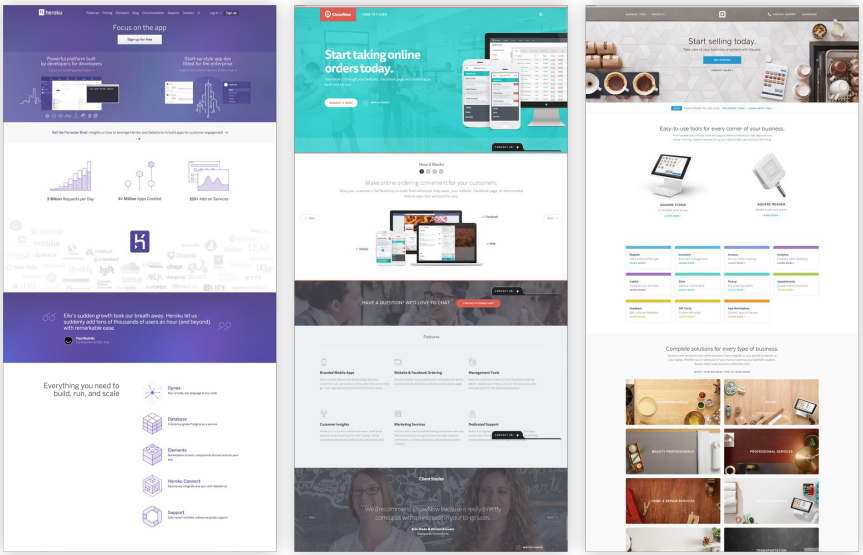
In a twelve-column layout, the page is broken apart into twelve invisible columns. These columns have small amounts of space between them, called *gutters*. The page is divided into rows, and the containers in the rows take up a certain number of columns.



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!

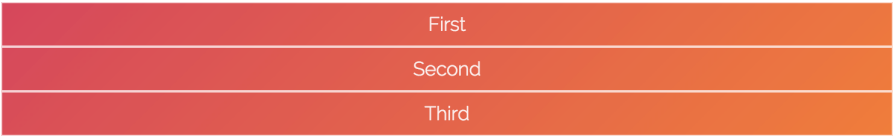


Examples of twelve-column layouts from Heroku, ChowNow and Square

Setting Up the Container

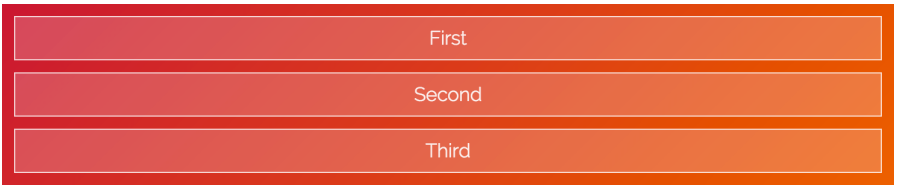
Let's say you want each of the `<div>` elements in the following HTML to take up a third of the `<section>`.

```
<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.

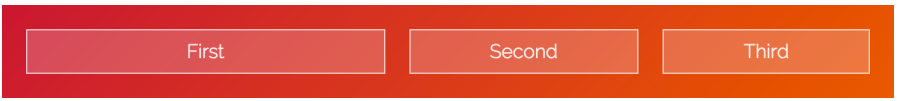
The `flex-grow` property tells flexbox how to grow the item to take up additional space, if necessary. `flex-shrink` tells flexbox how to shrink when necessary. Since we want the columns to behave the same while growing and shrinking, set both of these properties to `1`.

```
.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>
```

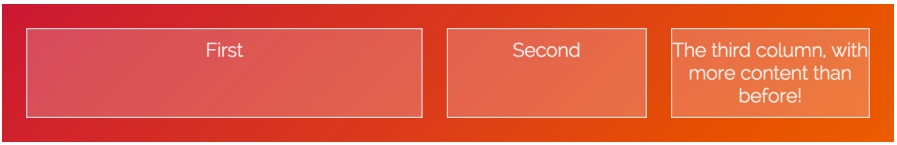
What's going on? Why is flexbox not flexing correctly?

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.

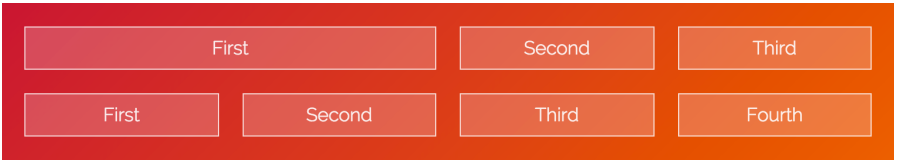
```
<section>
  <div class="column">First</div>
  <div class="column">Second</div>
  <div class="column">Third</div>
</section>
<section>
  <div class="column">First</div>
  <div class="column">Second</div>
  <div class="column">Third</div>
  <div class="column">Fourth</div>
</section>
```

```
.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:

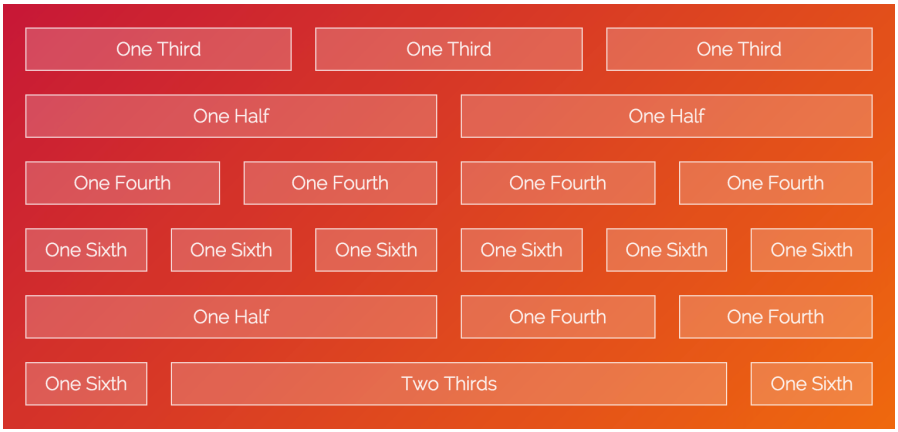
```
.column {
  flex: 1 1 0px;
}

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.

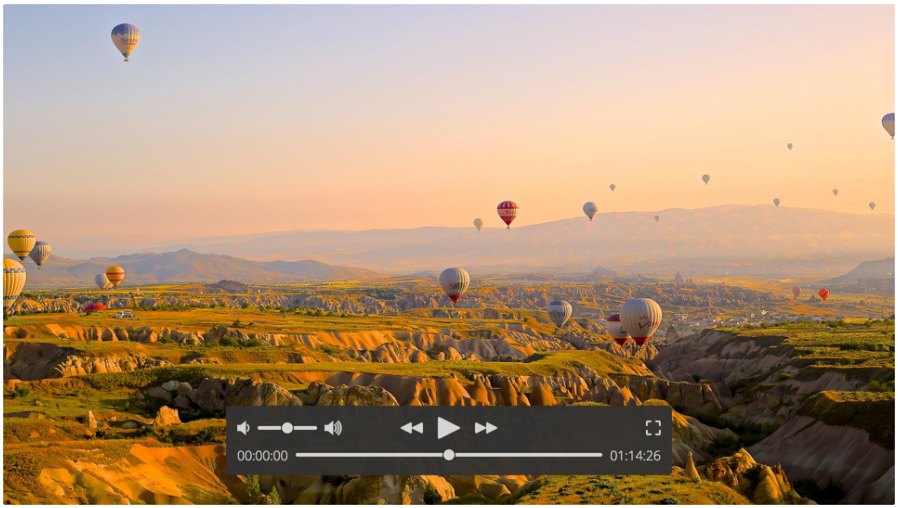
If you're looking for a challenge, try finishing off the entire grid. Here's what it looks like completed.



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

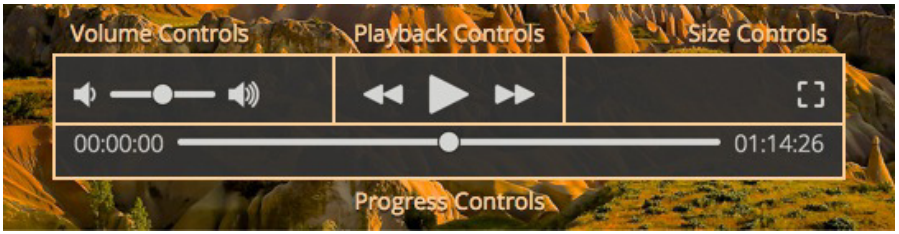


What's the best part about watching a movie? Is it the salty popcorn that coats your fingertips in hot, melted butter? How about the mountains of crunchy candy or the monolithic soda? Could it be the special effects and explosions, or the raw talent of the actors and actresses? Maybe it's the profound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

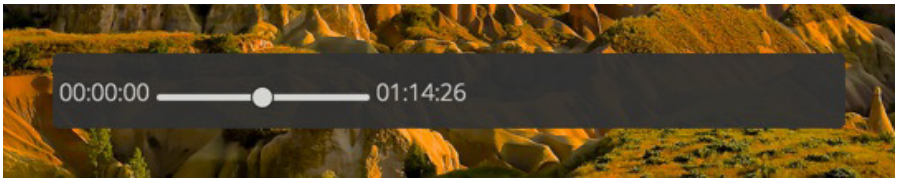
```
.controls-container {
  ...

  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;
}
```

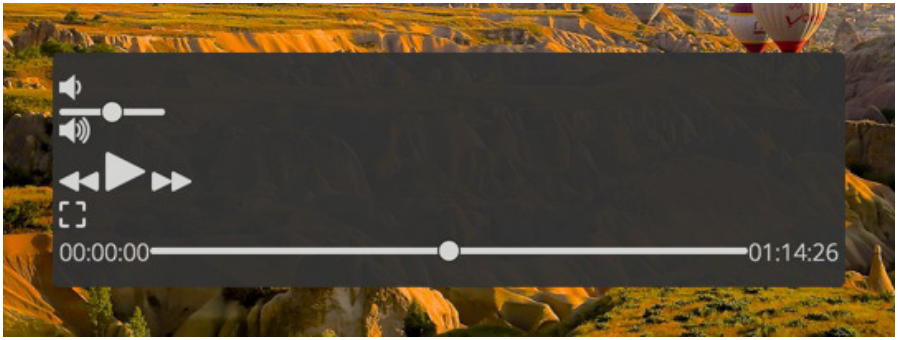
What's that `auto` value? Setting the `flex-basis` to `auto` tells flexbox to resize the container based upon the size of the content. In this case, the time elapsed and time remaining spans take up as much room as they need. Then, the progress controls container stretches to take up the rest of the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

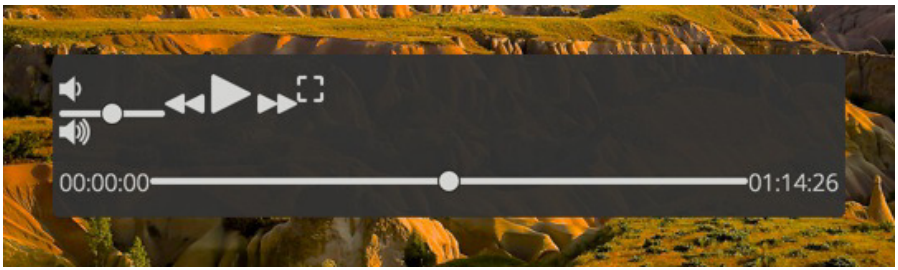
```
  <div class="playback-controls">
    <button>
      <img alt="Rewind" src="rewind.svg">
    </button>
    <button>
      <img alt="Play" src="play.svg">
    </button>
    <button>
      <img alt="Fast Forward"
        src="fast_forward.svg">
    </button>
  </div>
  <div class="size-controls">
    <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.

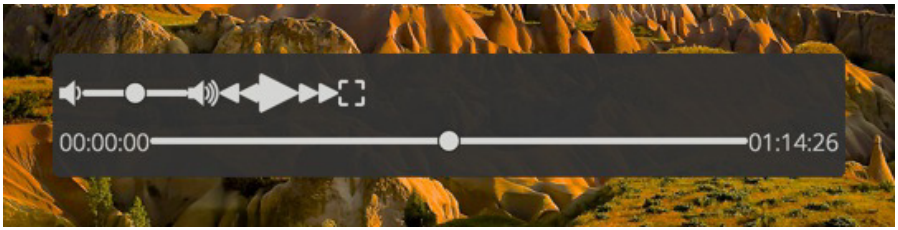
The first step in styling the top controls is to display them side by side. In order to do that, you need to set the top container's `display` to `flex`. Remember, the default value for `flex-direction` is `row`, so the container's contents will be displayed horizontally. While you're at it, add a little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

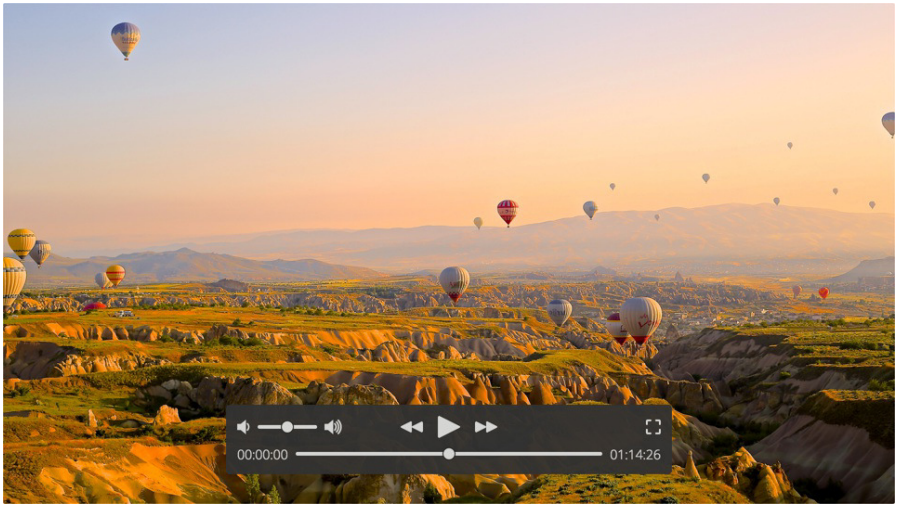
This works because the `flex-basis` of the playback controls is `auto`, so playback controls container is sized to the buttons it contains. The volume and size controls then evenly fill the remaining space.

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.

CONCLUSION

Thanks for Reading!

Thanks for reading this sample of Unraveling Flexbox. I hope you've enjoyed it! You've learned quite a bit about flexbox so far, but there's so much more in the eleven chapters you haven't seen yet.

In the full book, you'll learn how to:

- **Go beyond twelve-columns** to build layouts that were previously impossible.
- Build **responsive flexbox layouts** that work great in mobile, tablet and desktop browsers.
- **Support older browsers** while still using flexbox.
- Implement killer **flexbox form controls**.
- Create **video playback controls** with flexbox.
- Build layouts that **wrap across multiple lines**.
- Write your very own **flexbox grid system**.
- Test and fix your code across all **mobile and desktop browsers**.
- Style a flexible, **tiered pricing layout**.
- Use flexbox to **reorder elements** on a page.
- Use everything you've learned to build an **insanely awesome Minesweeper layout**.