TINY CSS Projects

Michael Gearon
Martine Dowden

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Tiny CSS Projects

Version 3

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Tiny CSS Projects*. To get the most benefit from this book, you'll be familiar with HTML and CSS and how to navigate through a basic web project. Working together we'll then build on your existing knowledge and take a deep dive into CSS.

One of the best parts of this book is how each chapter is its own project, with a particular focus on different CSS topics within each chapter. It allows you to learn through doing, rather than hypothetical concepts. We'd also encourage you to experiment with what you learn in your own projects, find inspiration from across the web and recreate it using your new found knowledge.

To get the most out of this book the main thing is to be creative and it's fine to not get it right the first time. CSS is a fun language to learn as it's visual, you can see the changes instantly and in most cases you can see when something goes wrong.

CSS has never been in a better place, in the last couple of years layout features like grid and flexbox, animations, transitions, media queries and much more have been introduced and expanded on, which we will look at. Whilst that's been happening old browsers like Internet Explorer have dropped away and browser support is more consistent.

We hope that you find this book useful, fun and interesting as you go through the projects and get a good understanding of where CSS is at now.

If you have any questions, comments, or suggestions, please share them in Manning's liveBook Discussion forum for our book:

—Michael Gearon and Martine Dowden

brief contents

1

CSS introduction

This chapter covers

- A brief overview of CSS
- Basic CSS styling
- Selecting HTML elements effectively

Cascading Style Sheets (CSS), are used to control the appearance of the elements of a web page. It uses syntactic structures called style rules to instruct the browser to select certain elements and apply visual qualities and effects to them.

Chapter 1 is a good place to start if you're new to CSS or in need of a refresher. We'll start with a brief history of CSS. Then swiftly moving on how to get started with CSS, looking at ways to link CSS with HTML.

Once we have our CSS up and running we'll look at the structure of CSS by creating a single static column article page with basic media components like headings, content and imagery and see how it all works together.

1.1 Overview

Håkon Wium Lie first proposed the idea of CSS in 1994, a few years after Tim Berners-Lee created HTML in 1990. CSS was introduced to separate styling from the content of the web page through the options of colors, layout and typography.

1.1.1 Separation of Concerns

This separation of content and presentation is based on the design principle of Separation of Concerns (SoC). The idea behind this is that a computer program or application should be broken up into individual, distinct sections, segregated by purpose.

The benefits of keeping good SoC include

- Decreased code duplication and therefore easier maintainability
- Extendability because it requires elements to focus on a single purpose
- Stability, because code becomes easier to maintain and to test

With this principle in mind, HTML serves as the structure and content of a web page, CSS is the presentation and Javascript provides additional functionality, which together form the web pages. Figure 1.1 displays a diagram of this process.

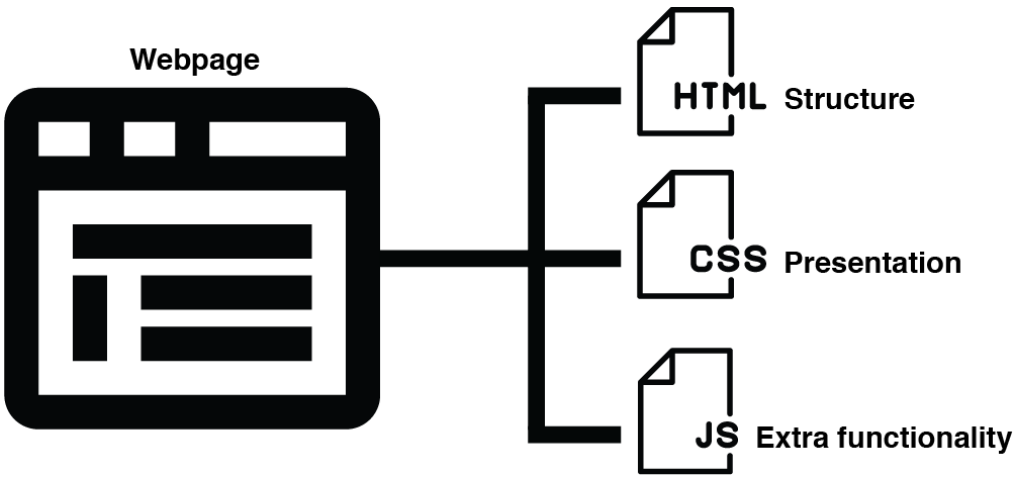

Figure 1.1 A breakdown of a webpage

Since the introduction of smartphones in the mid-2000s, the web then expanded into responsive and adaptive designs, and mobile websites which tended to have less features than the desktop experience. These mobile websites typically sat on a subdomain like m.example.com. There are pros and cons to whether we should create a responsive or mobile specific website. In general responsive and adaptive seems to be the way forward - especially as the CSS expands, giving us more ability to apply CSS targeted to window sizes and media types.

Since the first announcement of CSS in 1994, there have been three overall releases of CSS:

- CSS1 published in 1996
- CSS2 published in 1998
- CSS3 published in 1999

After 1999, the release strategy was changed to allow for faster, more frequent release of new features. Instead it is now divided into modules, with numbered levels starting at one going all the way up to level five. A CSS level one module would be something that is brand new to CSS such as a property that hasn't existed until now. Some examples are the masking module or the multi-column module, which we will look at later in this book. Modules which

have gone through a few versions and are now level 5 are: media queries, color, fonts, and cascading and inheritance modules.

The benefit of breaking CSS into modules is that each part can move independently rather than large sweeping changes. However, there have been some discussions that there is a need for someone to declare we are now at the stage of CSS4, even if it's just to acknowledge that CSS has changed a lot since 1999.

1.1.2 What is CSS?

CSS is a domain-specific language (DSL). DSLs are languages that are specialized and created to solve a specific problem. They are generally less complex than General Purpose Languages (GPL) like Java or C#. CSS's specific purpose is to style web content. Languages such as SQL, HTML and XPath are also DSLs.

CSS is also a declarative programming language. This means that the code tells the browser what needs to be done rather than how to do it. For example, our code says that we want this heading to be red in color, and then the browser determines how it is going to do it.

This is useful because, if we want to increase the line height of a paragraph to improve the reading experience, it is up to the browser to determine the new layout, sizing and formatting of that line height, which reduces the effort for the developer.

CSS has come a long way since 1994. There are now ways to animate and transition elements, create motion paths to animate SVG graphics, and conditionally apply styles based on viewport size. This type of functionality used to be only possible through JavaScript or Flash (now retired).

In the past, design choices such as the use of transparency, rounded borders, masking and blending, were doable but required unconventional CSS techniques and "hacks". As CSS evolved, properties were added to replace these hacks with standard, documented features. It could be said that CSS is in a golden age. With the continual development of the language, opportunities for new and creative experiences are virtually endless.

1.2 Getting started with CSS by creating an article layout

In our first project we will explore a common use case across the web, creating a single column article. This chapter will focus on how to link CSS to HTML, and then explore the selectors we can use to style our HTML.

The first thing we need to understand is how to tie our CSS to our HTML and how to select an element, then we can worry about what properties and values we want to apply. Let's go over some fundamentals.

1.2.1 Setup

If you are new to coding then there are a few, often free, tools you can use for these projects. You have the option of coding online or you can do it on your computer, using a code editor such as:

- Sublime
- Atom
- Visual Studio Code
- Basic text editor such as
- TextEdit (Mac)
- Notepad (Windows)
- Gedit (Linux)

You can also use a free online development editor like CodePen (https://codepen.io). Online development editors are a great way to test out ideas and provide quick and easy access for frontend projects.

Once you either have a code editor installed on your computer or have chosen an online editor and created an account, we will need to get the starter code for the chapter.

A code repository has been created in Github (https://github.com/michaelgearon/Tiny-CSS-Projects) containing all the code you will need to follow along with each chapter. Figure 1.2 shows a screenshot of the repository.

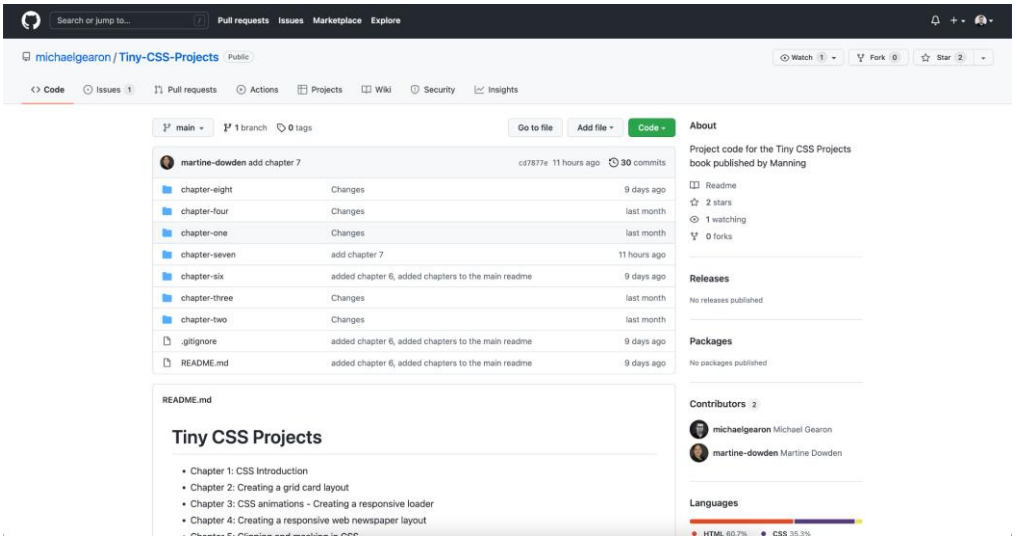

Figure 1.2 Tiny-CSS-Projects repository in Github

The code is organized into folders by chapter. Inside each chapter folder are 2 versions of the code:

- before: contains the starter code for the project. This is the version you will want if you are coding along with the chapter
- after: the completed project, as it is at the end of the chapter with the presented CSS applied

Download, (or if you are familiar with git, you can clone), the project selecting the "Code" drop down at the top of the screen.

Assuming you are coding along with the chapter, grab the files from the "before" folder in `chapter-01` and copy them to your project folder or into your pen. You should see an HTML file with some starter code, and an empty CSS file. If you open the HTML file in a web browser or copy the contents into CodePen you will see the content is currently unstyled except for the defaults provided by your browser as seen in Figure 1.3. We are now ready to start styling the content using CSS.

Title of our article (heading 1)

Posted on May 16 by Lisa.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque tincidunt dapibus eleifend. Nam eu urna ipsum. Etiam consequat ac dolor et dapibus. Duis eros arcu, interdum eu volutpat ac, lacinia a tortor. Vivamus justo tortor, porttitor in arcu nec, pretium viverra ipsum. Nam sit amet nibh magna. Sed ut imperdiet orci, id finibus justo. Maecenas magna mauris, tempor nec tempor id, aliquam et nibh. Nunc elementum ut purus id eleifend. Phasellus pulvinar dui orci, sed eleifend magna ullamcorper sit amet. Proin iaculis lacus congue aliquam sodales.

1. List item 1
    - Nested item 1
    - Nested item 2
2. List item 2
3. List item 3
4. List item 4

Curabitur id augue nulla. Aliquam purus urna, aliquam eu ornare id, maximus et tellus. Aliquam eleifend sem vitae urna blandit, non bibendum tellus dignissim. Aliquam imperdiet imperdiet sapien sit amet consectetur. Nam convallis turpis felis, sedvulputate lacus eleifend a. Mauris pharetra imperdiet lacinia. Sed sit amet feugiat lectus, in consectetur magna. Vestibulum accumsan porta enim at ultricies. Vestibulum vitae massa quis massa dignissim imperdiet.

> Nunc eleifend nulla lobortis porta rhoncus. Vivamus feugiat, sem vitae feugiat aliquam, orci nulla venenatis libero, vitae rhoncus nibh neque ac velit.

Etiam tempor vulputate varius. Duis at metus ut eros ultrices facilisis. Donec ut est finibus, egestas nisl eu, placerat neque. Pellentesque cursus, turpis nec sollicitudin sodales, nis tellus ultrices lectus, nec facilisis purus neque vitae diam. Nunc eleifend nulla lobortis porta rhoncus. Vivamus feugiat, sem vitae feugiat aliquam, orci nulla venenatis libero, vitae rhoncus nibh neque ac velit. Donec non fringilla magna. Vivamus eleifend ligula libero, fermentum imperdiet arcu viverra in. Vivamus pellentesque odio interdum mauris aliquam scelerisque.

Heading 2

In ac euismod tortor. Vivamus vitae velit efficitur, mattis turpis quis, tincidunt elit. In eleifend in dolor id aliquet. Vivamus pellentesque erat a magna ultricies rhoncus. Vestibulun at mattis purus, non lobortis risus. Mauris porta ullamcorper mollis. Sed et placerat nisi, quis porttitor lacus. Curabitur sagittis nisl egestas ipsum tristique, eu semper erat gravida. Vestibulum sagittis quam sit amet tristique ultricies.

In id lobortis leo. Nullam commodo tortor eu neque tempus accumsan. Vivamus molestie, felis consequat consequat iaculis, justo massa porttitor tellus, ac suscipit urna erat eu erat. Nunc malesuada eleifend erat nec pharetra. Sed eu magna iaculis, elementum dui ac, sagittis augue. Nam sit amet risus dapibus massa rutrum faucibus. Sed rhoncus finibus magna, vel tristique sem bibendum nec.

Heading 3

Mauris sit amet tempor ex. Morbi eu semper velit. Nullam hendrerit urna pellentesque, interdum lectus volutpat, gravida odio. Sed vulputate eget ante vel vehicula. Curabitur ac velit sed magna malesuada hendrerit. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Ut volutpat nisi purus. Morbi venenatis fermentum commodo. Nam accumsan mollis neque non interdum. Aenean cursus metus ac est gravida, placerat interdum justo pellentesque. Duis nec scelerisque lacus, elementum tincidunt est. Maecenas et leo justo. Nam porta risus porttitor vulputate laoreet. Nulla sodales sagittis nulla, non viverra erat consectetur et.

Figure 1.3 Starter HTML for our Article

1.3 Adding CSS to our HTML

When styling with CSS there are 3 ways to apply CSS to our HTML:

- Inline
- Internal
- External

1.3.1 Inline CSS

We can inline the CSS by adding a `style` attribute to an element. This method has us add the CSS to the element directly in the HTML.

Attribute tags are always specified in the start tag and typically consist of the name of the attribute, in this case: `style`. The attribute is then followed by an equals sign (`=`) and its value in quotes. All of the CSS goes inside of the opening and closing quotation marks.

As an example, let's set the color of our `heading` to `crimson`:

```
<h1 style="color: crimson"> Title of our article (heading 1) </h1>
```

If we save our HTML and view it in a browser, we will see it is now crimson. If we are using a code editor rather than a web client (codepen), we will need to refresh the browser page to view our changes. Figure 1. 3 shows the output. We notice that the only element affected is the `<h1>` on which we applied the styles.

Title of our article (heading 1)

Posted on May 16 by Lisa.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque tincidunt dapibus eleifend. Nam eu urna ipsum. Etiam consequat ac dolor et dapibus. Duis eros arcu, interdum eu volutpat ac, lacinia a tortor. Vivamus justo tortor, porttitor in arcu nec, pretium viverra ipsum. Nam sit amet nibh magna. Sed ut imperdiet orci, id finibus justo. Maecenas magna mauris, tempor nec tempor id, aliquam et nibh. Nunc elementum ut purus id eleifend. Phasellus pulvinar dui orci, sed eleifend magna ullamcorper sit amet. Proin iaculis lacus congue aliquam sodales.

1. List item 1
    - Nested item 1
    - Nested item 2
2. List item 2
3. List item 3
4. List item 4

Curabitur id augue nulla. Aliquam purus urna, aliquam eu ornare id, maximus et tellus. Aliquam eleifend sem vitae urna blandit, non bibendum tellus dignissim. Aliquam imperdiet imperdiet sapien sit amet consectetur. Nam convallis turpis felis, sedvulputate lacus eleifend a. Mauris pharetra imperdiet lacinia. Sed sit amet feugiat lectus, in consectetur magna. Vestibulum accumsan porta enim at ultricies. Vestibulum vitae massa quis massa dignissim imperdiet.

> Nunc eleifend nulla lobortis porta rhoncus. Vivamus feugiat, sem vitae feugiat aliquam, orci nulla venenatis libero, vitae rhoncus nibh neque ac velit.

Etiam tempor vulputate varius. Duis at metus ut eros ultrices facilisis. Donec ut est finibus, egestas nisl eu, placerat neque. Pellentesque cursus, turpis nec sollicitudin sodales, nisi tellus ultrices lectus, nec facilisis purus neque vitae diam. Nunc eleifend nulla lobortis porta rhoncus. Vivamus feugiat, sem vitae feugiat aliquam, orci nulla venenatis libero, vitae rhoncus nibh neque ac velit. Donec non fringilla magna. Vivamus eleifend ligula libero, fermentum imperdiet arcu viverra in. Vivamus pellentesque odio interdum mauris aliquam scelerisque.

Heading 2

In ac euismod tortor. Vivamus vitae velit efficitur, mattis turpis quis, tincidunt elit. In eleifend in dolor id aliquet. Vivamus pellentesque erat a magna ultricies rhoncus. Vestibulum at mattis purus, non lobortis risus. Mauris porta ullamcorper mollis. Sed et placerat nisi, quis porttitor lacus. Curabitur sagittis nisl egestas ipsum tristique, eu semper erat gravida.

Figure 1.4 Crimson colored header

The benefit (and downside) of inline CSS is that it takes the highest specificity in CSS, which we'll look at in more detail shortly.

Another major downsides to inline CSS is that it can quickly become unmanageable. For example, imagine we had 20 paragraphs within an HTML document. We would need to apply

the same style attributes with the same CSS properties 20 times to make sure all of our paragraphs look the same.

This scenario exposes 2 issues:

- Our concerns are no longer separated. Our HTML (responsible for the content) and CSS (responsible for styling) are now in the same place and tightly coupled.
- We are repeating the same code in a large number of places, making it extremely difficult to maintain the code and to keep our styles consistent.

The benefit to inline CSS is on page load performance. The browser will first load the HTML file and then any other files it needs to render the page. With the CSS being in the HTML file right away, the browser does not need to wait for it to load from a separate location.

Let's undo the style we just added to the `h1` and look at a different technique with the same benefits as inline but less drawbacks.

1.3.2 Internal CSS

To resolve the issue of repeating code, we can instead add our CSS within an internal `style` tag. It is recommended but not required that we add our `style` tag between the opening and closing `head` tags. To color all of our heading elements crimson we can use this snippet of code (Listing 1.1):

Listing 1.1

```
<!DOCTYPE html>
<html lang="en">
 <head>
   ...
  <style>
   h1, h2, h3, h4, h5, h6 {
    color: crimson;
   }
  </style>
 </head>
 <body>
  ...
 </body>
</html>
```

The benefit of this approach is that we are now grouping all of our CSS together and it will be applied to the whole HTML document. So in our example all `headings` (`h1, h2, h3, h4, h5, h6`) within that web page will be in crimson as we can observe in Figure 1.5.

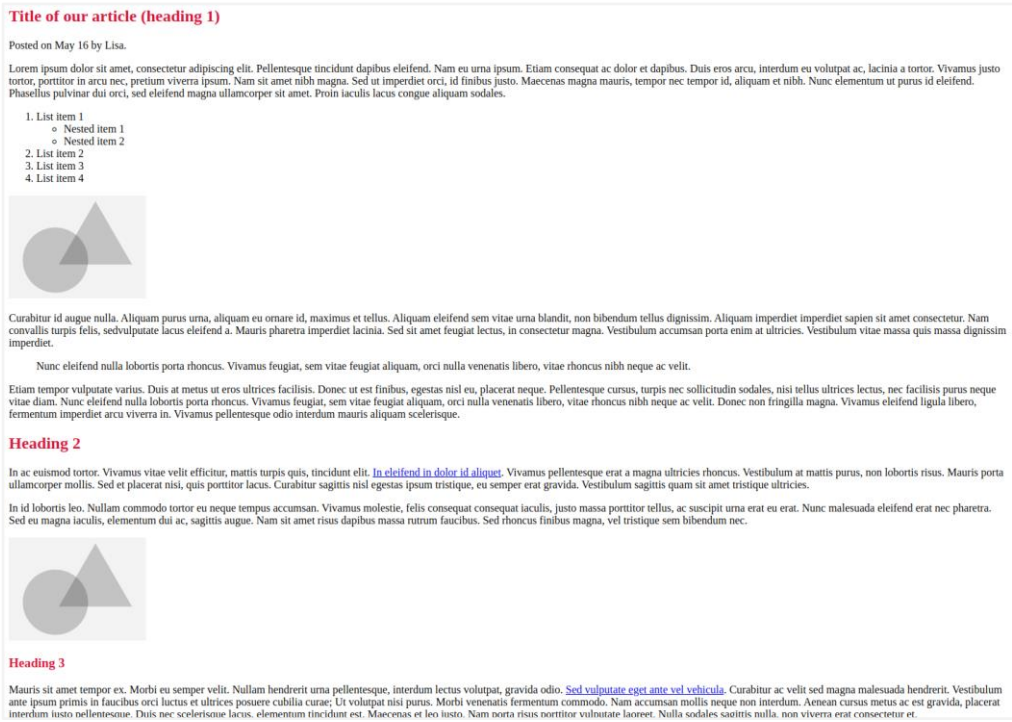

Figure 1.5 Styles applied to all headings

We also see a difference in how the internal CSS is written compared to inline CSS. When writing internal CSS, we create what are known as rules. They are composed of the following parts (Figure 1.6):

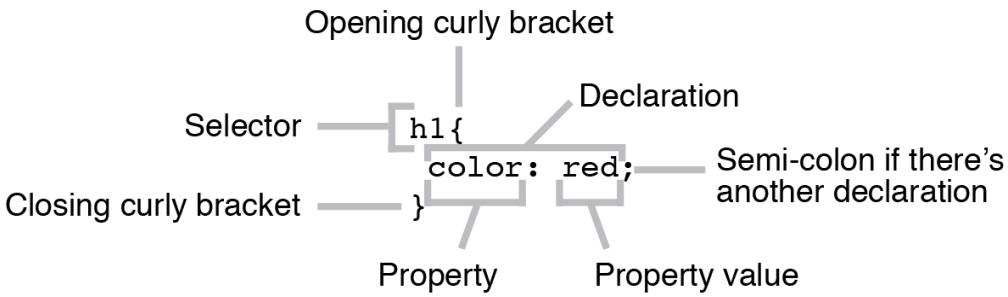

Figure 1.6 An example of a CSS rule

The part of the rule that defines which elements to apply the styles to is called the selector. The rule in Figure 1.6 will be applied to all `h1` elements, its selector is `h1`. To apply multiple

selectors, we write them as a comma delimited list before the opening curly bracket. To select all `h1` and `h2` elements for example, we would write `h1,h2 { … }`.

The declaration is made up of the property, in this case `color`, followed by the property value (`red`). The declaration defines how the element selected will be styled.

Both properties and values must be written in American English. Spelling variations such as "colour" or "capitalise" are not supported and will not be recognized by the browser.

When a browser comes across invalid CSS, it will ignore it. If a rule has an invalid declaration inside of it, valid declarations will still be applied, only those that are invalid will be ignored.

Internal CSS works well for one-off web pages where the styles are specific to that page. It groups CSS nicely, allowing us to write rules that are applied across elements, preventing us from having to copy and paste the same styles in multiple places. It also has the same performance benefits as inline styles in that the browser immediately has access to the CSS rather than having to wait for it to be fetched from a different location.

The downside of having our CSS within our HTML document is that the CSS will only work for that document. So if our website has multiple pages, which is often the case, we would need to copy that CSS into each HTML document. This will quickly become unmaintainable especially for large applications such as blogs or ecommerce websites because any style change would have to be applied to all pages.

Let's undo our changes to our project one last time, and look at a third technique.

1.3.3 External CSS

Like internal CSS, the external CSS approach keeps our CSS grouped together however it places the CSS in a separate `.css` file. By separating our HTML and CSS, we can effectively separate our concerns: content and style.

We link the stylesheet to the HTML by using the `<link>` HTML tag. The link element needs two attributes for stylesheets: the `rel` attribute, which describes the relationship between the HTML document and the thing being linked to; and the `href` attribute which stands for **h**ypertext **ref**erence and indicates where to find the document that we want to include. Listing 1.2 show how we link our stylesheet to our HTML for our project.

Listing 1.2

```
<!DOCTYPE html>
<html>
 <head>
 <link rel="stylesheet" href="styles.css">
 </head>
 <body>
  <h1>Inline CSS</h1>
 </body>
```

Most of the time this is the approach we will see across the web and is the approach we will use throughout this book. The benefit of external stylesheets is that our CSS is in one single document which can be modified once and whose changes will apply across all of our HTML pages. The downside to this approach is that it does mean it will take an extra request

from the browser to retrieve that document, losing the performance benefit provided by putting the CSS directly inside of the HTML.

1.4 The cascade of cascading style sheets

One of the fundamental points to understand about CSS is the cascade. When CSS was formed, it was around the concept of cascading, which allows styles to overwrite or inherit from each other. This then paved the way for multiple stylesheets which compete over the presentation of the web page.

Because of this, while inspecting an element using the browser's developer tools, we can sometimes see multiple CSS values fighting over each other to be the one rendered by the browser. How the browser decides which CSS property values to apply to an element is through specificity.

Specificity is when the browser (or the user-agent) decides which declarations are relevant to the HTML and applies the styling to that element. One aspect in which specificity is calculated is the order in which style sheets are applied. When multiple stylesheets are applied the styles in the later will override styles provided by the previous stylesheet. In other words, the last one declared wins.

In CSS there are three different stylesheet origins.

1.4.1 User-agent style sheets

The first origin is the browser's default styles. When we first opened the project, before we added any styles to it, our elements did not all look the same. Our headers are bigger and bolder than our text for example. This is defined by the user-agent (UA) style sheet. These have the lowest priority and we find that different browsers present HTML properties slightly differently.

Most of the time UA stylesheets set the font size, border styles, and some basic layout for form elements, like the text input or progress bar. This can be useful if the user stylesheet cannot be found or there is an error loading the file. The UA stylesheet provides some fallback styling, which allows the user to still use the website.

1.4.2 Author stylesheets

The stylesheets we as developers write are known as the author stylesheets and they typically have the second highest priority in terms of the styles the browser displays. When we create a web the CSS stylesheets we write and apply to our web pages are Author stylesheets.

1.4.3 User stylesheets

The user who is accessing our web page can use their own stylesheet to override both author and UA styles. This can improve their experience, especially for disabled users.

Instances of where a user may use their own style sheets include to set a minimum font size, choose a custom font, improve the contrast or increase the spacing between elements. Any user can apply a user stylesheet to a webpage. How these are applied to the webpage will depend on the browser, usually through the browser settings or by using a plugin. Once applied

the, the user stylesheet is only applied for the user that added it, on the browser they applied it to. Whether the change is carried over from one device to another, will again depend on the browser itself and its ability to sync plugin and setting cross devices.

1.4.4 CSS reset

Default styles provided by the browser are not consistent. Each browser has their own stylesheets. Default styles are different in Chrome than they are in Safari for example. This can create some challenges if we want our applications to look the same across all browsers. Luckily for us there are two options available, CSS Resets or Normalize. Although either can be used to solve cross browser styling issues, they work in radically different ways.

Reset says we don't want any of the browser default styling at all and resets them all. Without any author styles applied, all elements regardless of what they are, look like plain text as we can see in Figure 1.7.

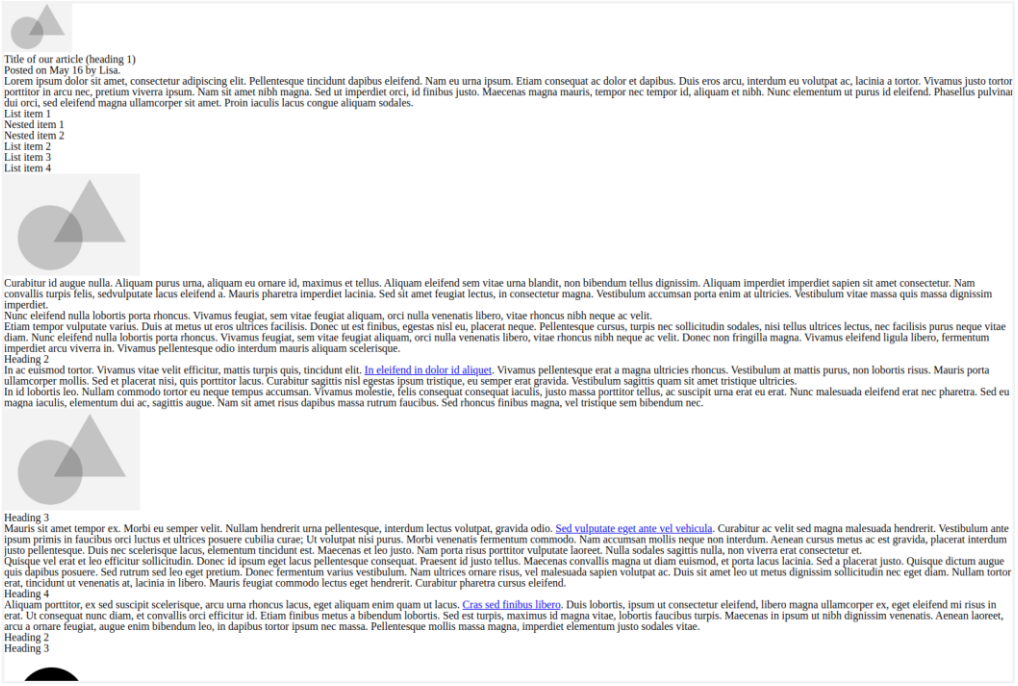

Figure 1.7 CSS Reset applied

To apply CSS Reset to our project first we create a reset stylesheet to add to our project. In our project folder we create a file called `reset.css`. We then copy the reset CSS into the file. Many reset options exist but a commonly used option can be found at https://meyerweb.com/eric/tools/css/reset/.

Finally we need to link our stylesheet to our HTML. Because order matters, we want to make sure we include the reset CSS **before** our author styles in our `head`. Our HTML will therefore look as seen in Listing 1.3.

Listing 1.3

```
<head>
  ...
  <link rel="stylesheet" href="reset.css">  #A
  <link rel="stylesheet" href="styles.css"> #B
 </head>
```

#A Reset stylesheet
#B Author stylesheet

The benefits of CSS Reset is that we now have a blank slate to start from. As seen in Figure 1.7 all or our elements now look like plain text so the downside is that we now need to define basic styles for all elements including adding bullets to lists, differentiating header levels, etc. Furthermore, each version of CSS Reset will be slightly different based on the version and the developer who authored it.

Our other option is Normalize. Instead of resetting the styles, it specifically targets elements that have differences across browsers and applies rules to standardize them.

1.4.5 Normalize

Like CSS Reset, Normalize will style things slightly differently depending on the version and author. A commonly used option for Normalize can be found at https://necolas.github.io/normalize.css/. We can apply it to our project in much the same way we did the CSS Reset. We create a file, copy the code into the file, and link it to our HTML. Once applied (Figure 1.8) we notice that our HTML looks the same as it did originally as most of the discrepancies it handles are on form elements and buttons. Depending on the browser we are using, we may notice a difference in the size of the `h1`.

Title of our article (heading 1)

Posted on May 16 by Lisa.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque tincidunt dapibus eleifend. Nam eu urna ipsum. Etiam consequat ac dolor et dapibus. Duis eros arcu, interdum eu volutpat ac, lacinia a tortor. Vivamus justo tortor, porttitor in arcu nec, pretium viverra ipsum. Nam sit amet nibh magna. Sed ut imperdiet orci, id finibus justo. Maecenas magna mauris, tempor nec tempor id, aliquam et nibh. Nunc elementum ut purus id eleifend. Phasellus pulvinar dui orci, sed eleifend magna ullamcorper sit amet. Proin iaculis lacus congue aliquam sodales.

1. List item 1
    - Nested item 1
    - Nested item 2
2. List item 2
3. List item 3
4. List item 4

Curabitur id augue nulla. Aliquam purus urna, aliquam eu ornare id, maximus et tellus. Aliquam eleifend sem vitae urna blandit, non bibendum tellus dignissim. Aliquam imperdiet imperdiet sapien sit amet consectetur. Nam convallis turpis felis, sedvulputate lacus eleifend a. Mauris pharetra imperdiet lacinia. Sed sit amet feugiat lectus, in consectetur magna. Vestibulum accumsan porta enim at ultricies. Vestibulum vitae massa quis massa dignissim imperdiet.

> Nunc eleifend nulla lobortis porta rhoncus. Vivamus feugiat, sem vitae feugiat aliquam, orci nulla venenatis libero, vitae rhoncus nibh neque ac velit.

Etiam tempor vulputate varius. Duis at metus ut eros ultrices facilisis. Donec ut est finibus, egestas nisl eu, placerat neque. Pellentesque cursus, turpis nec sollicitudin sodales, nis tellus ultrices lectus, nec facilisis purus neque vitae diam. Nunc eleifend nulla lobortis porta rhoncus. Vivamus feugiat, sem vitae feugiat aliquam, orci nulla venenatis libero, vitae rhoncus nibh neque ac velit. Donec non fringilla magna. Vivamus eleifend ligula libero, fermentum imperdiet arcu viverra in. Vivamus pellentesque odio interdum mauris aliquam scelerisque.

Heading 2

In ac euismod tortor. Vivamus vitae velit efficitur, mattis turpis quis, tincidunt elit. In eleifend in dolor id aliquet. Vivamus pellentesque erat a magna ultricies rhoncus. Vestibulun at mattis purus, non lobortis risus. Mauris porta ullamcorper mollis. Sed et placerat nisi, quis porttitor lacus. Curabitur sagittis nisl egestas ipsum tristique, eu semper erat gravida. Vestibulum sagittis quam sit amet tristique ultricies.

In id lobortis leo. Nullam commodo tortor eu neque tempus accumsan. Vivamus molestie, felis consequat consequat iaculis, justo massa porttitor tellus, ac suscipit urna erat eu erat. Nunc malesuada eleifend erat nec pharetra. Sed eu magna iaculis, elementum dui ac, sagittis augue. Nam sit amet risus dapibus massa rutrum faucibus. Sed rhoncus finibus magna, vel tristique sem bibendum nec.

Heading 3

Mauris sit amet tempor ex. Morbi eu semper velit. Nullam hendrerit urna pellentesque, interdum lectus volutpat, gravida odio. Sed vulputate eget ante vel vehicula. Curabitur ac velit sed magna malesuada hendrerit. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Ut volutpat nisi purus. Morbi venenatis fermentum commodo. Nam accumsan mollis neque non interdum. Aenean cursus metus ac est gravida, placerat interdum justo pellentesque. Duis nec scelerisque lacus, elementum tincidunt est. Maecenas et leo justo. Nam porta risus porttitor vulputate laoreet. Nulla sodales sagittis nulla, non viverra erat consectetur et.

Figure 1.8 Normalized applied to our project

The good news is that UA stylesheet differences are far less of an issue than they were 10 to 15 years ago. Today browsers are more consistent around their choice of styling than they once were, so using CSS Reset or Normalize is more of a personal choice rather than a necessity. Some differences do still exist however, and regardless of whether or not we use CSS Reset or Normalize, we should be testing our code across a variety of devices and browsers.

1.4.6 !important

The `!important` annotation is one we may have seen in some stylesheets. Often used as a last resort when all else fails, it is a way to override the specificity and declare that a particular value is the most important thing regardless of all else. However, with great power comes great responsibility. The `!important` annotation was originally created as an accessibility feature. Remember how we talked about users being able to apply their own styles in order to have a better user experience? It was created to help users define their own styles without having to worry about specificity. Since it overrides any other styles, it was created to ensure that the user's styles would always have the highest importance and therefore be the ones applied. Making use of `!important` is therefore considered bad practice and should not be used in our author stylesheets.

1.5 Specificity

When there are multiple property values being applied to an element, one will win over the others. How we determine the winner is through a multi step process. We will ignore `!important` for the time being as it breaks the normal flow, and we will come back around to it later.

We first look at where the value comes from. Anything explicitly defined in a rule will override inherited values. For example (listing 1.4), if we set a font color to red on the body tag, the elements that are inside the tag will have red text. The font color is inherited by those elements. If we specifically set a different color on a paragraph inside of the body, the inherited red value would be overwritten by the more specific blue value set on the paragraph. Therefore, that paragraph's text color would be blue.

Listing 1.4 Example of inheritance

```
<body> #A
  <h1>Example</h1> #B
  <p>My paragraph</p> #C
<body> #D

body { color: red } #E
p { color: blue }   #E
```

#A start of HTML
#B our header would inherit the red color
#C the paragraph's color would be blue as set by the paragraph rule
#D end of HTML
#E CSS Rules

Not all property values will be inherited. Theme related styles such as color and font size, will generally be inherited. Layout considerations are generally not. This is a very loose guideline, and there are definitely exceptions but this is a good place to start. We will cover exceptions as a case by case when they appear throughout the projects.

If inherited versus explicit doesn't act as a tiebreaker to determine the value, we then look at the type of selector which was used and mathematically calculate the specificity value. We

will get into more detail about what each selector type is in this chapter, but let's look at how the math is applied.

We look at the selector, categorize the types of selectors being used by the rule, then apply the type value. We then add all of the values and get a final specificity value. Figure 1.9 diagrams the process. The biggest number wins, therefore rule #1 in the diagram would win over rule #2.

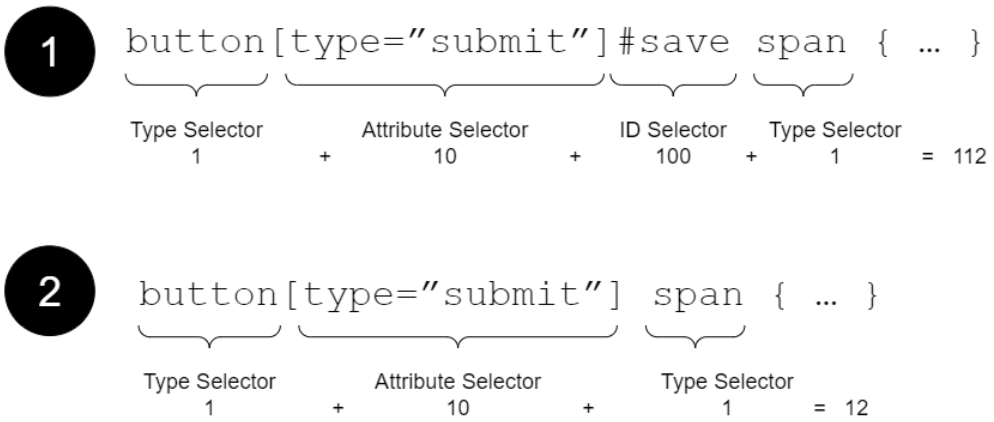

Figure 1.9 Calculating specificity

Specificity values by selector type are as follows:

- 100 - ID selectors
- 10 - Class selectors, attribute selectors, and pseudo classes
- 1 - Type selectoros, and pseudo elements
- 0 - Universal selectors

If we still have a tie we look at which stylesheet the style originated from. If both values come from within the same stylesheet, the one later in the document wins, if they come from different stylesheets, the order is as follows

- User stylesheet
- Author stylesheet(s) (in order in which they are being imported, last one wins)
- User Agent stylesheet

We set `!important` to the side earlier. Now that we understand the normal flow, let's add it back into the mix. When a value has the `!important` annotation, the process is short circuited and the value with the annotation automatically wins regardless of anything else. If both values have the `!important` annotation, the normal flow is followed. The diagram in Figure 1.10 shows the flow through the stylesheets including property values with the `!important` annotation.

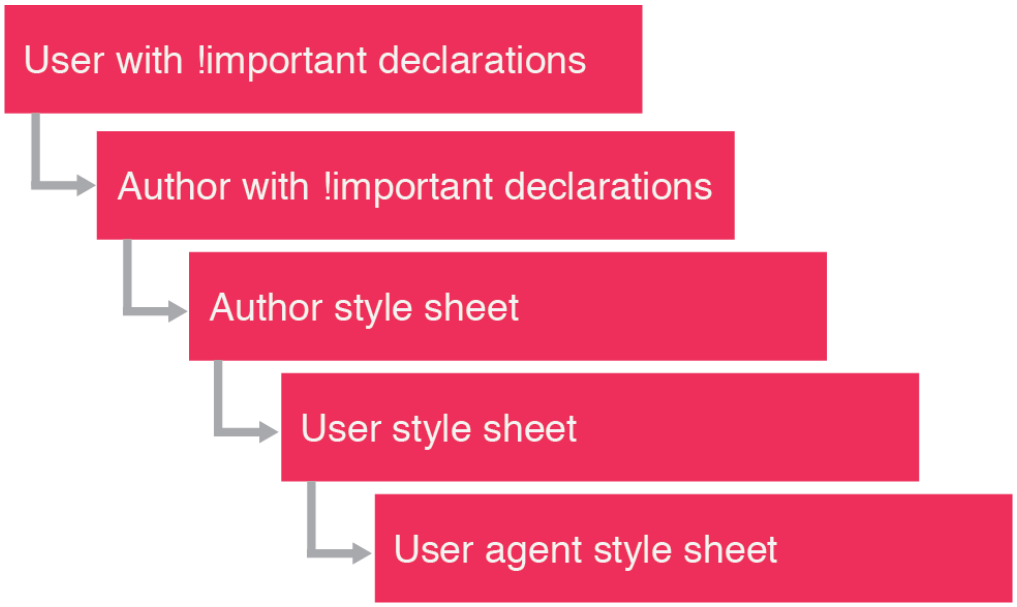

Figure 1.10 CSS order of precedence

We established that the type of selector will affect the class's specificity. Let's take a closer look at the selectors and use them in our project.

1.6 CSS selectors

The selector sets what HTML elements we want to target which is then followed by the declarations.

In CSS there are seven ways to target the HTML elements we want to style. Let's first look at the basic selectors.

1.6.1 Basic selectors

The most common method is selecting elements based on name, ID or class name.

TYPE SELECTOR

The type selector targets the HTML element by name. The benefit of using the type selector is that when reading through our CSS we can quickly work out which HTML elements would be impacted if we made changes to the rule. It also does not require us to add any particular markup to the HTML in order to target the element.

Let's use a type selector to target all of our headings (`h1` through `h6`), and change their color to crimson.

Our CSS would be: `h1, h2, h3, h4, h5, h6 { color: crimson; }`. Figure 1.11 shows that all our headers have now changed colors.

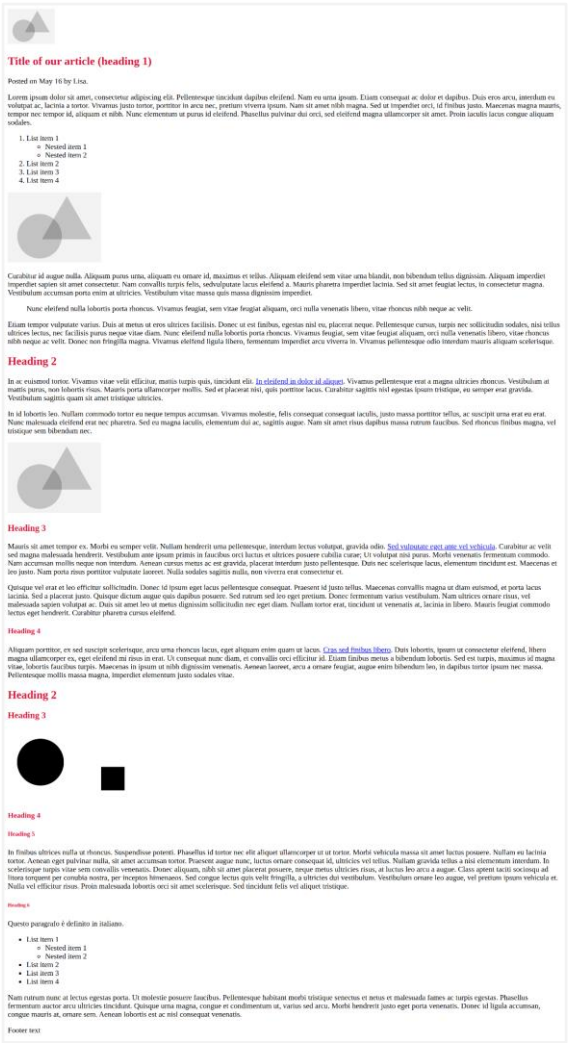

Figure 1.11 Header color change

CLASS SELECTORS

Class selectors can be used on as many different elements as we want. By applying a class name to elements, we are grouping multiple HTML elements together so when we apply styles, they will roll out to any element with that class name.

Classes can be added to HTML by using the class attribute. Within the class attribute, we can add as many values (or classes) as we want, each separated by a space. Unlike declarations which have to be written in American English, class names can be written in any language using any spelling variations.

There are many ways and methods to write our class names, such as Block, Element, Modifier (BEM) methodology or Scalable and Modular Architecture for CSS (SMACSS) which are style guides for writing consistent stylesheets. The main point is to write class names that make sense to anyone.

For example adding the class name "text" to our paragraph elements would be highly confusing. This is because other elements, like our headings, can also be thought of as text, so it might not be not clear which element we are specifically referring to. Applying class names based on a specific style such as a color can also be dangerous. Giving an element a class name of `blue` might work immediately, but if the design changes and the color applied is now red, our class name will no longer make sense.

In our HTML we find that some of our headings have a class of `small-heading`. We are going to create a rule that selects `small-heading` and changes the text of the elements to uppercase.

To select the `small-heading` class name, in the CSS we will first type dot (`.`) followed by the class name `small-heading` our styles then go into the curly brackets, same as for the class selector: `.small-heading { text-transform: uppercase }`. Figure 1.12 now shows our uppercased headings. Notice that the other headings were not affected. Only those on which the class was applied.

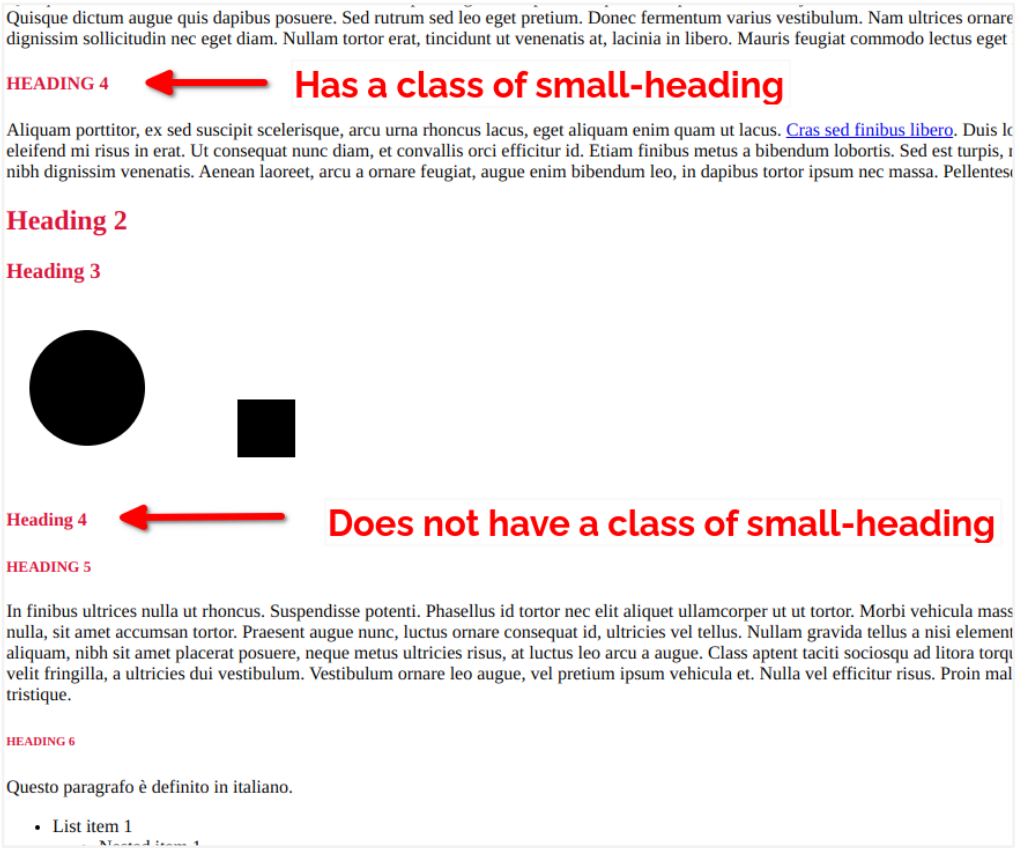

Figure 1.12 Class-selector applied to heading 4, 5 and 6

ID SELECTOR

In HTML ids are unique. Any given id should only be used once on a web page. If it is repeated more than once then our code is considered to be invalid HTML. Generally we should avoid using the id selectors because due to their need to be unique in the HTML, classes being constructed against the id are not easily reusable. Furthermore, an id selector is one of the most specific selectors available making the styles applied using an id difficult to override. Unless the uniqueness of the element is key, id attributes should be avoided. Like class names, id names can also be written in any language.

In our example article there is a blockquote with an `id` attribute containing a value of `quote-by-author`. In our CSS, to select the blockquote we use a hash (#) immediately followed by the `id` we want to target. We then have curly bracket inside of which we place our declarations as seen in Listing 1.4.

Listing 1.5 shows our rule.

```
#quote-by-author {
  background: lightgrey;
  padding: 10px;
  line-height: 1.75;
}
```

Figure 1.13 shows the code applied to our project.

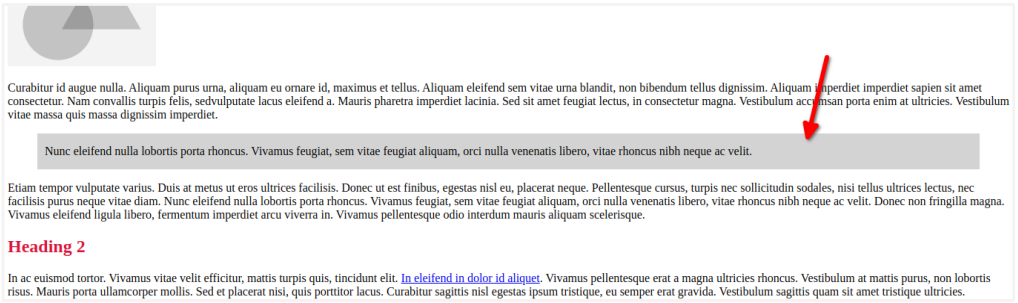

Figure 1.13 shows the output

1.6.2 Combinators

Another way to write CSS is through combinators, which allow for more complex CSS without overusing `class` or `id` names.

There are four combinators:

- Descendant combinator (`space`)
- Child combinator (`>`)
- Adjacent sibling combinator (`+`)
- General sibling combinator (`~`)

One important concept to understand is the relationships between elements. In the next couple of examples we'll look at how we can use the relationships between elements to target different HTML elements to style our article. Figure 1.14 introduces the types of relationships we are going to look at.

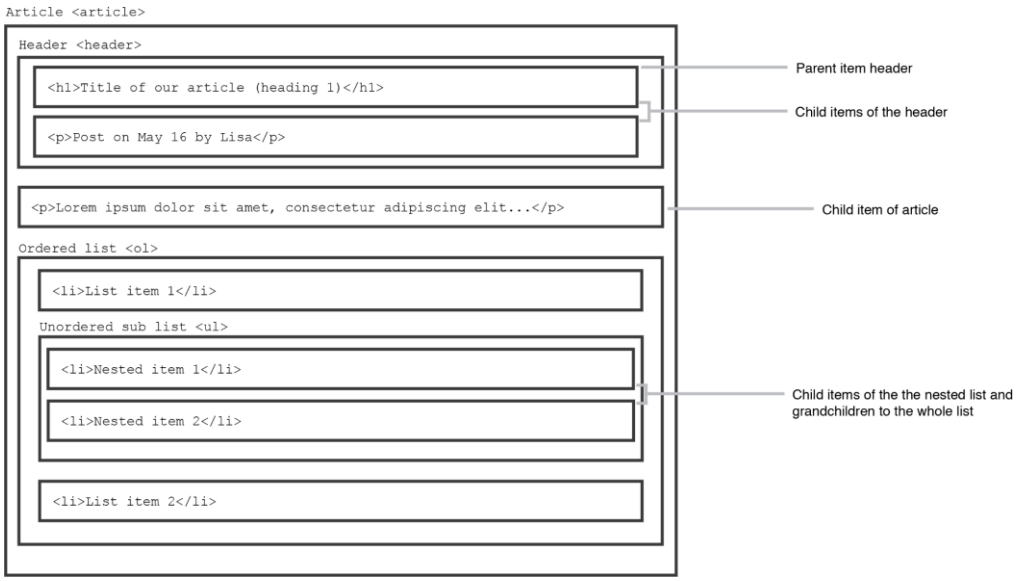

Figure 1.14 The relationship between elements in HTML

1. DESCENDANT COMBINATOR (SPACE)

Descendant combinators select all of the HTML elements within a parent. A descendant combinator is made up of three parts - the first is the parent, which in this case is the article element. The parent is followed by a space, and then any element we want to select. The syntax is diagrammed in Figure 1.15.

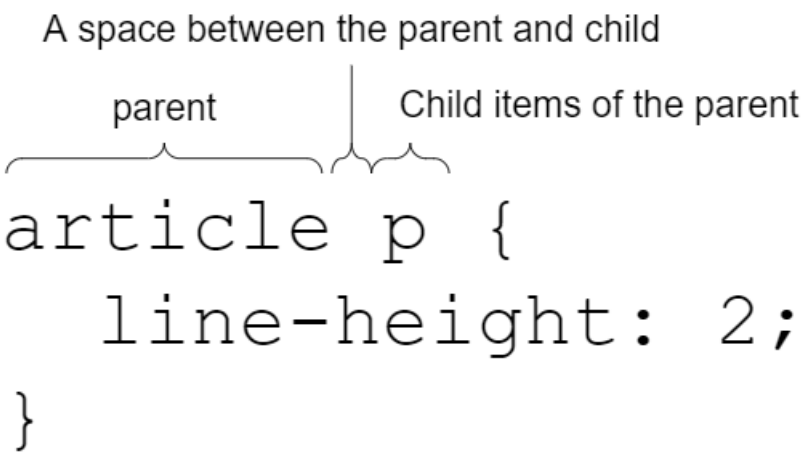

Figure 1.15 An example of a descendant combinator

In this example, the browser would find any use of the `article` element and then target all descendant paragraphs (`p`) in the parent `article` element and make the text double spaced. Applied, our article looks as seen in Figure 1.16.

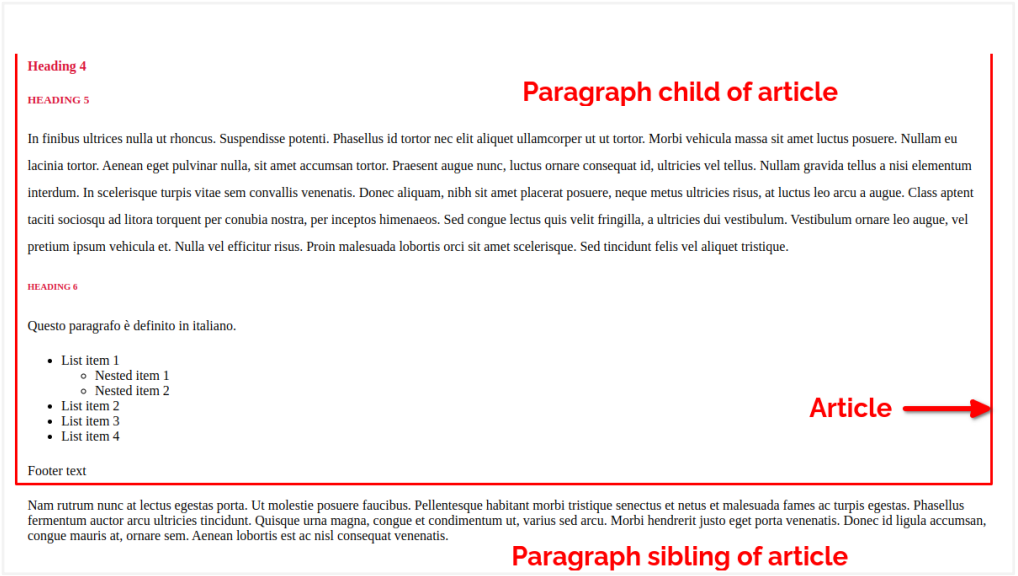

Figure 1.16 Child paragraphs are double spaced.

2. CHILD COMBINATORS (>)

Children combinators are elements that are children of a parent element. This is different from the descendant combinator because the targeted element must be an immediate child. A descendant combinator can select any descendent (child, grandchild, great-grandchild…).

In our project we will style the list items in the article. As we can see in our HTML below (Listing 1.4), we have an unordered list (`ul`) with list items (`li`). That first child element has its own nested items, which would be grandchildren and great grandchildren.

Listing 1.6 HTML list items .

```
<ul class="list">              #A
  <li>List item 1</li>         #B
   <ul>                        #C
    <li>Nested item 1</li>     #D
    <li>Nested item 2</li>     #D
   </ul>
  <li>List item 2</li>         #B
  <li>List item 3</li>         #B
  <li>List item 4</li>         #B
 </ul>                         #A
```

#A Parent Item (.list)
#B Children of .list
#C Grand child of .list
#D Great grandchildren of .list

We are going to style only the list items (the children) in a crimson color, without affecting the nested list items (the great grandchildren). So the browser will find the `list` class and target just the child list items (`li`) directly under the `list` class and change the `color` to `crimson`. We will use the following CSS: `.list > li { color: crimson; }`

As we can see in Figure 1.17, the list items are now in crimson while the nested list items are still black. This is one of the benefits of using the child combinator. It allows us to style just the child elements but not the grandchildren or anything further down the hierarchy.

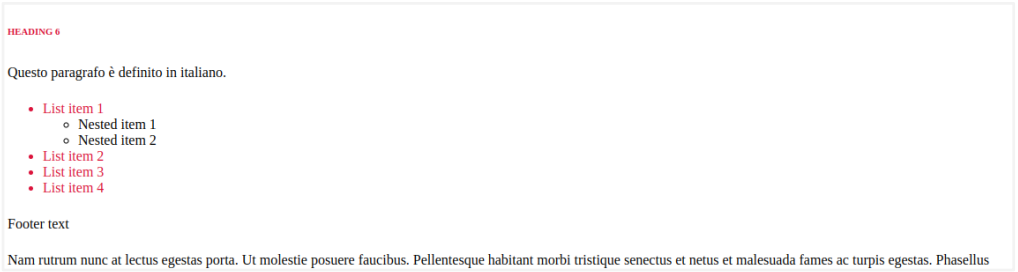

Figure 1.17 Child combinator applied to list items

So we can do this in reverse and select the parent of the child element, right? The short answer is no. There is currently no way to select the parent item in CSS, so the following example would not work: `article < p { color: blue; }`.

3. ADJACENT SIBLING COMBINATOR (+)

When we need to style an element that is at the same level as another (like how your brother or sister is on the same level of the family tree as you) we can use the adjacent sibling combinator.

If we want to target the element that is directly after another we can use the adjacent sibling selector.

In this example (Listing 1.7), the browser will find any uses of the `header` tag and then target the first paragraph (`p`) immediately after (or adjacent) to the `header` tag and change the `font-size` to `1.5rem` and the `font-weight` to `bold`.

Listing 1.7 Adjacent sibling combinator

```
header + p {
  font-size: 1.25rem;
  font-weight: bold;
}
```

Figure 1.18 shows the code applied to our article.

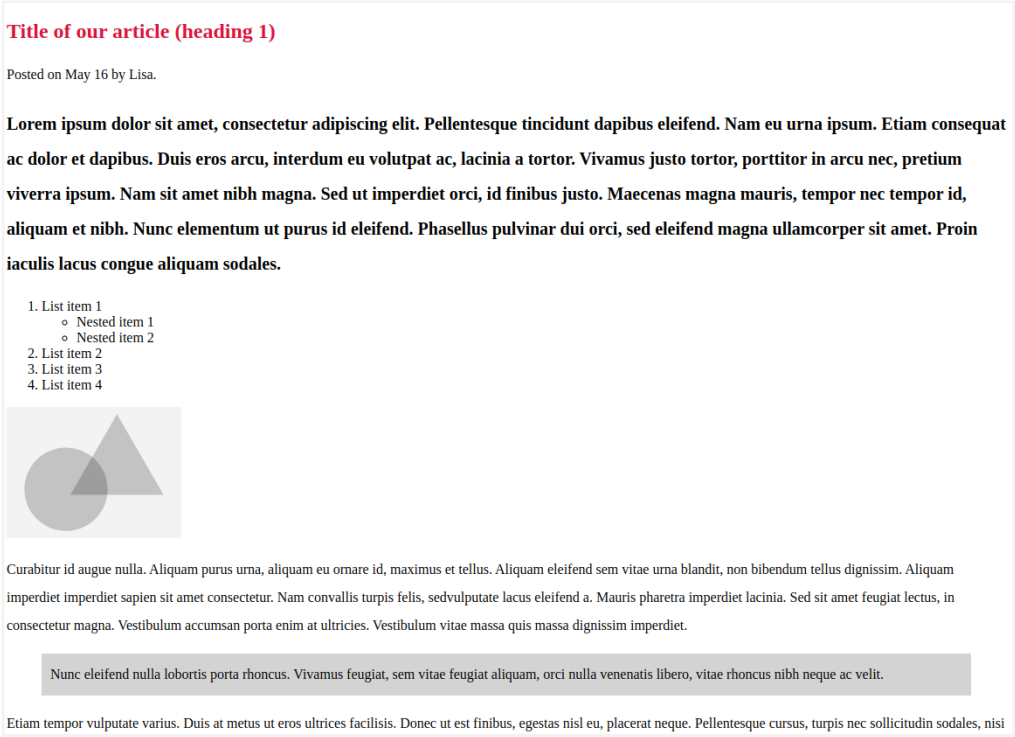

Figure 1.18 Adjacent sibling combinator applied to paragraph immediately after header

This could be useful if we are trying to style the first element differently to any others to make it stand out. We might see this when looking at a newspaper. The first paragraph might be made to look more prominent than the rest to catch our attention. Another use case is for error handling in forms. Adjacent sibling combinations allow us to display an error message to the user immediately following an invalid value in a form control.

It is worth noting that similar to the child combinator, we cannot find a previous sibling. `header - p { … }` to find a paragraph immediately before the header will *not* work.

4. GENERAL SIBLING COMBINATOR (~)

The general sibling combinator is more open-ended than the previous methods, as it allows us to target all elements that are siblings after the element targeted by the selector.

In our example, we will style all images that come after the element `header`. We will notice that we have three placeholder images. The first one is smaller and could be a logo or an author photo and resides above the `header`. We do not want to style it. The other two images are further down in the article. We want to apply a border around them to keep the color theme consistent with the rest of the article.

Our rule will look as follows: `header ~ img { border: 4px solid crimson; }`. The browser will find the `header` element and target all the sibling images (`img`) after that element and add a `border` that's `4px` in thickness, a `solid` line (as opposed to a `dotted`, `dashed`, or `double` line) and is colored `crimson`.

We can see the code applied to our article in Figure 1.19.

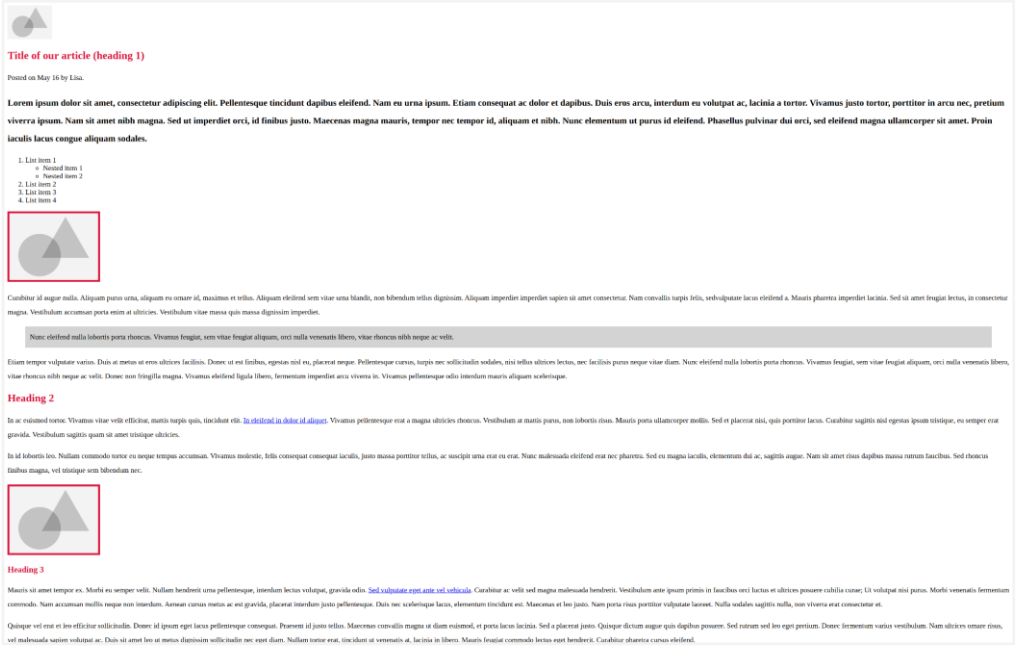

Figure 1.19 General Sibling Combinator targeting sibling images of header

1.6.3 Pseudo-class and pseudo-element selectors

PSEUDO-CLASS

A pseudo-class is added to a selector to target a specific state of the element. They are especially useful for elements the user will interact with such as links, buttons, and form fields. Pseudo classes use a single colon (`:`) followed by the state of the element. Within our article there are a few links. We have not styled them in any way, their styles therefore come from the user agent stylesheet. For most browsers, they have an underline and will vary in color based upon whether the link has been previously visited (the url is found in the browser's history).

With links, there are a few states to consider. The most common are:

- `link`: an anchor tag that contains an `href` attribute and a url that is not found in the user's browser history
- `visited`: an anchor tag that contains an `href`, and a url that is found in the user's browser history
- `hover`: the user has their cursor over the link, but has not pressed it
- `active`: the user is clicking the link and holding the click down
- `focus`: a focused element is an element that will receive keyboard events by default. When a user clicks on an element it automatically gains focus (unless there is JavaScript altering this behavior). Using the keyboard to navigate between form fields, links, and buttons will also change the element that is in focus.
- `focus-within`: is applied to a parent element. If a child of the parent has focus, focus-within styles will be applied.
- `focus-visible`: applies only when the focus has been gained via keyboard navigation or the user is interacting with the element via the keyboard.

There is also the `:target` pseudo-class which we will explore later in this book when we look at `text-fragments`.

In our current article project, we will first create an `a:link` rule to changing the color of anchor tags that contain and href element and have not been visited to a light blue using its Hex color code (`#1D70B8`).

The `:visited` state should be different from the `:link` state because it should indicate to the user that they have visited that page before. Often we see websites not differing between the two states however, differentiating the two can provide a better experience.

In our example, we will change the `:visited` state to a purple `color` using the Hex code value `#4C2C92`.

We will then handle the `:hover` state. The hover state does not apply to mobile users as there is no way to recognize a user hovering over a link. In our article, we will change the `:hover` state text color to a dark blue using the hex code value `#003078`.

Finally we will handle the `:focus` state. The focus state can be used on links, buttons, and form inputs. The focus state is shown when the user clicks or taps on an element. When the element is focused we will add a 1 pixel crimson outline to the element.

All put together our link rules are as follows (Listing 1.8):

Listing 1.8 Styling links using pseudo elements

```
a:link {
  color: #1D70B8;
}
a:visited {
  color: #4C2C92
}
a:hover {
  color: #003078;
}
a:focus {
 outline: solid 1px crimson;
}
```

We can view the different element states by going into our browser, right clicking and selecting "inspect". Then we will typically have a view of the HTML with the CSS on the side. By clicking the `:hov` button in the Styles section we can see a panel which may say something like "force element state" and then we can toggle different pseudo classes on and off. Figure 1.20 shows the Chrome developer tools with the :hov panel open.

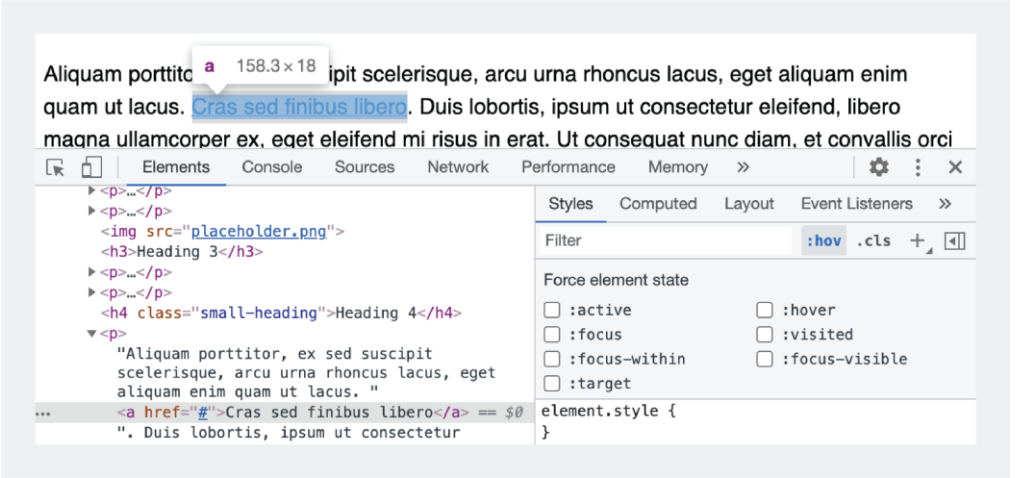

Figure 1.20 Viewing different elements states using the browser developer tools

1.6.4 Pseudo-elements

Pseudo elements use a double colon (`::`). The purpose of pseudo-elements is to allow us to style a specific part of an element. We may also sometimes see pseudo elements being written with a single colon. Either will work. For example we can target the first letter of a paragraph using the pseudo-element `::first-letter` rather than wrapping the letter in something like a `span` element which would break apart the word and clutter our HTML. This allows us to create complex CSS without having to complicate the HTML.

In our article, we used the adjacent sibling combinator to make our first paragraph bold and a larger font size than the rest. Now we're going to change the color of the first letter of that first paragraph and change the font style to `italic`.

First we get the `header` element, then we target the first letter (`::first-letter`) of the paragraph (`p`). With our selector created, when then add our declarations. Our CSS will therefore look as follows (Listing 1.7):

Listing 1.9 Selecting the first letter

```
header + p::first-letter {
  color: crimson;
  font-style: italic;
}
```

Applied, we can see in Figure 1.21 that our first letter is now red and italicized.

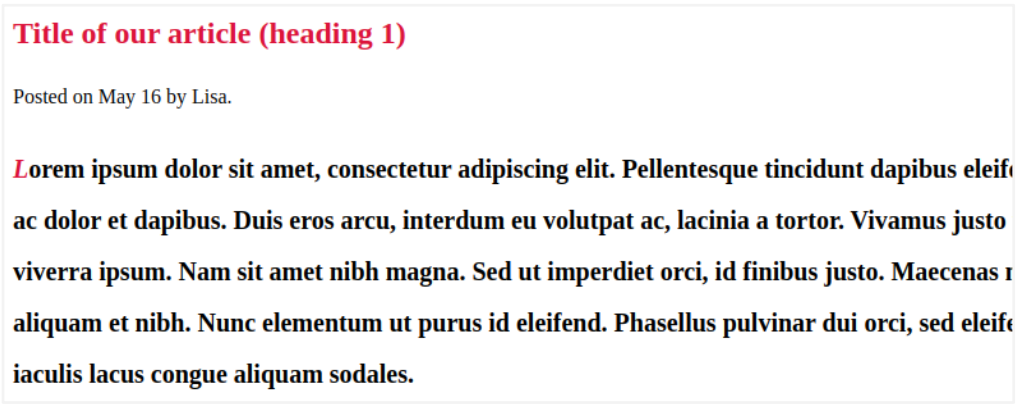

Figure 1.21 Pseudo class targeting the first letter of the first paragraph immediately after the header

You might wonder why these selectors are called pseudo-class and pseudo-elements. Let's look at what pseudo means:

> ***not genuine; false or pretend***

This makes sense as technically we are targeting a state or parts of an element that might not exist yet, so we are just pretending. Pseudo-classes can change as we have seen with styling our own links. We are creating `if` conditions that are set in motion by the user's actions. For example, `if` the user `hovers` then the browser will show one style, but `if` the user `visited` the link then the browser will show another style. Not all pseudo-elements and classes work on all HTML elements, throughout this book we will look at where we can use pseudo classes and with which HTML elements.

1.6.5 Attribute value selectors

Commonly used for styling links and form elements, the attribute selector styles HTML elements with a specified attribute. The attribute value selector looks for a specific attribute with the same value.

In our article we have some content that is in Italian. The language of the paragraph is specified using the `lang` attribute. (Listing 1.8).

Listing 1.10 Italian content

```
<p lang="it">Questo paragrafo è definito in italiano.</p>
```

To hint to users that this content is in Italian we will add the Italian flag emoji using CSS. The browser will find the language (`lang`) attribute with the value of Italian (`it`) and then add an Italian flag emoji `before` it. We will notice in Listing 1.9 that we used a pseudo-element `before` as well. We can use multiple types of selectors together to target the exact part of the HTML we want to style.

Listing 1.11 Using multiple types of selectors to add a flag before Italian content

```
[lang="it"]::before {
  content: "IT"
}
```

Once applied, we can see in Figure 1.22 that our Italian content now has an emoji flag before it.

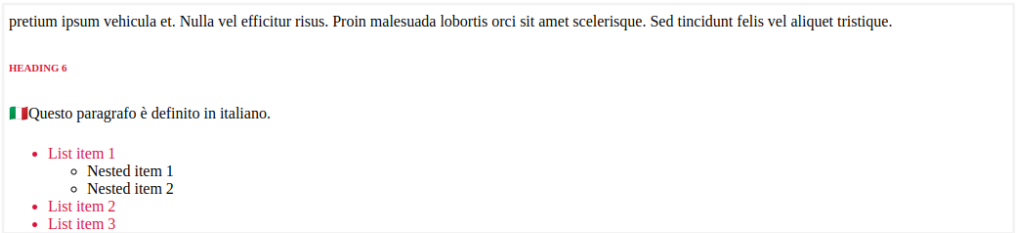

Figure 1.22 Italian flag being applied using attribute and pseudo element selectors

1.6.6 The universal selector

The broadest type of selector is the universal selector which uses the asterisk symbol (`*`). This means that any declarations made using the universal selector will be applied to all of the HTML elements. Sometimes this can be used to reset CSS, but in terms of specificity the universal selector has a specificity value of 0, which means it can be overridden easily if needed.

In the case of our example project, we will use a universal selector (`*`) to set the `font-family` to `sans-serif` so it will be consistently sans-serif throughout the article: `* { font-family: sans-serif; }`.

Once applied we can see that all of the text in our document uses a sans-serif font regardless of their element type (Figure 1.23).

Figure 1.23 Using the universal selector to change the font type on all elements

1.6.7 Namespace and the universal selector

It is possible to mix different languages together in the same document, such as embedding SVG code into HTML.

The issue with doing that is that it's difficult to distinguish between the various types of code. For example, we could have a link in our SVG graphic and a link in our HTML but we may want to style the SVG links differently than our HTML links.

To solve this we can use `namespaces` to uniquely identify code and set our definition of what that code does. Then, in the CSS, we can select that `namespace` and create styles just for that type of code.

In HTML5 when we insert code into a HTML file that is a known foreign element such as an SVG, MathML or XML it is automatically given its correct namespace.

Here a few examples of coding languages and their namespaces:

- HTML elements use the namespace of http://www.w3.org/1999/xhtml
- SVG elements use the namespace of http://www.w3.org/2000/svg
- MathML elements use the namespace of http://www.w3.org/1998/Math/MathML

In our project we have an SVG graphic within our HTML document. To declare a namespace in HTML we use the `xmlns` attribute. As we have learnt, it is optional to declare the namespace for an SVG graphic as it is a known foreign element, but we will declare it in this instance (Listing 1.10):

Listing 1.12 SVG Namespace

```
<svg xmlns="http://www.w3.org/2000/svg">
 <!--SVG Code-->
</svg>
```

Now that our SVG is in our HTML and we have optionally added the `namespace` value in the xmlns attribute, we need to add this known namespace to our CSS. In the CSS file at the top of the style sheet we add the `namespace` rule.

This is formed by writing `@namespace`. Then we give the namespace a prefix. The prefix given to the namespace can be whatever we choose, but for simplicity we have stuck with `svg` as this is the code we want to target. The name prefix is case sensitive, so `Svg` would be treated differently to `svg`. Then we add the `namespace` URI string in quotes: `@namespace svg "http://www.w3.org/2000/svg";`

Now that we have declared the namespace in our CSS let's look at how it can work with the universal selector.

We can use the universal selector to style all elements within that namespace. In other words, we style all elements within a certain coding language, like HTML or SVG. We can use the universal selector in the following ways:

- `*` or `*|*` - matches all elements, regardless of their namespace
- `ns|*` - matches all elements in the namespace `ns`
- `|*` - matches all elements without any defined namespace

The next step is to write our CSS rule. We want to apply a background color to our two SVG elements without styling the HTML. The first step is creating the selector, which is the namespace rule we called svg, followed by a pipe (|) and then an asterisk (*) which matches all elements in the `svg` namespace as seen in Listing 1.11.

Our declaration will be `fill` with a light gray Hex value of `#EDEDED`. We can specify as many namespaces as we want. But that namespace rule only applies to that stylesheet, so we will need to declare it once for each stylesheet we intend to use it in.

Listing 1.13 Namespace selector

```
svg|* {
  fill: #EDEDED;
}
```

Once applied, we can see that the elements in our svg image now have a gray background (Figure 1.24).

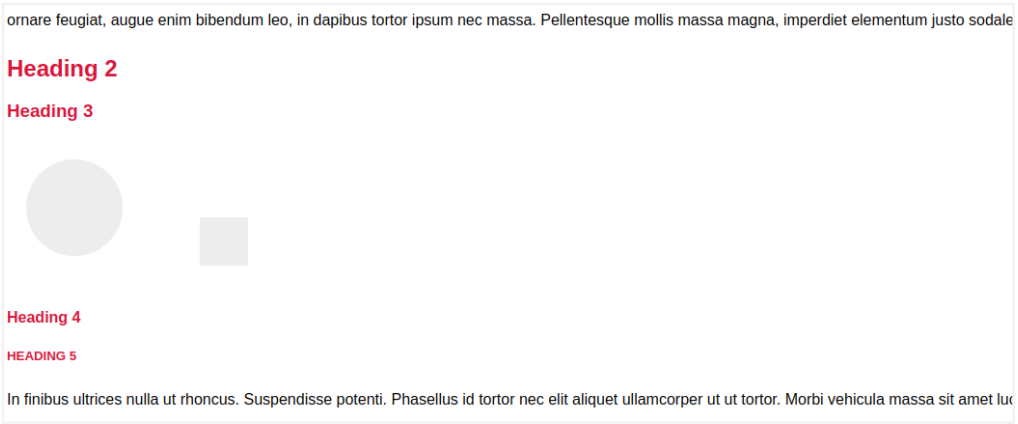

Figure 1.24: Using namespace and the universal selector to style the SVG element backgrounds

1.7 Different ways to write CSS

CSS has flexibility in the way we write our rules and formatting. We will now look at shorthand properties (which we will keep coming back to throughout the book) and different ways we can format CSS.

1.7.1 Shorthand

Shorthand is a way to replace writing multiple CSS properties by merging all the values into one property. We can do this with a few properties such as padding, margin, animation… all of which we will cover at various points throughout this book. The benefit of writing shorthand is that we reduce the size of our stylesheet which improves readability, performance, and memory usage.

Each shorthand property takes different values. Let's explore the one we used in our project. We have a `blockquote` in our article. When we styled it we used the `padding` property. We declared our padding as follows: `padding: 10px`. In doing so we used the shorthand. We could have written the following instead (Listing 1.12):

Listing 1.14 Padding expanded

```
  padding-top: 10px;
  padding-right: 10px;
  padding-bottom: 10px;
  padding-left: 10px
```

It is completely fine to write each declaration separately, but it is expensive in terms of computing performance. This is especially true since all of the property values are the same.

Instead we can use the `padding` property and put all four values on the same line. The order goes top, right, bottom, left as described in Figure 1.25.

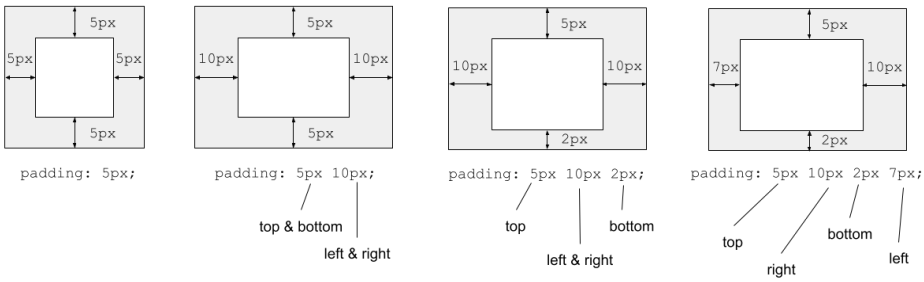

Figure 1.25 Padding shorthand property explained

1.7.2 Formatting

There are a few ways we can write CSS and often when viewing other people's code we will see different formats. Here are a few examples:

The multi-line format (Listing 1.13) is most likely the most popular of choices of formatting. Each declaration is on its own line and then either indented using either tabs or spaces.

Listing 1.15 The multi-line format

```
h1 {
  color: red;
  font-size: 16px;
  font-family: sans-serif
}
```

This variation on the multi-line format (Listing 1.14) places the opening bracket on its own line. It is something we may see in the PHP language. It could be considered unnecessary to place the opening bracket on its own line.

Listing 1.16 A variation of the multi-line format

```
h1
{
  color: red;
  font-size: 16px;
  font-family: sans-serif
}
```

The single-line format (Listing 1.15) makes a lot of sense, it is compact, and we can quickly scan through a file knowing the first part is the selector. However, the downside is that it can quickly become difficult to read if a rule contains many declarations.

Listing 1.17 The single-line format

```
h1 { color: red; font-size: 16px; font-family: sans-serif }
```

There are positives and negatives to all of the options, but we will notice that the projects in this book use a combination of option one and three. The main thing to know is that there is no right or wrong way, it generally comes down to what works best for you or your team. As long as the code is easy to understand, then that's all that matters.

Those with an eagle eye will also notice in the examples above there are no semicolons (;) on the declarations of the rules. Whether we want to add it or not is optional, but for better web performance it is best practice to remove it.

One of the best parts about CSS is that we can write it in the way that is most comfortable to us.

1.8 Summary

- CSS is a well established coding language and each part of CSS is made up of modules.
- Modules replaced large releases like CSS3, and there will not be a CSS4 release.
- Inline CSS can take the highest priority and has good performance, however it is very repetitive and hard to maintain.
- External CSS keeps our CSS separate from our HTML maintaining separation of concerns.
- As well as our own CSS, the browser applies default styling.
- The user may also apply their own CSS which can override the author and user agent stylesheets.
- Using !important is considered bad practice.
- A CSS rule consists of a selector and one of more declarations.
- There are many types of selectors against which we can create rules each with their own level of specificity

2

Designing a layout using CSS grids

This chapter covers

- Exploring grid tracks and how we can arrange our grid
- Using the minmax and repeat functions in CSS grids
- Working with the fractions unit and how its unique to CSS grids
- Creating template areas and placing the items on the areas
- Accessibility considerations when using grid
- Creating gutters between columns and rows within our grid

Now that we have a basic understanding of how CSS works, we can begin exploring our options for laying out HTML content, including grids, flexboxes and columns. First, we will focus on layout using grids.

A grid, in this sense, is a network of lines that cross each other to form a series of squares or rectangles. Now supported by all major browsers, grid has become a very popular layout technique.

Essentially a grid is made up of columns and rows, like a table. We first create our grid and then assign positions to our items, or elements, to that grid similarly to how we place boats on a grid when playing the game battleship.

Although sometimes compared to the concept of the table, grid layouts and tables have very different uses and fulfill very different needs. Grids are for layouts while tables are for tabular data. If the content being styled would be appropriate for an Excel sheet, then it's tabular data and should be placed in a table.

Historically we have used tables for layouts, and sometimes still need to (emails for example only support a small subset of CSS), but on the web it is considered bad practice. We can now use a grid instead.

CSS grid empowers us to be creative, and produce a range of layouts and adapt those layouts for different conditions in conjunction with media queries, as we will see. We will use grid to style our project, and by the end of the chapter, our layout will look as follows (Figure 2.1):

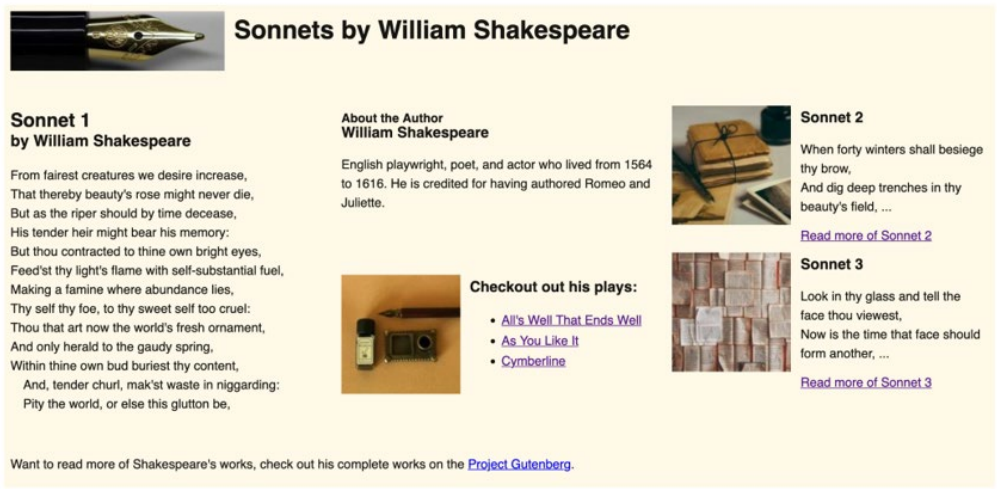

Figure 2.1 Final output

Our starting HTML found in the chapter-02 folder of the Github repository (https://github.com/michaelgearon/Tiny-CSS-Projects) looks as follows:

Listing 2.1 Project HTML

```
<body>
  <main> #A
    <header> #B
      <img src="./img/logo.jpg" alt="">
      <h1>Sonnets by William Shakespeare</h1>
    </header>
    <article> #B
      <h2>
        Sonnet 1
        <br><small>by William Shakespeare</small>
      </h2>
      <p>
        <span>From fairest creatures we desire increase,</span>
        ...
    </article>
    <aside> #B
      <section aria-label="sonnet 2">
        <img src="./img/image-1.jpg" alt="">
        <h3>Sonnet 2</h3>
        <p>
          When forty winters shall besiege thy brow,
          <br>And dig deep trenches in thy beauty's field, ...
        </p>
        <a href="">Read more of Sonnet 2</a>
      </section>
      <section>
        <img src="./img/image-2.jpg" alt="">
        <h3>Sonnet 3</h3>
        <p>
          Look in thy glass and tell the face thou viewest,
          <br>Now is the time that face should form another, ...
        </p>
        <a href="">Read more of Sonnet 3</a>
      </section>
    </aside>
    <section class="author-details"> #B
      <h3>
        <small>About the Author</small>
        <br>William Shakespeare
      </h3>
      <p>English playwright, poet, ...</p>
    </section>
    <section class="plays">  #B
      <img src="./img/play.jpg" alt="">
      <h3>Checkout out his plays:</h3>
      <ul>
        <li><a href="">All's Well That Ends Well</a></li>
        ...
      </ul>
      <h3></h3>
    </section>
    <footer>  #B
      <p>Want to read more ...</p>
    </footer>
  </main>
</body>
```

#A The container for our project
#B The child items within our container

We also have some starting CSS (Listing 2.2) just to guide us as we start to place our HTML elements in a grid format. We won't worry about these preset styles (such as margin, padding, colors, typography and borders) in this chapter, as these are concepts that are covered in other parts of the book and we specifically want to stay focused on the layout for this project.

We change the font from serif to sans-serif and increase the margin between the boundary of the browser window and the container using margin. We also float images to the left and adjust the line heights, typography, and padding.

Note that we added a border and some padding to the immediate children of the main element. These are to help us define our layout. We will be removing those later in the project.

Listing 2.2: Project starting CSS

```
body {
  margin: 0;
  padding: 0;
  background: #fff9e8;
  min-height: 100vh; #G
  font-family: sans-serif;
  color: #151412
}

main { margin: 24px }

img {
  float: left; #F
  margin: 12px 12px 12px 0
}

main > * { #A
  border: solid 1px #bfbfbf; #B
  padding: 12px;
}
main > *, section { display: flow-root } #C

p, ul { line-height: 1.5 }

article p span { display: block; }

article p span:last-of-type, #D
article p span:nth-last-child(2) { #D
  text-indent: 16px #D
} #D

.plays ul { margin-left: 162px } #E
```

#A Asterix and child combination selects any and all immediate children of main
#B Border to visually point out sections to be positioned using grid
#C Prevents images from bleeding out of their containers
#D Indents the last 2 lines of the sonnet

#E Indents the list; otherwise bullets are right up against the image
#F Allows text to wrap around the image
#G So that the background covers the whole page even when window is longer than the content

Our starting point looks as follows (Figure 2.2).

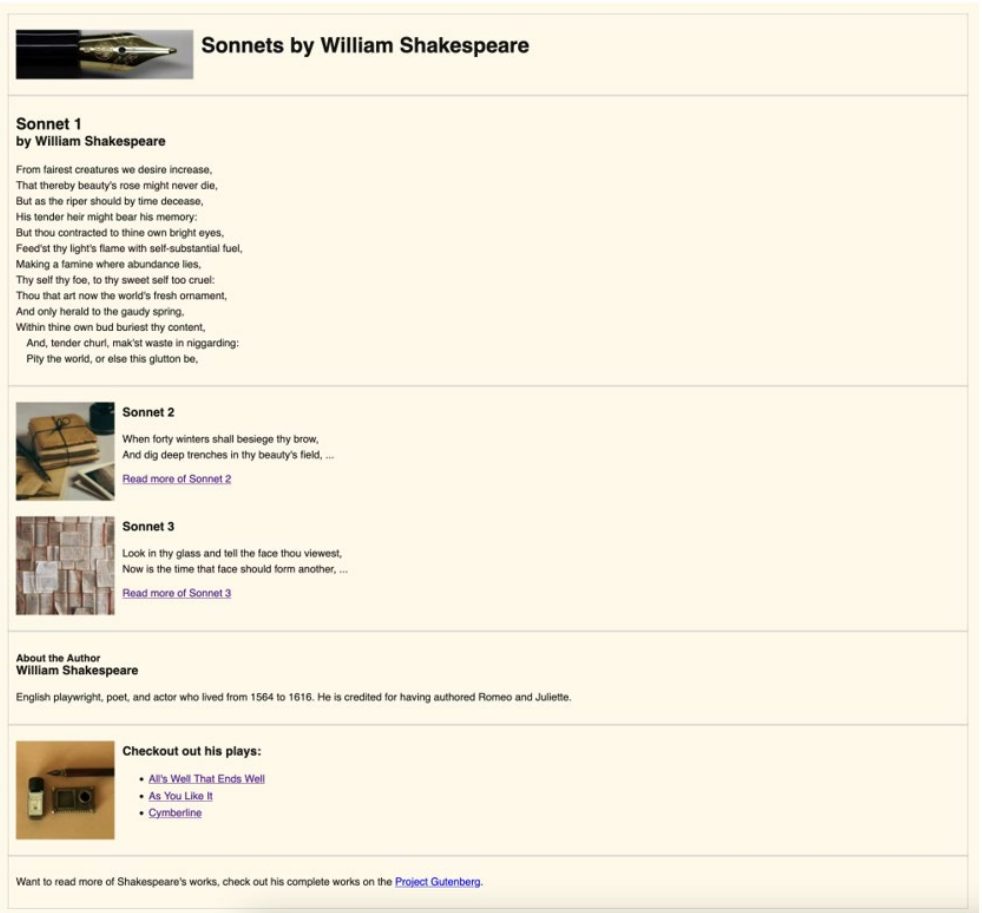

Figure 2.2 Starting point

CSS grids are a way to place items on a two-dimensional layout, horizontal (x-axis) and vertical (y-axis). In contrast, flexbox (covered in Chapter 6) is single-axis–oriented. It only operates on either X or Y depending on its configuration. In short, a CSS grid is for laying out items whilst a flexbox is for the alignment of items.

First let's set up our grid; we will then look at ways we can alter how our grid behaves based on widow size.

2.1 Display

The first part of arranging a grid is setting the `display` value to "grid" on the parent container item. When creating a grid playout, we can use one of 2 values:

- `grid`: when we want the browser to display the grid in a block-level box. The grid will take the full width of the container and set itself on a new line.
- `inline-grid`: this is when we want the grid to be an inline-level box. The grid will set itself inline with the previous content much like a span.

Difference between block-level and inline-level box

In HTML every element has a box surrounding it. A block-level box says that an element's box should use the entire horizontal space of its parent element, therefore preventing any other element being on the same horizontal line. In contrast, inline elements can allow other inline elements to be on the same horizontal line depending on the remaining space.

We will use the value grid for our layout:

Listing 2.3 Setting the display to grid

```
main {
  display: grid;
}
```

If we preview this in the browser we notice that visually nothing has changed. This is because the browser by default will display the direct child items in one column. The browser will then generate enough rows for all of the children elements.

Using the developer tools in our browser (Figure 2.3), we can see that programmatically the grid has been created even though the layout has not visually changed. To view the underlying grid on most browsers we can right click on the web page and select inspect from the menu.

In the Firefox inspect window, we select the parent container and we will see 2 things to indicate this is now a grid:

1. Purple lines around each direct child item
2. In the CSS panel to the right of the HTML, we see the `main` has a display of grid and next to it an icon
3. Once we click the grid icon beside `<main>`, the layout panel shows us our grid structure

Although we can follow similar steps for Chrome or Safari, Firefox does provide better functionality when viewing and editing the grid in the developer tools.

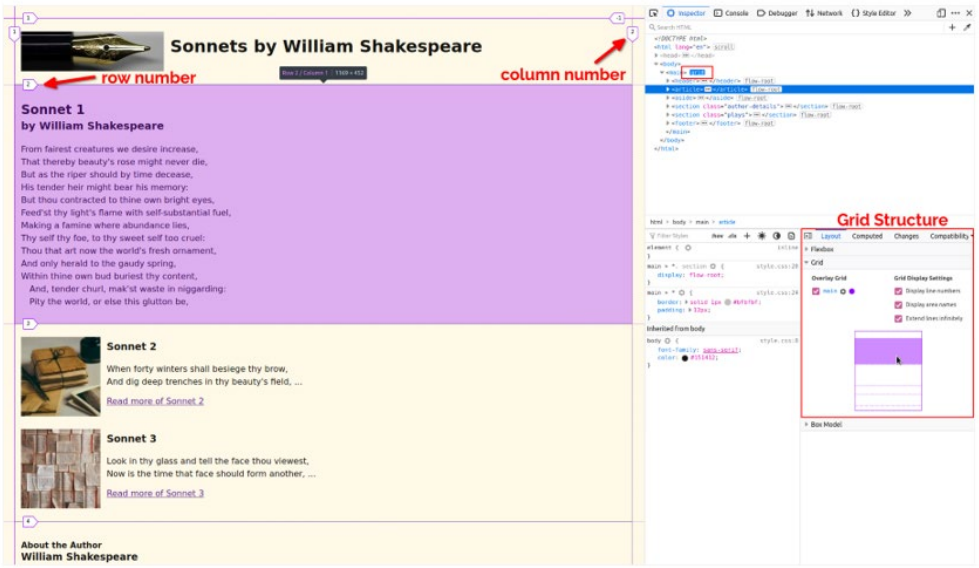

Figure 2.3 Development tools in Firefox

2.2 Grid tracks and lines

When grid was introduced it brought in new terminology to describe laying out items. The first concept is grid lines. Lines run horizontally and vertically and are numbered starting from 1 in the top left, on the opposite side to the positive numbers are the negative numbers.

Writing mode and script direction

The number assigned to each line is dependent on the writing mode (whether lines of text are laid out horizontally or vertically) and script direction of the component. For example, if the writing mode is English then the first line on the left is numbered 1. If the language direction was set to right-to-left, for example because the language was Arabic (written from right to left), then line 1 would be the furthest line to the right.

In-between the grid lines are the grid tracks that are made up of the columns and rows. Columns go from left to right whilst rows go from top to bottom. A track is the space between any two lines on a grid. In our example below (Figure 2.4) the track highlighted is our first row track in our grid. A column track would be the space between two vertical lines.

Then within each track are grid cells. A cell is the intersection of a grid row and a grid column.

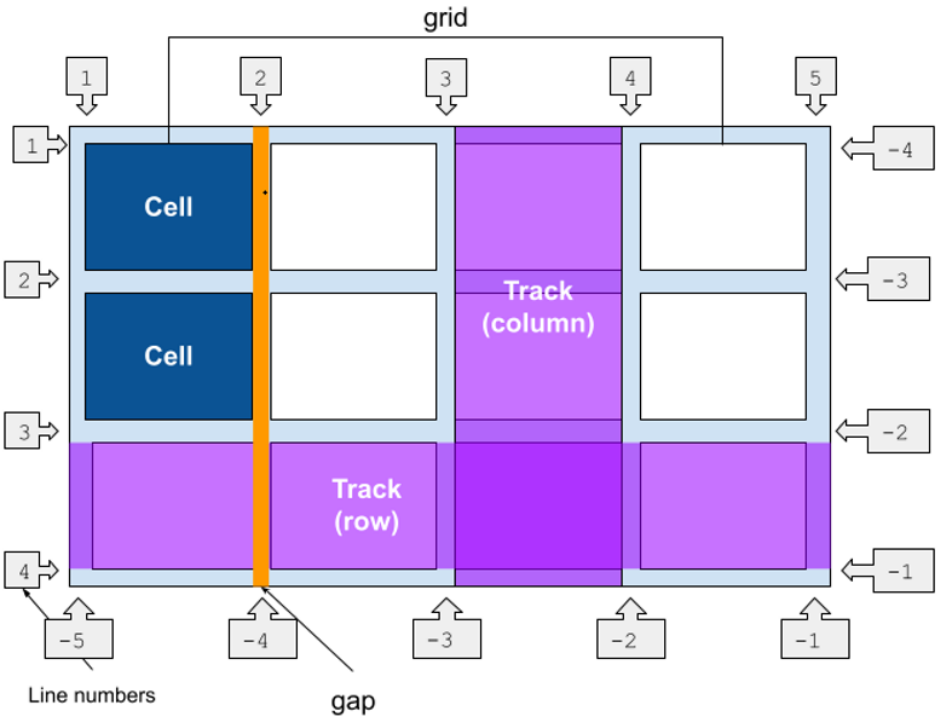

Figure 2.4 Grid structure based on writing mode being English with the direction set to left-to-right

The `grid-template-columns` and the `grid-template-rows` properties can be used to lay out your grid.

> **DEFINITION** These properties specify, as a space-separated track list, the line names and track sizing functions of the grid. The `grid-template-columns` property specifies the track list for the grid's columns, while `grid-template-rows` specifies the track list for the grid's rows.

Before we set our columns we first need to understand a few concepts that are specific to CSS Grids.

2.2.1 Repeating columns

To save repetition in your code you can use the `repeat()` function to specify how many columns you need.

The `repeat()` function needs two values that are comma separated:

1. the first value indicates how many columns to create
2. the second value is the sizing of each column

For our project we will specify that we want 2 columns and then for the sizing of each column we will use the minmax function. Our column definition will therefore looks as follows: `grid-template-columns: repeat(2, minmax(auto, 2fr)) 250px;`

This declaration will produce 3 columns, 2 of equal width using the fraction unit and one of 250 pixels. Let's dissect this declaration a bit further. Notice that inside the repeat function, we use another function `minmax()`.

2.2.2 Minmax

The `minmax(min, max)` function is made up of two arguments, the minimum and maximum range the grid track should be. The W3C specification states that the minmax function "defines a size range greater than or equal to min and less than or equal to max"[1].

Valid arguments

To make the function valid the min value (the first argument) needs to be smaller than the max, otherwise max gets ignored by the browser and the function will rely just on the min value.

For our project we set the minimum value to auto and the max to 2 fractions. Let's look at what `auto` means.

2.2.3 The auto keyword

The `auto` keyword can be used for either the minimum or maximum value within the function.

When the keyword `auto` is used for the maximum value then the minimum dimension the container will take is equal to the `max-content` keyword. To make layouts that include fixed headers and footers, we can assign overflow to the area for whom auto was assigned. The track that has a size value of auto will then shrink and grow with the window size as seen in Figure 2.5.

[1] W3C. "CSS Grid Layout Module Level 2" W3C, https://www.w3.org/TR/css-grid-2/. Accessed 18 June 2022.

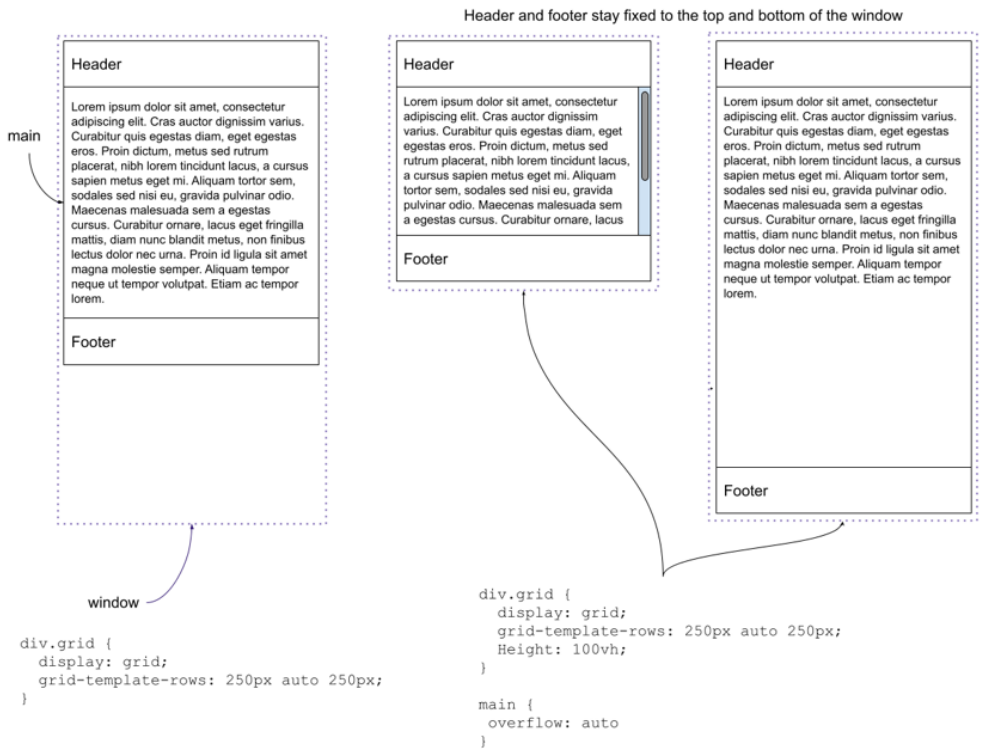

Figure 2.5 Using the auto keyword

For our use case, in the statement `grid-template-columns: repeat(2, minmax(auto, 2fr)) 250px;` the auto keyword dictates that for our first 2 columns, the column should be, at a minimum, as wide as the element it contains.

Let's take a look at the Flexible Length unit (`fr`) used to set their maximum width.

2.2.4 Flexible length

The fractions (`fr`) unit was introduced in the CSS Grid Module. The fractions unit is unique to grid as it is a way to tell the browser how much room a HTML element should have compared to other elements by distributing the left over space once minimums have been applied.

Let's first explore what a fraction is through some tasty cake diagrams (Figure 2.6) (sorry if this makes you crave a slice of cake). If you had a whole cake it would equal 100%. If we decided to eat all of the cake that would be 1 fraction. In our CSS that would be the same as `grid-template-columns: 1fr` which would be 100% of the column.

But we are friendly and decide we will give out some of our cake to our 4 friends, so we need to determine how many slices of our cake each person is going to have.

If we were fair we could say that our cake can be divided into 4 equal slices. In our CSS this would be the same as `grid-template-columns: 1fr 1fr 1fr 1fr`. We are telling the browser to give each HTML element an equal slice of the whole thing.

But what if we decided to be sneaky and have a larger size? After all, we baked the cake, right?

So we decide to take half of the cake for ourselves, and then divide the remaining half into three slices. To do this we need to have a total of 6 fractions, 3 fractions for our 50% of the cake and then one fraction three times to equally split the other 50% of the cake.

So in our CSS that would be `grid-template-columns: 3fr 1fr 1fr 1fr`, so we are saying the first column should have 50% and then the remaining 50% should be equally split between the other 3 columns.

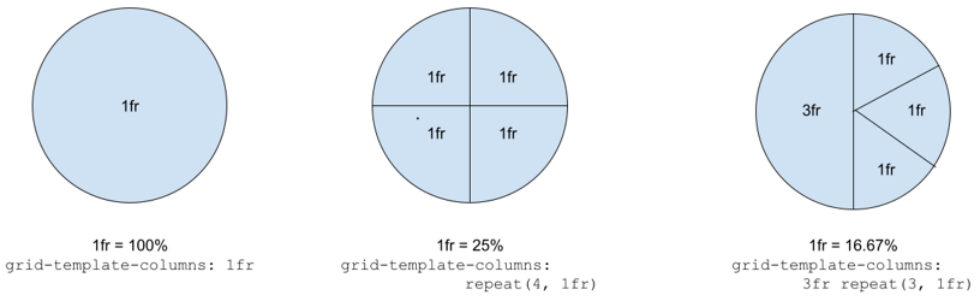

Figure 2.6 Diagram showing fractions value

For our project we will create our grid lines for the columns by adding the following to our `main` rule:

Listing 2.4 Setting the amount of columns

```
main {
  display: grid;
  grid-template-columns: repeat(2, minmax(auto, 2fr)) 250px
}
```

When previewed in the browser (Figure 2.7) we can see that now our grid has numbers set across each line. What we can do with this information is explicitly choose where to place our HTML elements within the grid based on the grid line number.

We also notice that the browser has assumed that we want to place our HTML elements within each grid cell so rather than each element being stacked vertically the browser fills up

each column cell until it runs out and then creates a new row and fills in those columns. This is also known as the implicit grid.

Explicit vs implicit grid

When we use either `grid-template-columns` **or** `grid-template-rows` **we create an explicit grid, we are clearly stating to the browser the exact amount of columns and rows this grid should have. The implicit parts (both for the rows and columns) are those that are automatically created by the browser, this can happen when there are more child items than grid cells, the browser will then implicitly extra cells to. We can control implicit behavior through** `grid-auto-flow`, `grid-auto-columns`, **and** `grid-auto-rows`

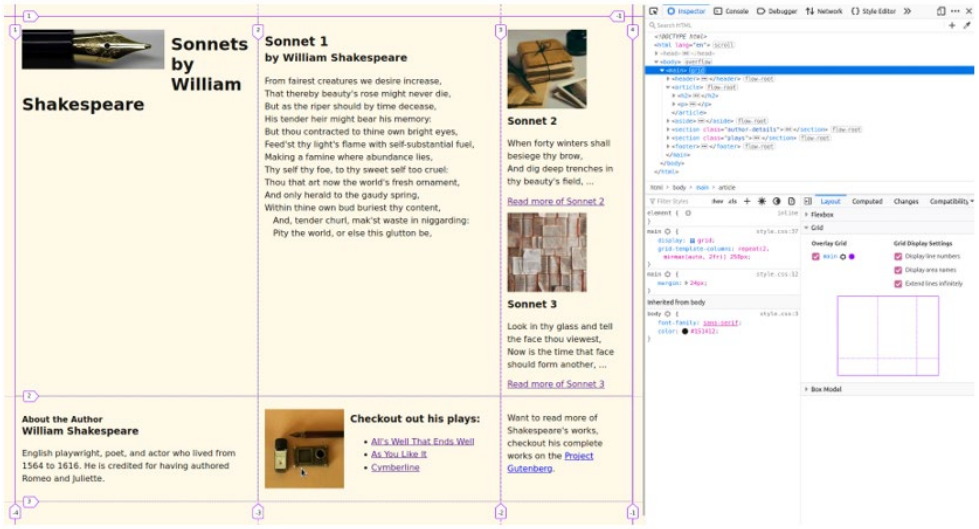

Figure 2.7 Firefox browser preview showing the grid lines and the associated numbers for each line

At this juncture, we have created a grid containing 3 columns. Two of those columns use `minmax()` and then our third column has a fixed value of 250px. This will allow us to have a three column layout. We will want to distribute the main content in the first 2 columns and use the 3rd one as an aside to put other less important content hence it being given less visual real estate (on most screens it will be narrower than the first 2 columns).

2.3 Grid template areas

If we want to explicitly set an element on a particular row and column of our grid, we have 2 different options. First, we can use the line numbers and dictate the position of the child as follows: `grid-column: 1 / 4`. This would place the element in the first column spanning through the second and third. Remember than grid is all based on lines, and the first line is

the far left edge so the 4th line is the right edge of the grid. In our example, the element would be full width. To define the row we would use the same syntax as for columns with the `grid-row` property. To define that an element should start on the 3rd row and span 2 rows we would write: `grid-row: 3 / 5`.

Rather than dealing with numbers, we can instead use named areas to be referenced when explicitly placing elements on the grid. To do this we use the `grid-template-areas` property. This allows us to define the guide for how we want the web page to be laid out.

The `grid-template-areas` property takes multiple strings, each composed of the names of the areas they describe. Each string represents a row in the layout as seen in Figure 2.8. Each name represents a column within the row. If 2 adjacent cells have the same name (horizontally or vertically) the two cells are treated as one area.

```
grid-template-areas:
  "header  header  header"
  "content content author"
  "content content aside "
  "plays       .   aside "
  "footer  footer  footer";
```

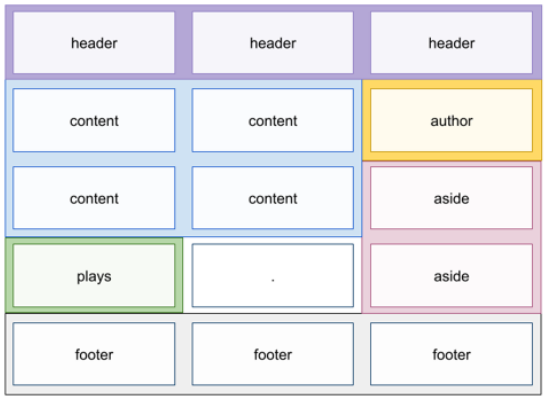

Figure 2.8 Syntax of the `grid-template-areas` property

The benefit of named areas is in the visualization of the final outcome. We will define our `grid-template-areas` as follows (Listing 2.6). Notice the dot (.) in the 4th row. The dot is used to define a cell that we intend to keep empty. Since it does not have a name, content cannot be assigned to it.

Listing 2.5 Creating our template areas with media queries

```
main {
  display: grid;
  grid-template-columns: repeat(2, minmax(auto, 2fr)) 250px;
  grid-template-areas:
  "header  header  header"
  "content content author"
  "content content aside "
  "plays       .   aside "
  "footer  footer  footer";
}
```

Although we have defined areas, the content is still implicitly positioning itself in each available cell, ignoring the areas we have defined (Figure 2.9). We now need to assign our content to each of these areas.

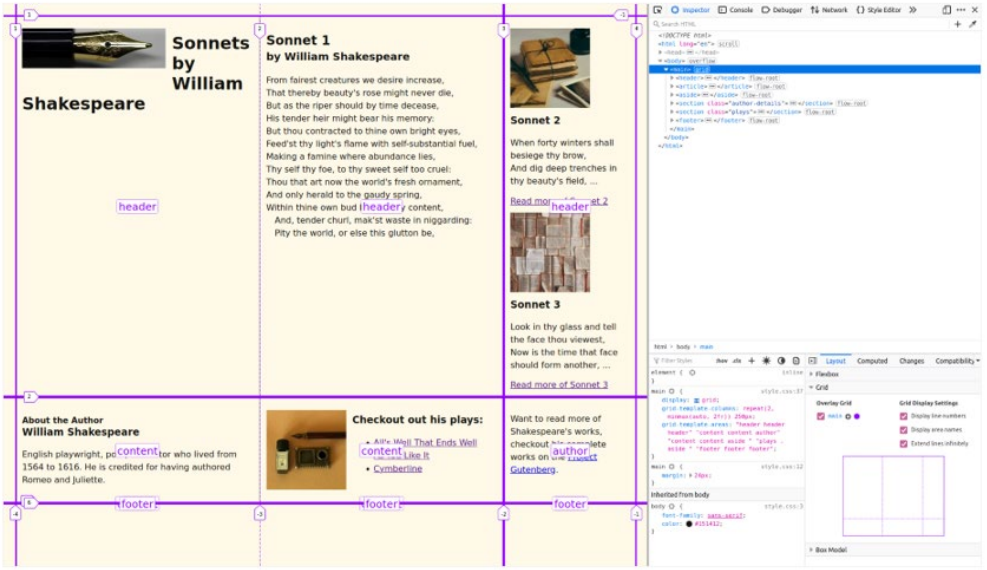

Figure 2.9 Defined grid areas (Firefox)

2.3.1 grid-area

To place an element in a defined area we use the `grid-area` property. The `grid-area` value is the name we assigned in the `grid-template-areas` property. For example, if we want the `<header>` element to be placed inside of the area we defined as `header`, we would define: `header { grid-area: header; }`. Applied to our project, we set our areas as follows:

Listing 2.6 Assigning content to grid areas

```
header { grid-area: header }
article { grid-area: content }
aside { grid-area: aside }
.author-details { grid-area: author }
.plays { grid-area: plays }
footer { grid-area: footer }
```

Now that we explicitly defined where the content should go, we can see that the content falls into place (Figure 2.10).

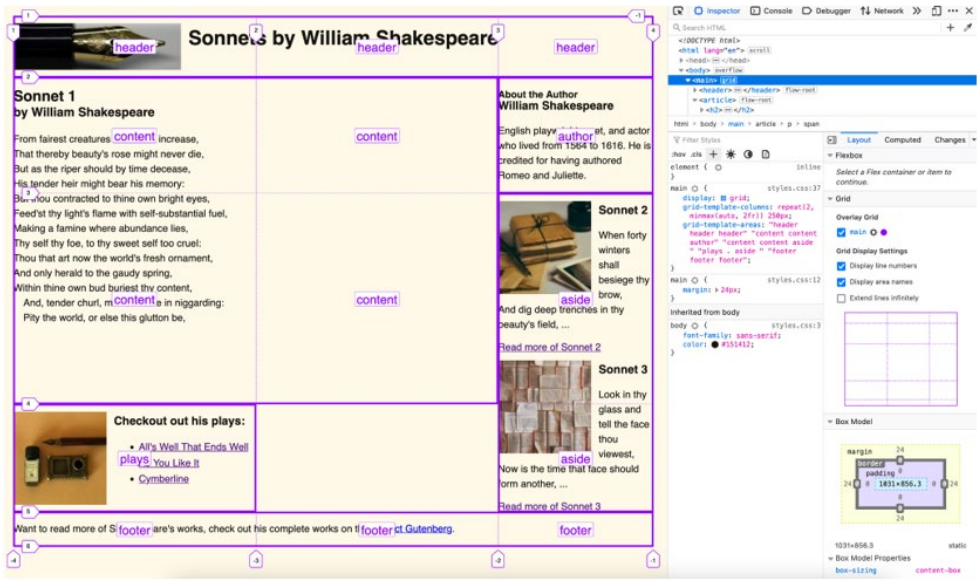

Figure 2.10 Content explicitly placed on the grid

With the layout setup, let's remove some of the styles we had added for the purpose of understanding what our layout was doing. As seen in Listing 2.7 we remove the padding and borders of our content sections.

Listing 2.7 Removing debugging styles

```
main > * {
  border: solid 1px #bfbfbf;
  padding: 12px;
}
```

With those styles removed and the screen width narrowed (Figure 2.11) we can see that the content in adjacent columns or rows appears really close together.

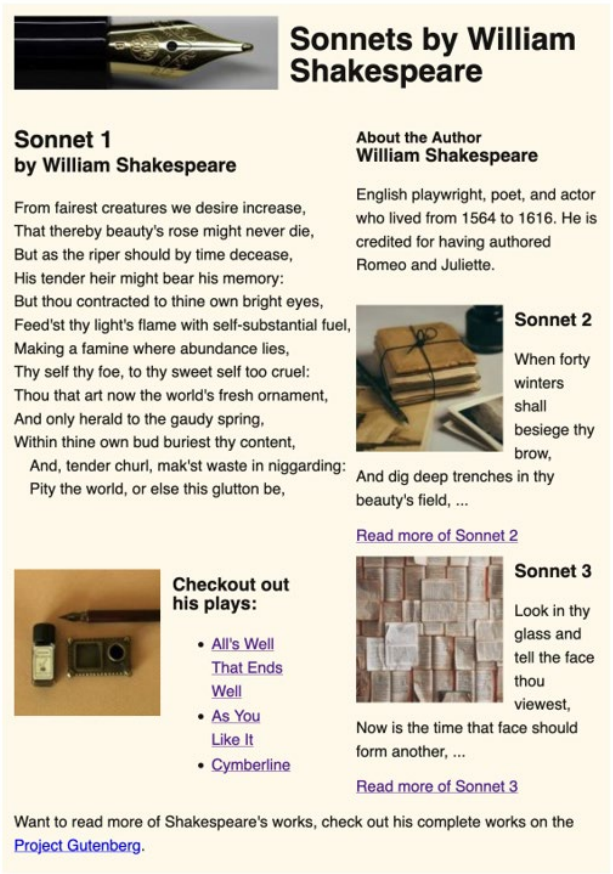

Figure 2.11 Narrow screens

Let's add space between the areas. To accomplish this, we will use the `gap` property.

2.4 Grid gap

The `gap` property is shorthand for `row-gap` and `column-gap` properties. By setting the row and column gaps we are defining the gutters between each row and column. Gutters is a term from print design, defining the margin between columns. By default the gap between columns and rows is 0.

When using the gap property, the extra space only applies between the tracks of the grid. No gutters are applied before the first track or after the last track. To set space around the grid we use padding and margin properties.

grid* vs *grid-gap

As CSS Grid was being defined, the specification for this property was called the `grid-gap` property but now it is recommended to use just `gap`. However, we still may see `grid-gap` in older projects.

The gap property can have up to two positive values:

1. the first sets the row gap
2. the second is for the column gap

If there is only one value declared it is applied to both the `row-gap` and the `column-gap` properties. For our project we will set a `20px` gap between our rows and columns by adding `gap: 20px` to our `main` rule.

Figure 2.12 shows our gaps added to our layout.

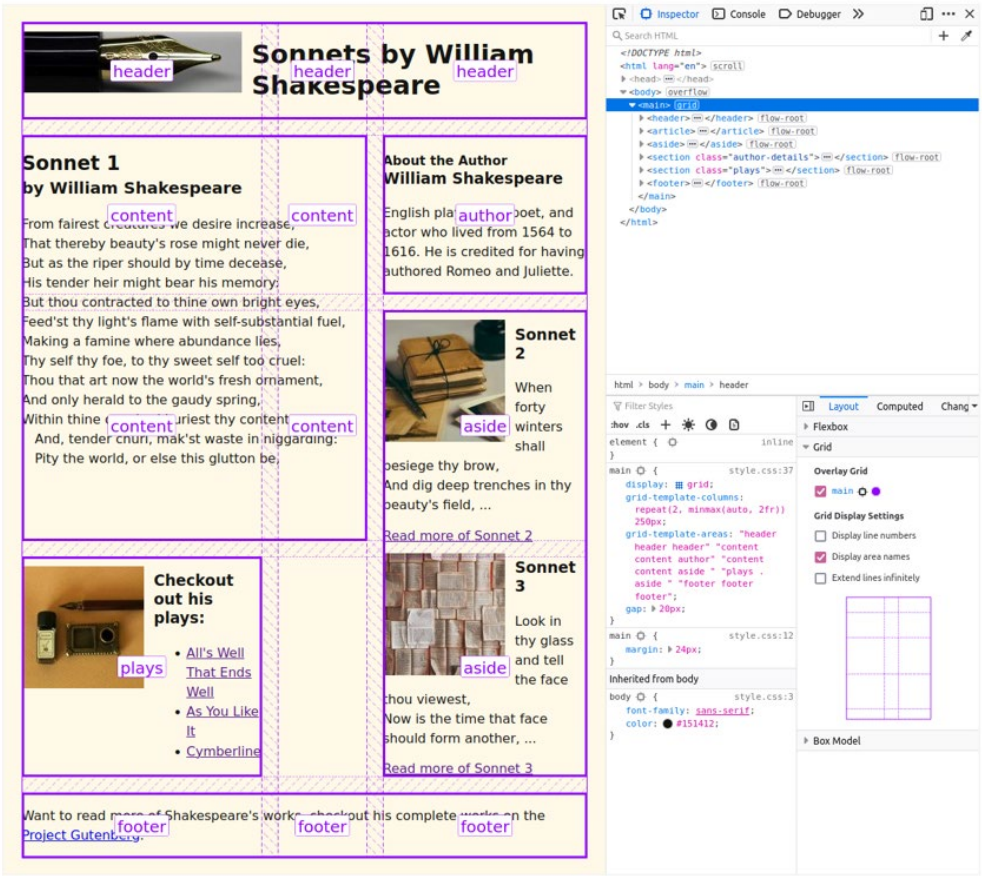

Figure 2.12 Grid layout with added gap

With the gaps added, let's switch our focus to adjusting our layout based on screen size.

2.5 Media Queries

CSS allows us to conditionally apply our styles to our layout based upon the screen size. Media queries are called at-rules because they start with the @ symbol. We then define the condition under which our styles should be applied.

Currently if we look at our layout on a wide screen (Figure 2.13), we notice we have a large amount of space in the center of the page that could be better utilized.

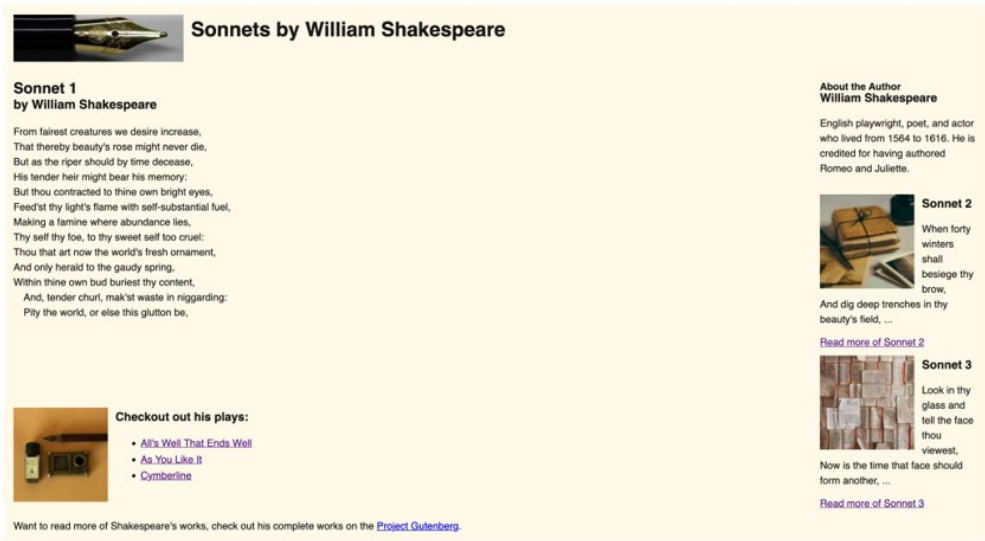

Figure 2.13 Our layout on a wide screen

Let's create a media query that targets screens wider than 955px. The query looks as follows: `@media (min-width: 955px) { }`. All of the rules we place inside of the curly brackets (`{}`) will only be applied if the screen size is greater than or equal to 955px. These styles will override the styles we had already assigned to the grid since by being inside of the media query, they have a higher level of specificity then our original layout.

Listing 2.8 shows our media query. After defining the query, we redefine our grid-template-areas to have a different configuration. We also update the column sizes to all have equal widths.

Listing 2.8 Creating our template areas with media queries

```
@media (min-width: 955px) { #A
  main {
    grid-template-columns: repeat(3, 1fr); #B
    grid-template-areas: #C
    "header  header  header" #C
    "content author aside"  #C
    "content plays aside " #C
    "footer  footer  footer"; #C
  }
}
```

#A media query
#B redefines the column sizes
#C reconfigures where the content should be placed

Our layout now looks as seen in Figure 2.14 and 2.15.

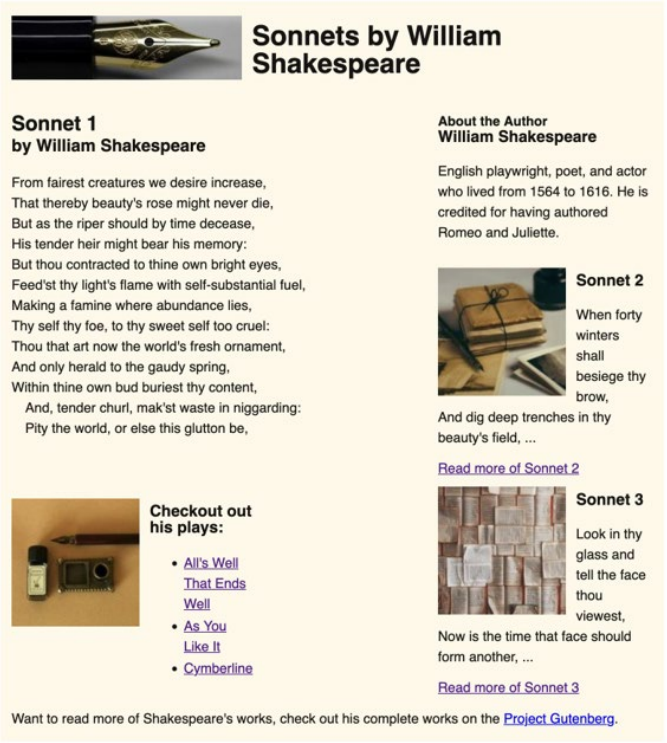

Figure 2.14 Narrow screen uses original layout

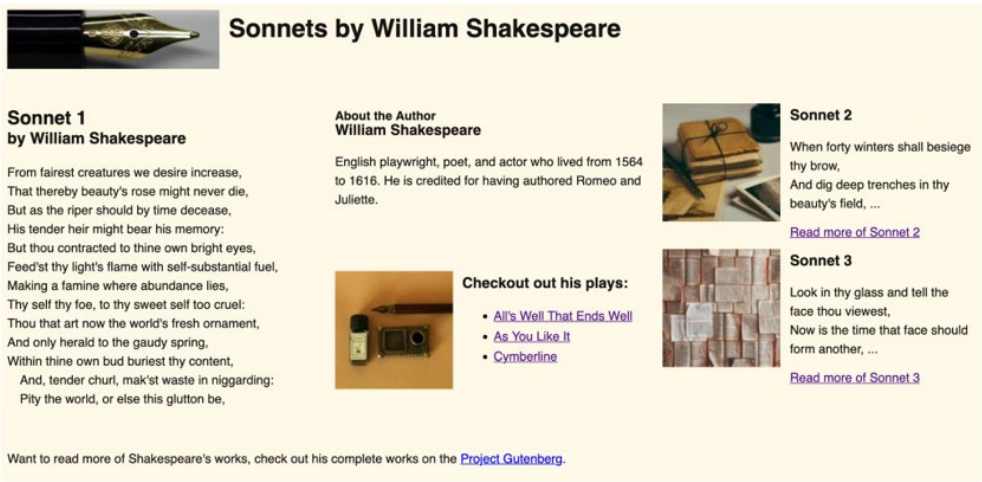

Figure 2.15 Wide screen using layout from media query

Using grid-template-areas in conjunction with media queries allows us to reconfigure our layout with minimal code. However, there are some accessibility pitfalls we must avoid.

2.6 Accessibility

When we placed our items in the grid area we mostly kept the elements in the order they appeared in the code: the header stayed at the top, footer remained at the bottom, and the content was also in a logical visual order. But what if the HTML code arrangement and the visual order were different from each other?

The problem is that if a user is following along with a screen reader or navigates the page via keyboard and the programmatic order does not match what is being displayed, the behavior they will be presented with will seem random. This will make it very difficult for the user to navigate and to comprehend what is going with the page. Visually changing the location of a piece of content using a grid will not affect the order in which assistive technology presents the information to the user.

W3 specification says the following about this case:

> **W3 RECOMMENDATION** Authors must use order and the grid-placement properties only for visual, not logical, reordering of content. Style sheets that use these features to perform logical reordering are non-conforming.

The solution is to keep the source code and the visual experience the same, or at least in a sensible order. This will give you both the most accessible web document and a good

structure to work from. For English this means content and HTML should follow the same order, from top left to bottom right.

After assigning our elements to their respective areas on the grid we should always test our page to ensure that regardless of how the user accesses the page, the order will still be logical. A couple of ways we can do this are by visiting our page using a screen reader and by tabbing through the keyboard to make sure the tab order still works.

There are tools and extensions that can help with visualizing tab order. As we are using Firefox for this, in the Firefox dev tools, for example, we can select the accessibility tab and check the "show tabbing order" checkbox, which will outline and number links and buttons as shown in Figure 2.16.

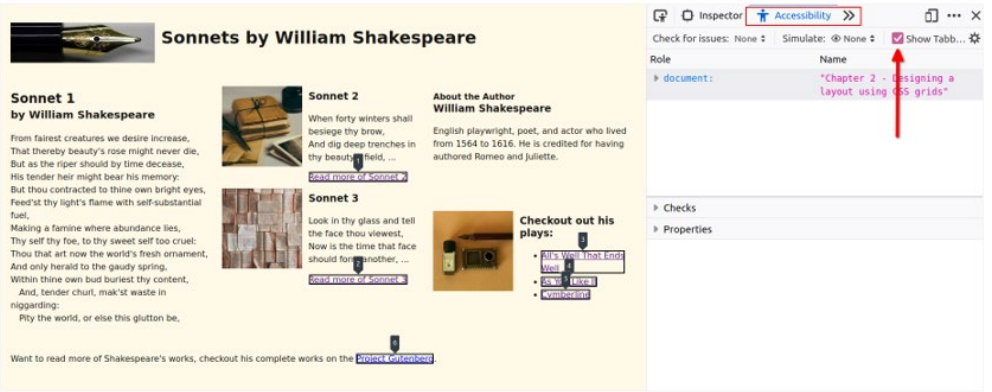

Figure 2.16 Tabbing order of HTML exposed using Firefox dev tools

We have now completed the project (Figure 2.17).

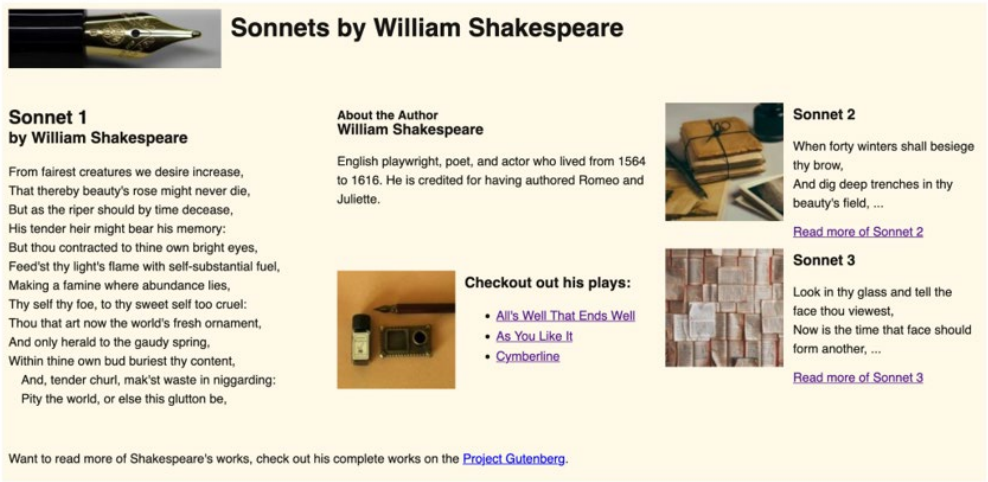

Figure 2.17 Final product on wide screen

We have used the CSS grid module to create a layout that is responsive depending on the browser width. Many aspects of the grid are still being developed and iterated on, most notably subgrids, which would allow for grids within grids. To keep an eye out for future enhancements and development check out the grid specification at https://www.w3.org/TR/css-grid-2/

2.7 Summary

- A grid, in this sense, is a network of lines that cross each other to form a series of squares or rectangles.
- The `display` property with a value of `grid` allows us to place items on a grid layout.
- The `display` property is applied to the parent item that contains the child elements that are to be placed in the grid
- `grid-template-columns` and `grid-template-rows` are used to explicitly define the quantity and size of the columns and rows the grid should contain
- The flexible length (`fr`) unit is a unit of measurement that was formed as part of CSS grids as an alternative way to set the dimension of items
- We can use the `repeat()` function as a way to improve code efficiency where one or more rows or columns are the same size
- The `minmax()` function allows us to set two arguments, the minimum width a column should be and the maximum width a column should be.
- `grid-template-areas` allows us to define what each grid area is called, and then we can use the `grid-area` property on the child items to assign them to those named locations
- The `gap` property is a way to add spacing (create gutters) between grid cells

3

Creating a responsive animated loading screen

This chapter covers

- Creating a Scaleable Vector Graphic (SVG) by creating basic shapes in SVGs to create our loader
- Finding out the difference between viewbox and viewport in SVG
- What keyframes are and how we can animate SVG graphics from one size to another
- Using the animation property to set the duration, style, how many times it should animate and how to delay when animations should start
- How to style SVG graphics using CSS
- How to style an HTML progress bar element with vendor prefixes

We see loaders in most web and mobile applications today. These loaders are used as a way to communicate to the user something is loading, uploading, or waiting for something. It gives the user confidence that something is happening and that the application is not frozen.

Without some sort of indicator to tell the user something is happening the user may try reloading, going back and clicking the link again, or just give up and leave. We should be using some sort of progress indicator when an action takes longer than 1 second, this is when users tend to lose focus and question if there is a problem. As well as having a visual graphic showing something is happening, the loader should be accompanied with text that tells the user what is happening.

For our animation, we'll be looking into the CSS animation module, understanding the animation property, keyframes, transitions as well as accessibility and how we respect our user's preferences regarding how much animation they are presented with.

We'll be creating the rectangles using an SVG graphic and take a look at what SVGs offer us to understand the slight differences that exist in CSS properties between styling HTML elements and SVG elements.

For the progress bar, we will use the HTML `progress` element and then look at how we can remove the browser's default styles and apply our own. Overall we want to create a consistent, responsive loader that can work across devices. The end result we will achieve is shown in figure 3.1.

The code for the before and after for this project can be found in the GitHub repository: https://github.com/michaelgearon/Tiny-CSS-Projects

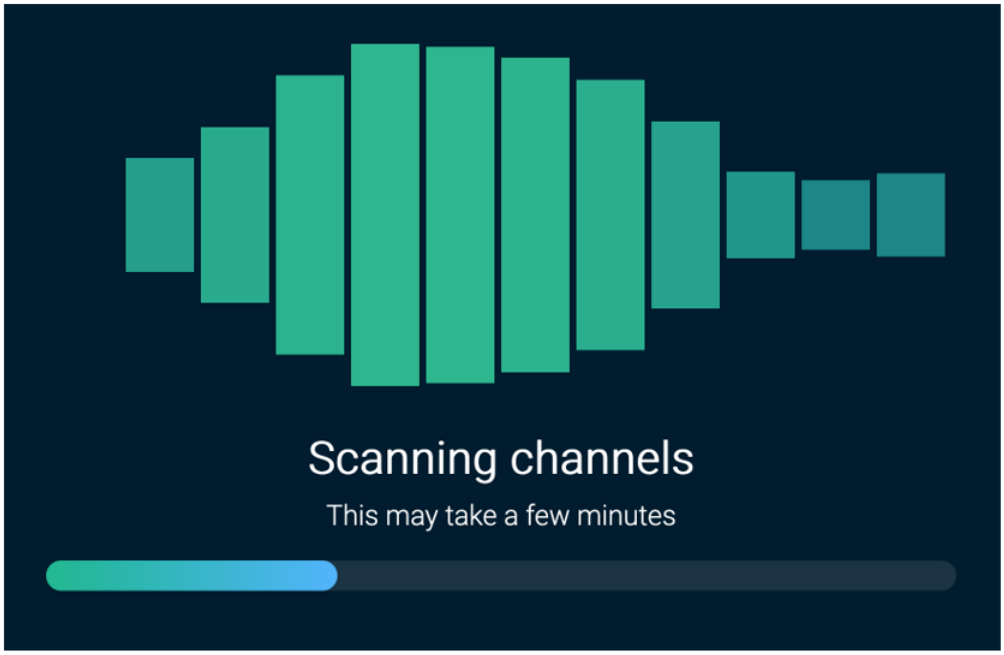

Figure 3.1: End goal for this chapter

3.1 SVG and graphics

SVG stands for Scalable Vector Graphics. SVGs are written using an XML-based markup language and consists of vectors on a Cartesian plane.

A vector is a mathematical formula that defines a geometric primitive. Lines, polygons, curves, circles, and rectangles are all examples of geometric primitives.

A Cartesian coordinate system in a plane is a grid based system that defines a point using a pair of numerical coordinates based on its distance from 2 perpendicular axes. Where these

two axes cross is the origin, which has a coordinate value of (0, 0). Think back to math class, when we were being asked to plot lines on a graph, that was a cartesian coordinate system. Essentially, SVGs are shapes on a coordinate plane written in XML.

PNGs, JPEGs, and GIFs, by contrast, are raster images. Raster images are created using a grid of pixels. Figure 3.2 illustrates the difference between raster and vector graphics .

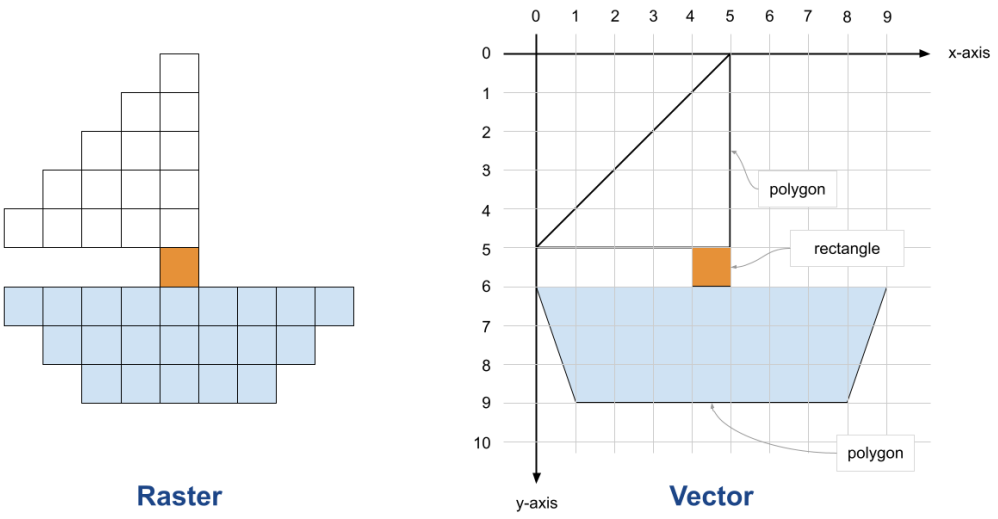

Figure 3.2: Raster Versus Vector Graphics

By virtue of their construction, SVGs have many advantages over raster images including:

- Being infinitely scalable. We can shrink or enlarge the image as much as we want. We can shrink raster images, but we cannot enlarge them without seeing pixelation which results from enlarging the grid of pixels rendering the individual squares of the grid visible. When we enlarge an SVG, since we are programatically setting shapes and lines on a coordinate plane, the paths between points are redrawn and the quality does not degrade.
- Because SVGs are written in XML, we can place SVG code directly into our HTML and access it, manipulate it, edit it in much the same way we do our other HTML elements. SVGs are to graphics as HTML is to text.

Where rasters are a better choice however, is when dealing with images that are highly complex or photo realistic. Not that it wouldn't be possible to create a photo realistic image using an SVG, it would not be practical. The file size and therefore performance to load the image would be significantly larger when creating it using vector graphics than as a raster image.

The most common use case for SVGs are logos, icons, and loaders. We use them for logos because they are often simple images that need to be crisp regardless of the size or medium. Furthermore, it is not uncommon for a company or product to have several versions of a logo for usage on a dark background versus a light background for example. Recoloring, but also simplicity, and scaling, are also the reasons we use them for icons.

We use SVGs for loaders because, unlike their raster counterparts, we can add animations inside the image itself. We can isolate individual elements inside the graphics and apply CSS or JavaScript to that individual piece, an exercise which is not possible with rasters.

Earlier we mentioned that SVGs are based on a Cartesian plane (a 2 dimensional coordinate plane), let's dig further into what that means and how it works.

3.1.1 Positions

When working with SVG elements, the way to think about positioning is to imagine we are placing elements on a grid. Everything starts at 0,0 (the origin) which is the very top, left corner of the SVG document.

The higher the X or Y value the further away it will be from that top left corner. The position is measured in pixels. Figure 3.3 expands on the example of the boat shown in Figure 3.2 adding the origin and the coordinate values for each shape.

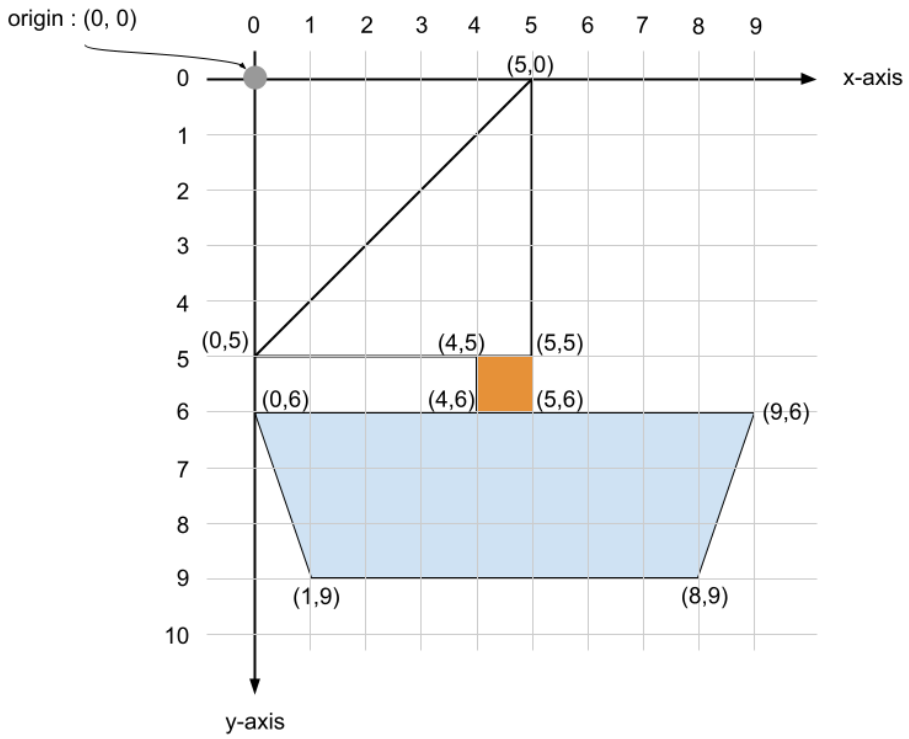

Figure 3.3: Positioning elements on a coordinate plane

Our loader is composed of a series of 11 rectangles. To place them we will need to think of their positions on a coordinate plane taking both their widths and the gaps between them into consideration.

3.1.2 Viewport

The viewport is the area in which the user can see the SVG. It is set by two attributes, width and height. The way to think of the viewport is that it's a picture frame, it sets the size of the frame. It does not affect the size of the graphic it contains however, if we place an image inside of a picture frame that is larger than our frame, then we will have overflow. The same will happen to our SVG. Like positioning, the viewport measurements will have their origin in the top left hand corner of the SVG. The diagram in Figure 3.4 illustrates this concept.

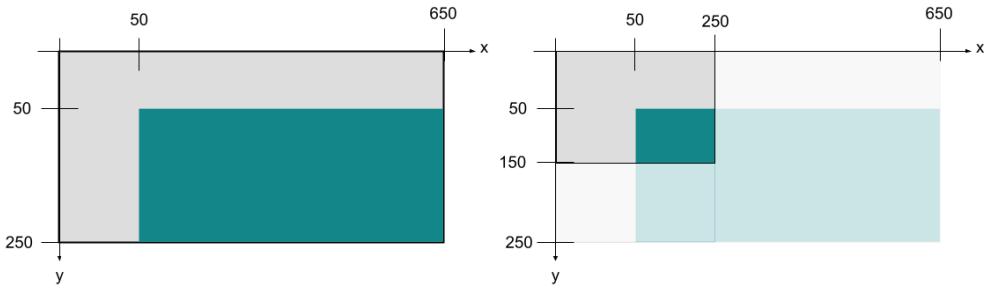

Figure 3.4: SVGs with and without a defined viewport

The viewport for our loader will be:

```
<svg width="100%" height="300px"> <!--SVG code --> </svg>
```

The width is set at 100%, but 100% of what? We are dictating that the loader will take 100% of the available space it is given by its parent item. Listing 3.1 shows our starting HTML. We see that our loader is nested inside of a section, therefore our loader will be the same width as our section.

Listing 3.1: Starting HTML

```
<body>
  <section>
    <svg width=”100%” height=”300px”></svg> #A
    <h1>Scanning channels</h1>
    <p>This may take a few minutes</p>
    <progress id="file" width="100%" value="32" max="100">32%</progress> #B
  </section>
</body>
```

#A Loader with added viewport of 100% width by 300 pixel height
#B The progress bar we will address later in the chapter

We have some starting CSS as well (Listing 3.2), the background, section, header, and paragraph have been pre styled in order to focus on the loader, progress bar, and animations.

Listing 3.2: Starting CSS

```
body { background: rgb(0 28 47); }

section {                                #A
  display: flex;                         #B
  flex-direction: column;                #B
  justify-content: space-between;        #B
  align-items: center;                   #B
  max-width: 800px;                      #B
  margin: 40px auto;                     #C
  font: 300 100% 'Roboto', sans-serif;   #D
  text-align: center;                    #D
  color: rgb(255 255 255);               #E
}                                        #F

h1 {
  font-size: 4.5vw;
  margin: 40px 0 12px;
}

p {
  font-size: 2.8vw;
  margin-top: 0;
}
```

#A Start of rule styling the loader's container
#B layout
#C Margin written using the shorthand property. Top and bottom: 40px margin, left and right: auto.
#D Typography
#E Color
#F End of rule styling the loader's container

We can see that our section has its width capped at 800 pixels. `<section>` is a box-level element, therefore by default, it will take up the full width available to it. `<body>` and `<html>` are also box-level elements.

Since we don't specify a width, padding, or margin on either <body> or <html>, they will take the full width of the window. <section>, will take the full width of the <body>. However we assigned a maximum width to the <section>, therefore once the window reaches a width of 800 pixels or wider, the section will stop growing with the <body>, and remain at 800 pixels in width. Because it has a top and bottom margin of auto, it will also horizontally center itself relative to the <body>.

Our loader is contained within the section, the section will take the full width of the body up until it reaches 800 pixels therefore our loader will do the same. Figure 3.5 diagrams how the width of the loader will be affected by the screen size.

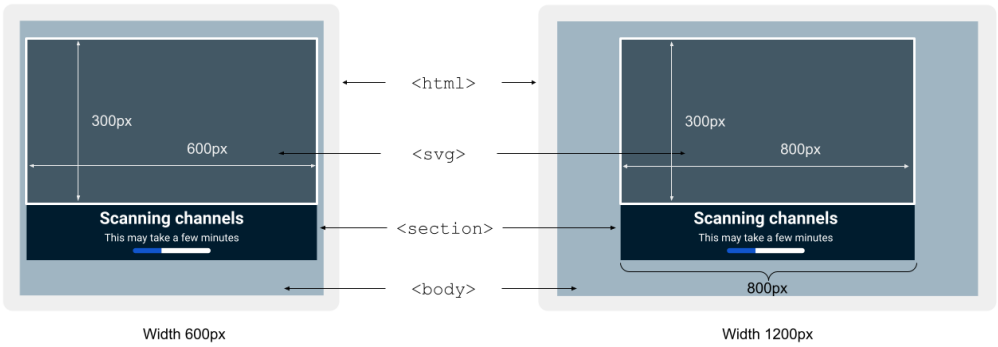

Figure 3.5: Window width effect on SVG width

With the viewport set, let's set the viewbox so that the contents of the SVG can scale with its container. Because remember, right now, we have only dealt with the frame, not the innards.

3.1.3 ViewBox

The viewbox sets the position, height, and width of the graphic within the viewbox. Earlier we compared the viewport to a picture frame, the viewbox allows us to adjust the image to fit our frame. It can position the image, and also scale the graphic so that it can fit inside of the frame. We can think of the viewbox as our pan and zoom tools. To set the viewbox, we apply the `viewBox` attribute to the SVG with the following 4 values and syntax: `viewBox="min-x min-y width height"`. Listing 3.3 shows the `viewBox` applied to our loader.

Dissecting the numbers in order, we start with min-x and min-y which are both set to 0. We want the top left corner of the graphic to be in the top left of our frame. Min-x and min-y allow us to adjust the position of the graphic in its frame; it's the pan tool. Since we want it exactly in the top left corner, we set the values to 0.

Next we apply the width. It is set to 710 because our loader has 11 total bars each composed with a width of 60. 60 multiplied by 11 is equal to 660, we also have 10 gaps. The gap width between each bar is equal to, `5 × 10 = 50` therefore our loader's width will be `660 + 50 = 710`.

We will also base the height of the viewBox on the height of the bars in our loader. They will have a height value of 300, we therefore also set the viewport height to 300. Our loader will therefore fit exactly inside of its viewport.

Listing 3.3: Declaring the viewbox

```
<svg viewBox="0 0 710 300" width="100%" height="300px">
 <!--SVG code-->
</svg>
```

Notice that both our viewbox and viewport heights are equal to 300. This is how we zoom. If the viewbox figures are less than the viewport then you are effectively zooming out to the frame, the graphic will be smaller. If the viewbox figures are more than the viewport then you are zooming in. Since we have equal viewport and viewbox heights, we are not zooming.

Now that we have defined the space we will be working in, we can start adding shapes to the loader.

3.1.4 Shapes in SVG

There are a few standard SVG shapes/elements:

- `rect` (Rectangle)
- `circle`
- `ellipse`
- `line`
- `polyline`
- `pologyon`

If we want to create an irregular shape, we can also use path. But we will not need it for this loader. Most often, they are what we see when we look at the XML behind logos, icons and complex animation graphics. For our project, we will use the basic rectangle shape to create the wave.

To define our rectangles which will create the bars in our loader, we will use the `<rect>` element and add 4 properties: `height`, `width`, `x`, and `y`. The `x` and `y` attributes will determine the position of the top left corner of the rectangle relative to the top left hand corner of the SVG.

We will want to create 11 rectangles (seen in Listing 3.4) that have a width of 60, a height of 300 and then we will use the `x` attribute to move the rectangles across the graphic. We will start at 0 and increase the value by the width of our bar (60) plus an additional gap of 5. Each rectangle's `x` value will be 65 more than the previous. Our 11th rectangle should have an `x` value of 650.

Listing 3.4: 11 rectangles

```
<svg viewBox="0 0 710 300" width="100%" height="300">
    <rect width="60" height="300" x="0" />
    <rect width="60" height="300" x="65" />
    <rect width="60" height="300" x="130" />
    <rect width="60" height="300" x="195"/>
    <rect width="60" height="300" x="260"/>
    <rect width="60" height="300" x="325"/>
    <rect width="60" height="300" x="390"/>
    <rect width="60" height="300" x="455"/>
    <rect width="60" height="300" x="520"/>
    <rect width="60" height="300" x="585"/>
    <rect width="60" height="300" x="650"/>
</svg>
```

We now have our rectangles positioned inside of our viewport and they are being correctly resized as we increase and decrease the window size by our viewBox. Figure 3.6 shows our SVG at different window sizes. (A white border has been added to the SVG and bars in order to make them more visible in the screenshots). The contents shrink and grow within their available space, without skewing the rectangles they contain as the width to height ratio changes with the resizing of the window.

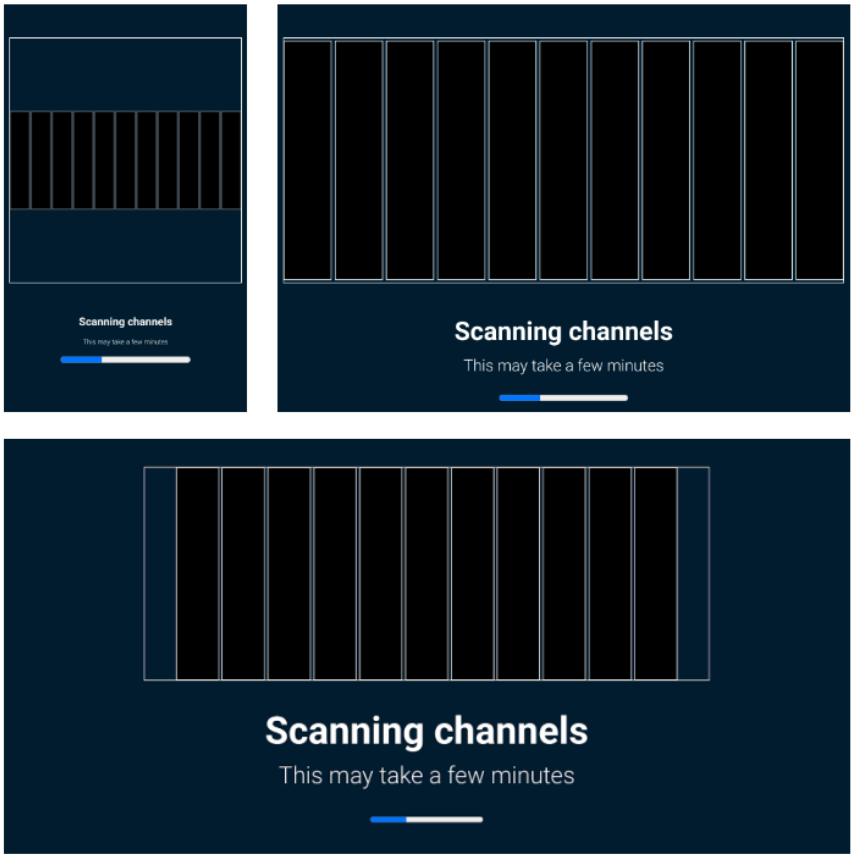

Figure 3.6: Added 11 rectangles inside of an SVG

Notice that our rectangles are black however, our next order of business will be to style them.

3.2 Applying styles to SVGs

We can apply styles to SVG elements in much the same way we do to HTML, inline, internally in a `<style>` tag, or in a separate stylesheet. However there are some minor differences.

First and foremost, how the SVG is being imported into our HTML affects where the styles need to live in order to affect the elements.

The easiest way to add a vector graphic to a web page is via an image tag. We reference the image file the same way we would any other image: `<img src="myImage.svg" alt="">`. We can also add them as a `background-image` inside of our CSS: `background-image: url("myImage.svg");`

In both of these cases our HTML and styles can affect the SVG itself but not the elements within. We can affect the size of the image for example but we cannot change the color of a particular shape contained inside the SVG. The image is essentially a black box we cannot penetrate to affect changes. In order to manipulate elements within it, the styles would need to be placed inside of the SVG itself.

Our third option, and the one we are using in this chapter is to place the SVG's XML inline, directly in our HTML rather than in an external file preventing the black box issue we would encounter if the code was in an external file. The drawback is that our concerns are not as well separated because our image code is now mixed in with our HTML.

With our SVG placed inline in our HTML, the standard ways we apply CSS to any other HTML element apply. We can therefore place the styles we want to apply to our SVG inside of our CSS as it was any other HTML element.

SVG Presentation Attributes

Unlike HTML where when applying styles inline we need to include a style attribute (ex: `<p style="background: blue">`), SVGs have a number of styles that can be added directly to the element as attributes. These are called Presentation Attributes. `fill` for example, (the SVG equivalent to `background-color`) can be applied directly to the element without a style tag: `<rect fill="blue">`.

These properties do not have to be applied inline directly on the element. They can be added inside of a style tag or stylesheet the same way we apply any other CSS style: `rect { fill: blue: }`.

A comprehensive list of SVG Presentation Attributes can be found at https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/Presentation

Although the techniques to apply styles to our SVG elements remain the same as if they were HTML (except for the aforementioned SVG presentation attributes when applied inline) some of the properties we will use to style our elements will be different. Let's take a closer look at the one we will be using for this project.

3.2.1 Fill

To set the background color of the loader bars, instead of using `background-colol`, we will use the `fill` property, as the `background-color` property does not work for SVG elements. The fill property still supports the same values as the background-color such as color name, RGB(a), HSL(a) and hex. So instead of rect `{ background-color: blue; }` we would instead write `rect { fill: blue; }`.

If no `fill` value is assigned to a particular shape, the `fill` will default to black, which is why our rectangles are currently black.

Let's go ahead and add the fill color to our rectangles. Because they aren't all the same color (they have varying colors of blue green to give the loader a bit of a gradient effect), rather than giving each element a class, we are going to use the pseudo-class `nth-of-type(n)` which selects elements based on their position relative to its siblings. We will look for the n^{th} rectangle, to which we will apply the fill. Therefore, `section rect:nth-of-type(3)`, would find the third rectangle of the section container.

REMEMBER A pseudo-class targets the state of an element. In this case its position relative to its siblings.

Listing 3.5 shows the fill color applied to each of our rectangles.

Listing 3.5: Adding a fill color to each of our rectangles

```
rect:nth-child(1) { fill: #1a9f8c }
rect:nth-child(2) { fill: #1eab8d }
rect:nth-child(3) { fill: #20b38e }
rect:nth-child(4) { fill: #22b78d }
rect:nth-child(5) { fill: #22b88e }
rect:nth-child(6) { fill: #21b48d }
rect:nth-child(7) { fill: #1eaf8e }
rect:nth-child(8) { fill: #1ca48d }
rect:nth-child(9) { fill: #17968b }
rect:nth-child(10) { fill: #128688 }
rect:nth-child(11) { fill: #128688 }
```

Figure 3.7 shows our output where we can see that each of the bars in the loader are no longer black and have their color applied to them.

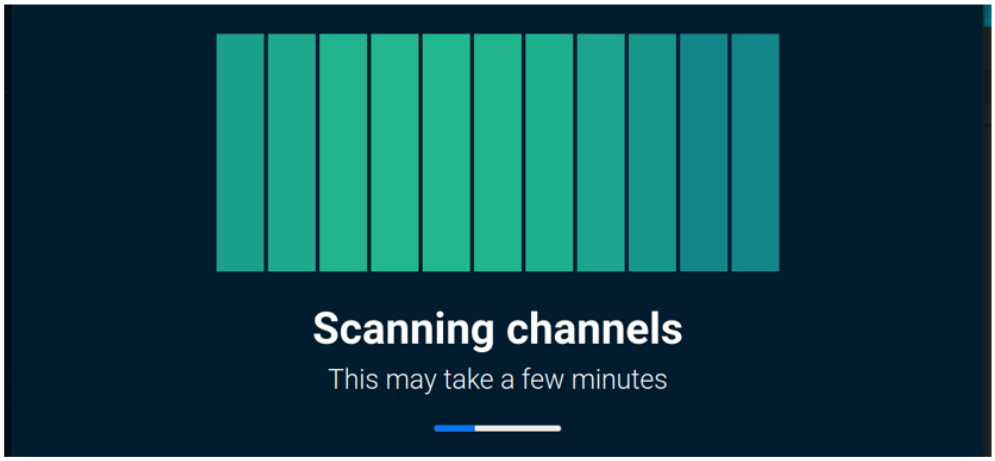

Figure 3.7: Fill applied to loader rectangles

3.3 Animation module

The CSS Animation Module allows us to animate properties using keyframes, which we will look at shortly. As well as the keyframes we can then control parts of the animation like how long the animation should last, and how many times it animates.

3.3.1 Animation property

The animation property is shorthand for the following properties:

- animation-name
- animation-timing-function
- animation-iteration-count
- animation-fill-mode
- animation-duration
- animation-delay
- animation-direction
- animation-play-state

For our animation we will focus on 4 of these properties:

1. `animation-name`
2. `animation-duration`
3. `animation-iteration-count`
4. `animation-delay`

The effect we want to create is each rectangle shrinking and growing over a period of time but not all in sync. At any given point in time we want the heights of the elements to be slightly different. When they are shrinking and growing, we want the top and bottom of the

rectangle to come towards the center and then expand back out to full height. Essentially, we will be creating a squeezing effect, going from large, to small, back to being large.

Although we will apply the same animation to all the rectangles, in order to stagger their sizes we will apply a slightly different delay to the start of the animation of each rectangle. As each rectangle starts animating at a different time each one will be in a different stage of expanding and shrinking creating a ripple wave effect.

Let's first look at how we create the animation itself. Then we will look at how to apply it to the rectangles. Finally we will add the individual delays to stagger the size at any given point in time. To create the animation, we will use keyframes. The animation property will then reference the keyframe and dictate the duration, delay, and how many times we want the animation to run.

3.3.2 Keyframes and Animation name

When we create our keyframes we need to give the keyframe a name. The animation-name declaration value matches the keyframe name to join the two together. With the animation-name property we can list multiple animations separated by a comma.

Keyframes come to us from the animation and motion picture industry. When companies used to do animation by hand they would compose many, individual pictures, and then, within each picture or frame, there would be a change. Over time we kept making changes to each frame and gradually we would get to the end frame. A simple example of this technique are flipbook animations. The more frames and subtler the tweek over a short period of time, the more fluid the animation becomes.

A keyframe represents the most important (key) changes in your animation (the frame). Then the browser works out all of the frames between each frame. This is known as in-betweening. Allowing the hardware to do the work it can then quickly fill in all of those between frames creating a smooth transition between one state and another. This concept is diagrammed out in Figure 3.8.

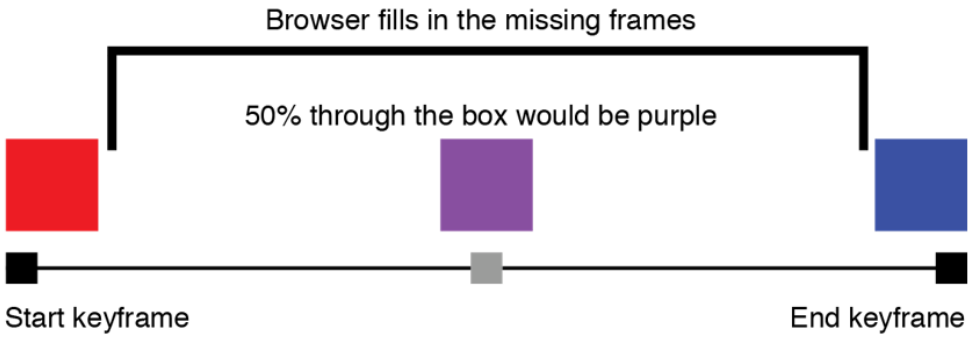

Figure 3.8: In-betweening

In CSS keyframe is an at-rule which controls the steps to happen within an animation sequence. At-rules are CSS statements that dictate how our styles should behave or when they should be applied. They begin with an @ symbol followed by an identifier (in our case keyframes). The syntax looks as follows `@keyframes animation-name { … }`. The code inside of the brackets define the animation's behavior.

Each step inside of the keyframe is defined by a percentage (percent of time passed in the animation) and the styles applied when we reach that point in time. Let's look at an example (Listing 3.6, this example can also be found on codepen at https://codepen.io/martine-dowden/pen/dydmWZO where we can see the animation run).

The example has 2 keyframes, one named `color` and one named `border-radius`. We have applied both of the animations to a `div`. We then defined how long the animation should take to run (3 seconds), and how many times it should run (10 times).

Inside of each keyframe is what styles should be applied to the elements. We notice we have 2 different types of notation, we have keywords, and we have percentages. Let's break down what we are defining in the first keyframe.

We assert that when the animation begins (0%), we want to set the background color of the `div` to `blue`. By the time we reach 50% of our animation (half of 3 seconds or 1.5 seconds), our background will be `yellow`. And when the animation ends (100%, or at 3 seconds), our background will be `red`. In between the steps we will see a smooth change from one state to the next.

In the second keyframe, `border-radius`, instead of percentages we use the keywords `from` and `to`. `from` is the equivalent of `0%` and `to` is equivalent to `100%`. We also note that we can mix the notation we want to use within the same keyframe.

Listing 3.6: Example Animation

```
@keyframes color {                #A
  0% { background: blue }         #A
  50% { background: yellow }      #A
  100% { background: red }        #A
}                                 #A

@keyframes border-radius {        #B
  from { border-radius: 0 }       #B
  50% { border-radius: 50% }      #B
  to { border-radius: 0 }         #B
}                                 #B

div {
  animation-name: color, border-radius;    #C
  animation-duration: 3s;                  #D
  animation-iteration-count: 10;           #E
}
```

#A First keyframe, named "color"
#B Second keyframe, name "border-radius"
#C `animation-name` property referencing both of the keyframes

#D Setting how long the animation should take to complete
#E Setting how many times the animation should run

When we apply the animation to the `div`, we also set a duration and iteration count. Notice that these 2 values are being applied to both of our animations. Before we take a closer look at these 2 properties and how they work, let's go ahead and create our keyframes for our loader.

For our loader, we want to grow and shrink, or scale, our rectangles over time. We will therefore call our keyframe `scale`. Our at-rule will look as follows: `@keyframes scale { }`.

Inside of the at-rule, we then define the steps (or frames) for the animation. We will start with the rectangle having its full height. Halfway through the animation, we want the height of the rectangle to be 20% of its original height. When the animation terminates, we want the rectangle's height to be back to full size. We will therefore have 3 steps to define: `from` (or 0%), 50%, and `to` (or 100%).

To change the size of the rectangle we will use the `transform` property. It allows us to change the appearance of an element (rotate, scale, distorted, move…) without affecting the elements around it. If we were to reduce the height of a `div` using the `height` property, the content below it would move up to fill in the newly available space. With `transform`, the amount of space and location of the element in terms of the page flow does not change, only its visible aspect. Using the same scenario, if we were to decrease the height of that same `div` using `transform`, the content below it would not move up. We would have a blank space.

To affect the element, the `transform` property takes a `transform-function`. We will use `scaleY()`. (A full list of available functions can be found here: https://developer.mozilla.org/en-US/docs/Web/CSS/transform-function).

The `scaleY()` function vertically resizes an element without affecting its height, skwishing or stretching it. To define how much an element should be scrunched or stretched, we pass the function either a percent or a number. The number value maps to the decimal value of its percentage equivalent therefore `scaleY(.5)` and `scaleY(50%)` achieve the same result of decreasing the element's height to 50% of its original value. Values above 100% increase the size of the element while values between 0 and 100% shrink it.

Negative values applied to `scaleY()` will flip the element vertically, so `scaleY(-0.5)` would flip the element upside down and shrink its height by 50%. `scaleY(-1.5)` will flip the element upside down and make the height 1.5 times the original value.

For our loader bars, we want our rectangles to be full height at the beginning and end of the animation, and 20 percent of the original height halfway through the animation. Our completed keyframe with transforms applied will therefore look as follows (Listing 3.7).

Listing 3.7: Completed keyframe

```
@keyframes scale {                #A
  from { transform: scaleY(1) }   #B
  50% { transform: scaleY(0.2) }  #C
  to { transform: scaleY(1) }     #D
}                                 #E

rect { animation-name: scale; }   #F
```

#A Start of the `scale` at-rule
#B Start the animation full height
#C Halfway through the animation, the height should be 20% of the original value
#D By the end of the animation, return the rectangle to full height
#E End of the `scale` at-rule
#F Applying the keyframe to the rectangles

If we run the code, we will notice that nothing has changed, our rectangles are not growing and shrinking yet even though we applied the keyframe to our rectangles. We still need to define the duration and iteration count. Let's dig into those properties a bit further.

3.3.3 Duration

The duration sets how long we want the animation to happen from start to finish. The duration can be set in seconds (`s`) or milliseconds (`ms`). The longer the duration the more slowly the animation takes to complete. With accessibility in mind, we want to consider users who have seizures and choose a duration that is reasonable.

Animations, Seizures, and Flash Rate

The W3C recommends that in order to prevent inducing seizures that are due to photosensitivity in our users, we need to make sure that our animation does not contain anything that flashes more than three times in any one second period[1].

A lot goes into choosing appropriate animation timing. An animation that is too fast can become imperceivable or cause seizure depending on its nature. An animation that is too slow can make our application look laggy. For small micro animations such as flipping an arrow from pointing up to pointing down when reordering a list, our duration should be short. Most micro animations are transitional. They animate the change of an element from one state to another. A generally accepted duration for this type of animation is around 250 milliseconds.

If the animation is larger or more complex, such as opening and closing a large panel or menu, then we can increase the duration up to around 500 milliseconds. A loader is a bit different though, it is not a quick passing change responding to a user's action, it's a large visual element the user will be focused on for a period of time.

[1] W3C. "Understanding Success Criterion 2.3.1 | Understanding WCAG 2.0." W3C, https://www.w3.org/TR/UNDERSTANDING-WCAG20/seizure-does-not-violate.html. Accessed 30 May 2022.

Most often when determining the "correct" timing for a loader, we will use trial and error to find a speed that works best with our graphic. For our project we want to set the animation to happen over 2.2 seconds. To apply the amount of time the animation should take, we add the `animation-duration` property to our rectangles as seen in Listing 3.8.

Listing 3.7: Added animation duration

```
rect {
  animation-name: scale;
  animation-duration: 2.2s;
}
```

When we run the code, we notice that our loader animates once and then never again unless we reload the browser window. We also notice that all of the bars increase and decrease in size at the same time. Let's first handle making our loader continue to animate over time and then we can stagger the animation across our rectangles so that they appear to be different heights.

3.3.4 Iteration count

To make our animation restart once it has completed we need to use the `iteration-count` property. It sets the number of times the animation should repeat. By default the value is 1. Since we didn't set a value yet, the browser assumed we wanted the animation to run once and be done. For our animation we want it to continuously repeat, to do this we use the `infinite` keyword value.

By applying this value, we are declaring that the animation should keep playing forever. If we wanted to run a specific number of times we would use an integer value. After adding our iteration count, our code is as follows (Listing 3.8).

Listing 3.8: Added animation iteration count

```
rect {
  animation-name: scale;
  animation-duration: 2.2s;
  animation-iteration-count: infinite;
}
```

When running the code we can see that all the rectangles grow and shrink in sync starting from the top and that the animation restarts itself once it completes. We still have some work to do in order to set the animation to start in the middle of the rectangle rather than the top and to stagger the animation between our elements but let's take a quick pause to look at the animation shorthand property.

3.3.5 Animation shorthand property

We currently have 3 declarations that define our animation: animation-name, animation-duration, and animation-iteration-count.

We can simplify our code by combining all three via the animation shorthand property. `animation: scale 2.2s infinite;` is functionally identical to the code we currently have applied to our rectangle. Using shorthand properties makes our code more concise and can make it easier to read, however if you find that writing out each property is easier for you, either method is perfectly valid. Do what works best for you.

Using the animation shorthand property our updated rect rule will be: `rect { animation: scale 2.2s infinite }`. After making the change to our code, we will notice that our animation has not changed.

Next let's address the staggering of heights for each of our rectangles.

3.3.6 Delay

The `animation-delay` property does what its name implies, it allows us to delay an animation on an element. The delay applies to the start of the animation. Once the animation starts, it will loop normally. Like duration, we can use seconds (`s`) or milliseconds (`ms`) to set the delay's duration value. The default value is 0 seconds. By default, an animation will not have a delay. To create the staggered effect in our animation, we will assign different delay values to each of our rectangles as seen in Listing 3.9. The first rectangle's animation will start immediately. We give it a delay of 0. We could omit this declaration entirely since 0 is the default value for animation-delay. It has been added here for clarity while explaining the code.

The second rectangle gets a 200 millisecond delay and we continue to increment the delay by 200 millisecond for every rectangle thereafter. Notice that on the 6th rectangle we switch to using seconds instead of milliseconds. We do this to make the code more readable since either second or millisecond values are acceptable.

Listing 3.8: Added animation iteration count

```
rect:nth-child(1) {
  fill: #1a9f8c;
  animation-delay: 0ms;
}

rect:nth-child(2) {
  fill: #1eab8d;
  animation-delay: 200ms;
}

rect:nth-child(3) {
  fill: #20b38e;
  animation-delay: 400ms;
}

rect:nth-child(4) {
  fill: #22b78d;
  animation-delay: 600ms;
}

rect:nth-child(5) {
  fill: #22b88e;
  animation-delay: 800ms;
}

rect:nth-child(6) {
  fill: #21b48d;
  animation-delay: 1s;
}

rect:nth-child(7) {
  fill: #1eaf8e;
  animation-delay: 1.2s;
}

rect:nth-child(8) {
  fill: #1ca48d;
  animation-delay: 1.4s;
}

rect:nth-child(9) {
  fill: #17968b;
  animation-delay: 1.6s;
}

rect:nth-child(10) {
  fill: #128688;
  animation-delay: 1.8s;
}

rect:nth-child(11) {
  fill: #128688;
  animation-delay: 2s;
}
```

After adding the delay, we can see (Figure 3.8) that we have achieved our staggered effect. However, the elements are growing and shrinking from the top rather than the center.

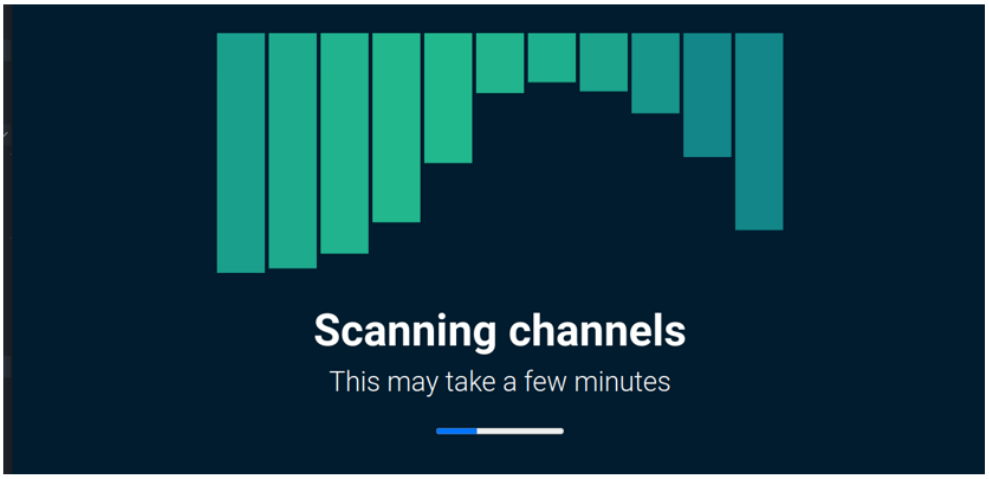

Figure 3.9: Animated rectangles with height change emanating from the top

To dictate where we want the element to grow and shrink from we need to tell the browser where on the rectangle the animation should originate from. To tackle this we will now look at the `transform-origin` property.

3.4 Transform origin

The `transform-origin` property sets the position the animation should start from (the point of origin). If the transform is happening in three-dimensions (3D) then the value can be up to three values (x, y, and z); when it's two-dimensions (2D) then it is up to 2 values (x and y).

There are 3 ways to declare the value of the transform-origin:

1. Length
2. Percentage
3. Keywords
    a. Top
    b. Right
    c. Bottom
    d. Left
    e. Center

The first value is the horizontal position or the X axis, where the second value is the vertical or the Y axis and then when working in 3D the third value would be the forwards and backwards or the Z axis.

In HTML, the initial value for this property is 50% 50% 0 so this is center, center, flat. However, for SVG elements the initial value is 0, 0 which would place it in the top left hand corner.

For our animation we want the rectangle transform origin to be at the center. We want the top and bottom of the rectangles to shrink rather than the top staying fixed in place and expanding and retracting from that point. To do this we can either apply the keyword value `center` or set a percentage value of `50%` to the `transform-origin` property for our rectangles.

Either way we are saying we want the point of origin to happen from the center of the rectangle. For our project, we will use the keyword value. Listing 3.9 shows our updated `rect` rule.

We mentioned earlier that when working with 2D animations, the property would take 2 values, but we only passed one. When only one value is passed, it will be applied to both the vertical and horizontal position, therefore `transform-origin: center;` is equivalent to `transform-origin: center center;`.

Listing 3.9: Updated rect rule with transform-origin property

```
rect {
  animation: scale 2.2s infinite;
  transform-origin: center;
}
```

We have finished our loader animation (Figure 3.10) however we still need to consider how accessible our design is.

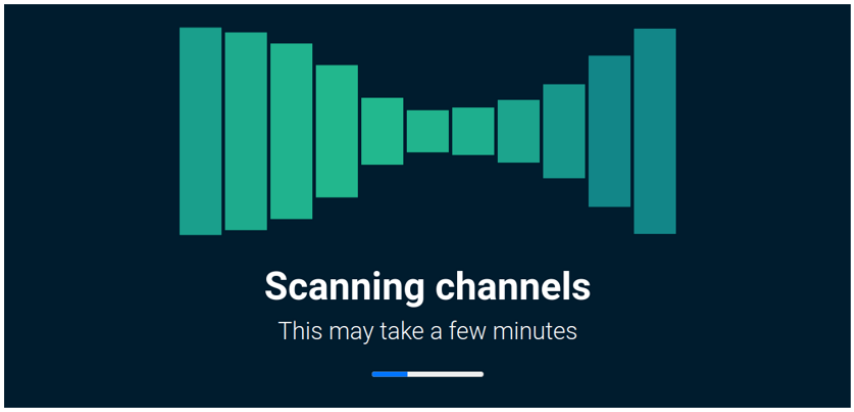

Figure 3.10: Finished Loader Animation

The next section will dive into some of the ways we can make sure to provide a positive experience for all of our users.

3.5 Accessibility and the reduced motion query

The use of motion, parallax (where the background moves slower than the foreground) and animations on the web has increased as it has become easier to implement and browser support has improved. By using these techniques we can create richer user interfaces that are interactive and provide greater depth in our experiences.

However, the use of these techniques come at a cost. For some users, especially individuals with vestibular disorders, movement on the screen can cause headaches, dizziness, and nausea. As we mentioned earlier, animations can also cause seizures, especially if they contain elements that flash.

In many operating systems users can choose to disable animations on their device. In our applications, we need to make sure that we respect those preferences. To check user settings the level 5 media queries module has introduced the `prefers-reduced-motion` media query. This is an at-rule, which checks the user's preferences regarding motion on the screen and allows us to apply conditional styles based on the user's preferences. There are two values for this query:

- no-preference
- reduce

We can either choose to disable an animation when a user prefers reduced motion, or to enable it when they have not specified a preference. The use of reduced motion by a user does not mean we cannot use any animation at all, but that we should be selective of which animations we should keep. Aspects that may determine which animations are kept enabled include:

- how fast it is
- how much of the viewport it uses
- flash rate
- how essential it is to the functioning of the site or understanding of the content

It is worth mentioning that a user may prefer reduced or no animations but may not be aware of the system preference settings to opt out of animations so providing an on site opt out button may be useful depending on how much animation our website has.

Accessibility Guidelines for Animations

A user should be able to pause, stop, or hide animation that lasts more than 3 seconds which are not considered essential. Loaders are a bit tricky when it comes to this as they convey important information to the user (the application is doing something, it is not frozen).[2]

[2] W3C. "Understanding Success Criterion 2.3.1 | Understanding WCAG 2.0." W3C, https://www.w3.org/TR/UNDERSTANDING-WCAG20/seizure-does-not-violate.html. Accessed 30 May 2022.

Our loader could be considered essential content but we also provide a progress bar below it which also gives the user an indication of what the application is doing. Because the information is also being conveyed below using a different media, and the animation is very large, has a lot of movement, and could last more than three seconds, we are going to disable it for users who prefer reduced motion using the code found in Listing 3.10.

Listing 3.10: Disabling the animation for users who prefer reduced motion

```
@media (prefers-reduced-motion: reduce) { #A
  rect { animation: none; }               #A #B
}                                         #A
```

#A Conditionally applies styles within the at-rule when the user has prefer-reduced-motion enabled
#B Disables the animation previously applied to the rectangles

To verify that we have successfully disabled the animation, instead of having to edit our machine's settings, we can go into our browser's developer tools and in the console tab display, select the rendering tab (if it is not already showing, use the vertical ellipses and select it from the menu by clicking the ellipses), and enable the reduced motion emulation. Figure 3.11 shows the disabled animation and developer tools in Chrome Version 102.0.5005.61.

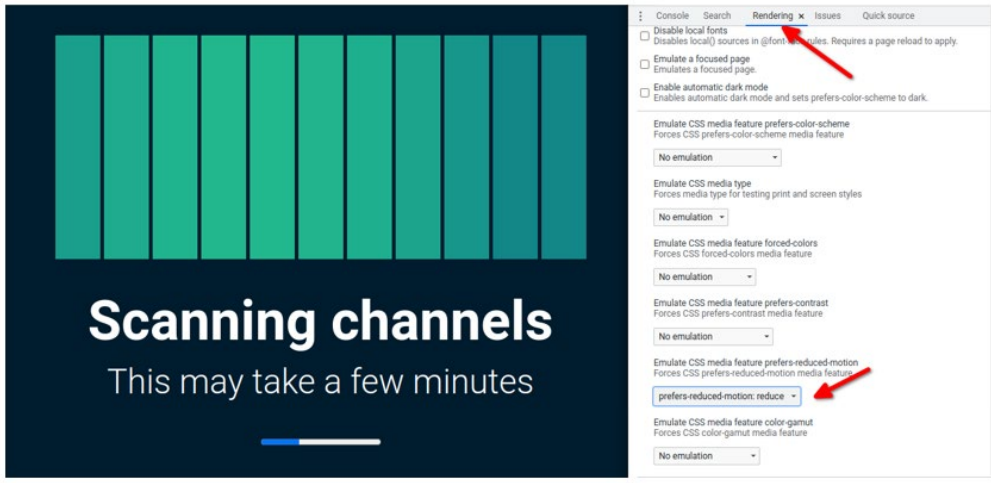

Figure 3.11: Emulating reduced motion preference using Chrome dev tools

With our loader animation finished, and accessibility needs handled, let's turn our attention to the progress bar at the bottom of the screen.

3.6 Styling an HTML progress bar

The `<progress>` HTML element can be used to show something is loading, uploading or that data is transferred. It's often used as a way to show the user how much of a task has been completed.

The default styles of the `progress` element will vary between browsers and operating systems. Much of the functionality of the progress bar is handled at the operating system level, as a result, we have few properties available to use to restyle the control especially when it comes to the colored progress indicator inside the bar itself.

We will look at some of the workarounds available to use and some of their pitfalls, but first let's start with what we can easily do.

Figure 3.12 shows our starting point generated by the HTML found in Listing 3.11. At this point, no styles have been applied to the control. Being shown are the defaults generated by the author's machine.

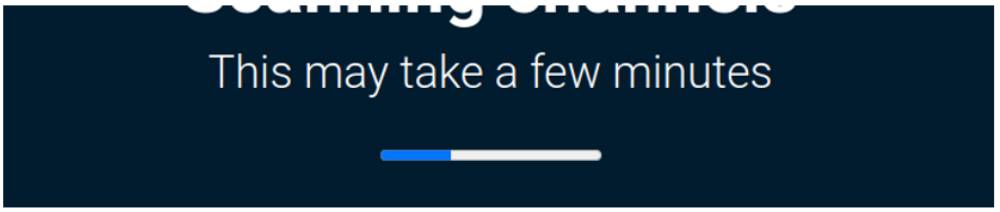

Figure 3.12: Progress bar starting point

Listing 3.11: Progress bar HTML

```
<body>
  <section>
    ...
    <progress value="32" max="100">32%</progress> #A
  </section>
```

#A The progress bar

3.6.1 Styling the progress bar

Let's start with changing the height and the width. To increase the width of the progress bar to match the width of the section, we will give its `width` property a value of 100%. We also want to increase the height to 24px.

To change the color of the progress indicator, the colored portion of the control, we can use a fairly new property: `accent-color`. This property allows us to change the color of form controls such as checkmarks, radio inputs, and the progress element. We will set it to `#128688`, matching the color of the last bar of our loader. Listing 3.12 shows our progress rule thus far.

Listing 3.12: Progress rule

```
progress {
  height: 24px;
  width: 100%;
  accent-color: #128688;
}
```

Figure 3.13 shows the Lising 3.12 styles applied to our control.

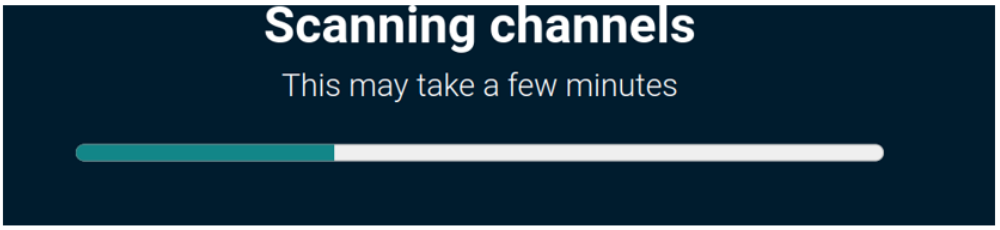

Figure 3.13: Width, height, and accent color applied to the progress element

If we try to add a background color to our element (`background: pink`), we will notice that it does not work. As a matter of fact, it fails quite spectacularly (Figure 3.14) as it radically changes the appearance of the element and alters the accent-color we had previously set. Furthermore, the background color changes to gray rather than pink.

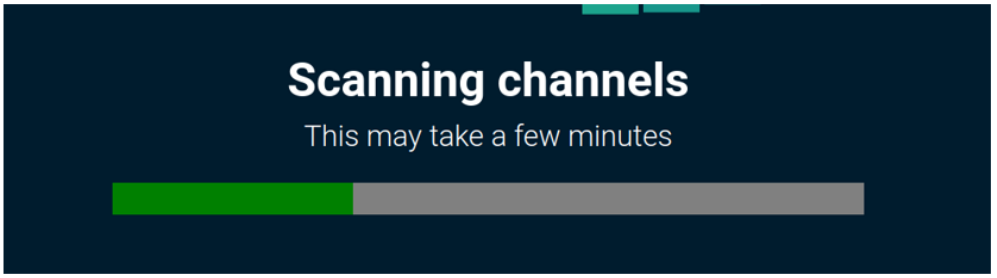

Figure 3.14: Background color failure

So how do we get around this problem? In order to restyle the control, we need to ignore the default, and recreate the styles from scratch. To do that though we will need to use vendor prefixed properties.

3.7 Vendor Prefixes

Historically when browsers were introducing new properties they would be added with a vendor prefix before the property name. Each browser's prefix is based on the rendering engine they use. Table 3.1 displays major browsers and their prefixes.

Table 3.1: Vendor prefixes and their browsers

Prefix	Browsers
-webkit-	Chrome, Safari, Opera, most IOS browsers (including Firefox for IOS), Edge
-moz-	Firefox

However, vendor prefixed properties were never meant, and still aren't fit, for production use. They are often incomplete or non-standard implementations that browsers may choose to either remove or completely refactor at any time. Although this fact has been clearly documented for years, developers, eager to use the latest available properties, regularly used them in production nonetheless.

To prevent this continued behavior most major browsers have moved to shipping experimental features behind a feature flag. To enable the feature and play with it, the user must go into their browser settings and enable that specific flag. By moving to a flag based method, the browsers are able to let developers play with experimental cutting edge features without fear that a non standard implementation might be leveraged in a piece of production code. However many vendor prefixed properties are still available in the wild.

The rest of this chapter covers how we could continue to style the progress bar, however, the code presented is not suitable for production use. We cover it, to show how a developer could test out an experimental feature that has a browser prefix. If we want a safe, stable, production worthy implementation, we should stop where we were before attempting to add the background color. To add a custom background color and continue to customize the control, we will need to use vendor prefixed properties.

The first thing we will do to fix our background color issue is remove the default appearance of the control.

3.7.1 Appearance property

To reset the appearance of the progress element we use the `appearance` property. By setting its value to `none`, we cancel out the default styles provided by the user agent. Since we will be creating all of the styles from scratch, we can also remove the accent-color property, as it will no longer have any effect. We will keep our height and width, and also add a border-radius since we are going to have a curved finish. Although the appearance property is supported by all major browsers, we will still need to include the vendor prefixed

version in order for the vendor prefixed values we add next to work. Listing 3.13 shows our updated rule.

Listing 3.13: Progress rule

```
progress {
  height: 24px;
  width: 100%;
  border-radius: 20px;
  -webkit-appearance: none;
  -moz-appearance: none;
  appearance: none;
}
```

At this juncture, our progress bar looks the same as when we broke it adding the background color. This is to be expected. With appearance none added, we can now start altering the control in ways we previously could not. We will first focus on browsers with a -webkit- prefix.

3.7.2 Styling the progress bar for -webkit- vendor prefixed browsers

There 3 vendor prefixed pseudo elements we can use to edit the styles of our progress bar:

- `::-webkit-progress-inner-element` - is the out most part of the progress element.
- `::-webkit-progress-bar` - the entire bar of the progress element, the portion under the progress indicator. Is the child of `::-webkit-progress-inner-element`
- `::-webkit-progress-value` - is the progress indicator. Is the child of `::-webkit-progress-bar`

We will use all three values to style our element.

Let's start from the inside and work our way outwards. The first part we will want to style is the progress indicator. For that we will need to use `::-webkit-progress-value`.

We curve the edges and change the color of the bar to be a light blue as seen in Listening 3.14.

Listing 3.14: Styling the progress indicator in Chrome

```
::-webkit-progress-value {
  border-radius: 20px;
  background-color: #7be6e8;
}
```

Figure 3.15 shows our output in Chrome.

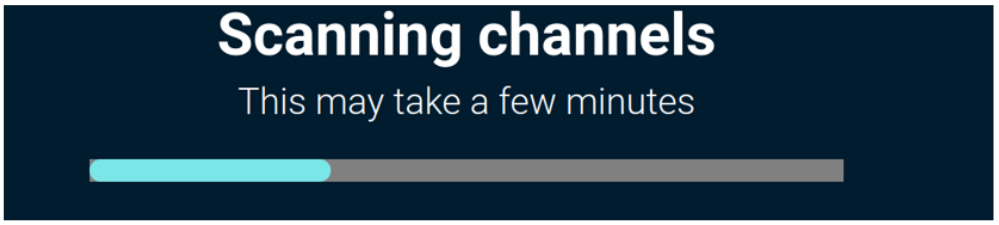

Figure 3.15: Progress Value styled in Chrome

Next we will edit the background behind the progress indicator using `::-webkit-progress-bar`. We will also add rounded corners to the background, and change the color to a linear gradient going from a dark green to a light blue keeping with the theme of the whole piece.

The linear-gradient function takes a direction followed by a series of color and percentage pairs. The direction dictates the angle of the gradient, the color percentage pairs dictate at which points within the gradient we want to shift from one color to another. We will use the keyword value `to right` as our direction. Then, we set a starting color of `#128688` and an ending color value of `#4db3ff`. Our gradient will therefore go from left to right fading from our start color to our end color.

CSS Gradient Generators and Vendor Prefixes

As gradients can be tedious to write by hand, many CSS gradient generators have been created and are freely available on the web. Many still include vendor prefixes in their generated code. These prefixes are no longer necessary as gradients are now supported by all major browsers.

Finally we also add a border radius to the outermost container. The CSS for our progress-bar is shown in Listing 3.15.

Listing 3.15: Styling the progress indicator container in Chrome

```
::-webkit-progress-bar {
  border-radius: 20px;
  background: #0eb98f;
  background: linear-gradient(to right, #128688 0%,#4db3ff 100%);
}
::-webkit-progress-inner-element {
  border-radius: 20px;
}
```

Our progress indicator looks great in Chrome (Figure 3.16), but let's take a look at what it looks like in Firefox.

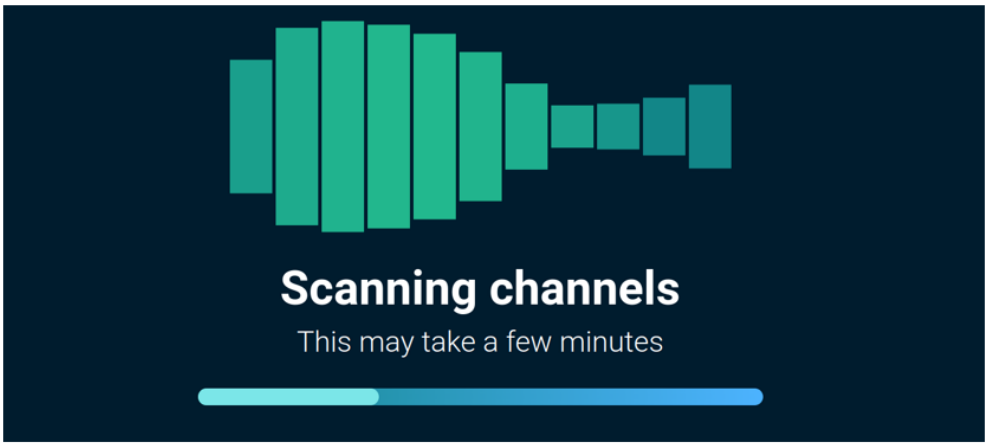

Figure 3.16: Styled progress indicator in Chrome

In Firefox (figure 3.17) we see that our control remains fairly unstyled because instead of `-webkit` vendor prefix, it uses `-moz`.

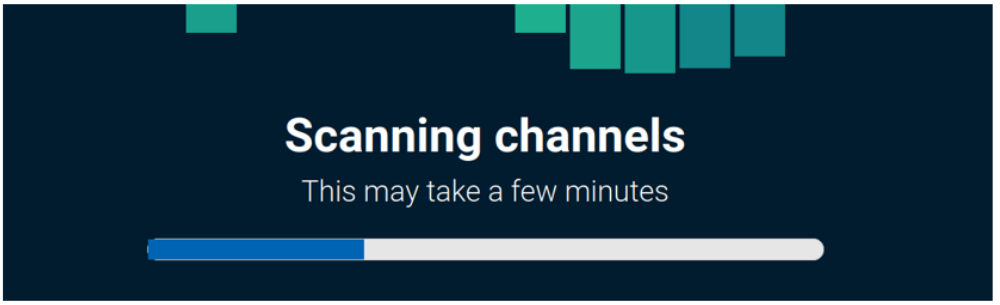

Figure 3.17: Unstyle progress bar in Firefox

Having written the `-webkit` prefixed code, we now need to do the same for browsers that use a `-moz` vendor prefix.

3.7.3 Browser prefixes Styling the progress bar for -moz- vendor prefixed browsers

We will approach the styles a bit differently for Firefox since we don't have as many properties to play with. The only `-moz` prefixed property at our disposal is `::-moz-progress-bar`. Also a pseudo element, it targets the progress indicator itself.

We will therefore style it the same way we styled `::-webkit-progress-value` for Chrome, because we want to achieve the same look in both browsers. Since we are using the same

styles it is logical to add the `-moz` selector to the already existing rule: `::-moz-progress-bar, ::-webkit-progress-value { … }`. It works great in Firefox (Figure 3.18), however it will break Chrome (Figure 3.19).

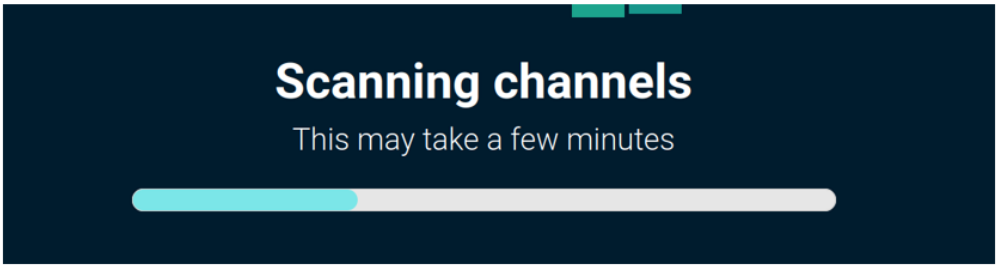

Figure 3.18: Firefox Progress Bar Styled

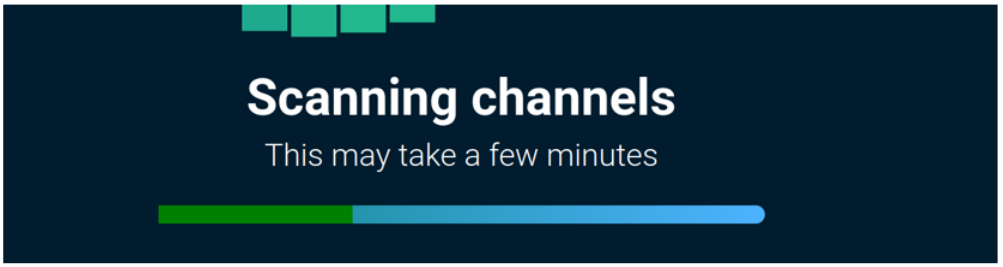

Figure 3.19: Adding both selectors the the same rule breaks Chrome

Having multiple selectors on the same rule should not cause this type of side effect, however, we are dealing with experimental properties which sometimes have non standard behaviors. To prevent this unfortunate side effect, we will write 2 identical rules, one for each selector as seen in Listing 3.16.

Listing 3.16: Styling the progress indicator container in Chrome

```
::-webkit-progress-value {   #A
  border-radius: 20px;       #A
  background-color: #7be6e8; #A
}

::-moz-progress-bar {        #B
  border-radius: 20px;       #B
  background-color: #7be6e8; #B
}
```

#A Rule for Chrome
#B Rule for Firefox

To change the background color for Firefox, we add a background property value to the progress element itself. We will use the same gradient we used in the `::-webkit-progress-bar` rule. Figure 3.20 shows our progress in Firefox.

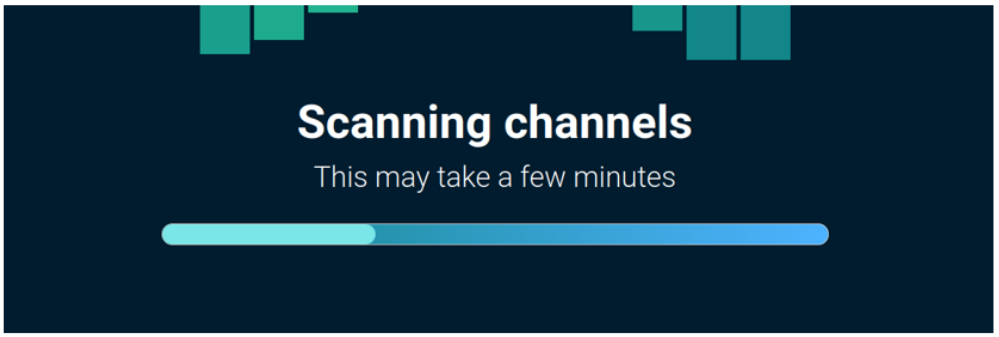

Figure 3.20: Firefox with background applied to the progress element

The last thing we need to do is remove the border, which we will apply to the `progress` rule. To achieve this we set the border property value to `none`. Our final progress rule is shown in Listing 3.17.

Listing 3.17: Finalized progress rule

```
progress {
  height: 24px;
  width: 100%;
  -webkit-appearance: none;
  -moz-appearance: none;
  appearance: none;
  border-radius: 20px;
  background: linear-gradient(to right, #128688 0%,#4db3ff 100%); #A
  border: none;                                                   #B
}
```

#A Gradient background
#B Removing the border

As we can see in Figure 3.21, we have achieved the same look in Chrome and in Firefox.

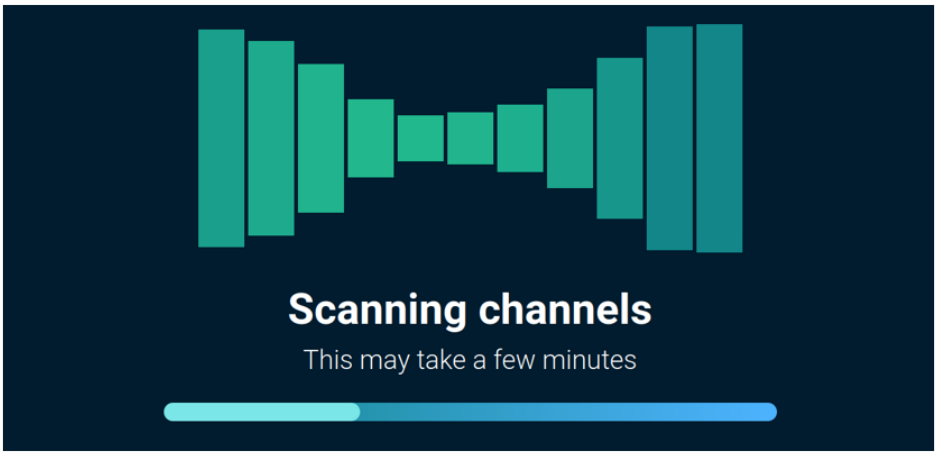

Figure 3.21: Progress bar styles finished in Firefox

We must stress that the styles were achieved using experimental features that are non-standard and therefore should not be used in production. The value here is in being able to experiment with new features before they become readily available. This can also be an opportunity to get involved in the community as it is not uncommon for the working groups developing browser features and specifications to request feedback from the community before new standards are accepted and rolled out for general use.

3.8 Summary

- The animation property is a way to change the position, color or some other visual element using CSS.
- Keyframes are a way to describe what an element should look like throughout its animation.
- We can then delay the start of an animation using the `animation-delay` property
- The `animation-duration` set how long the animation should take to complete
- SVG can be styled using CSS
- The prefers-reduced-motion media queries allows us to conditionally style animations per the user's settings
- The HTML progress bar is a way of showing how far something is uploaded or is being loaded.
- By default the browser applies their own styling to the progress bar but it can be reset using the `appearance` property with a value of `none`.
- Our ability to style progress element is fairly restricted
- There are some experimental properties available to style the progress element but they require the use of vendor prefixes
- Vendor prefixed properties are experimental and we should not use vendor prefixes in our production code

4

Creating a responsive web newspaper layout

This chapter covers

- Using the CSS multi-column layout module to create a newspaper layout
- Using the counter styles CSS at-rule to create custom list styles
- Styling images using the filter property
- Handling broken images
- Formatting captions
- Using the quotes property to add quotation marks to HTML elements
- Using media queries to change our layout based on screen size

In chapter 1 we looked at creating a single column article which taught us the basic principles of CSS. The design, however, was quite simple. Let's revisit the concept of formating articles but make it much more visually interesting. In this chapter we will style our content to look like a page out of a newspaper as seen in Figure 4.1.

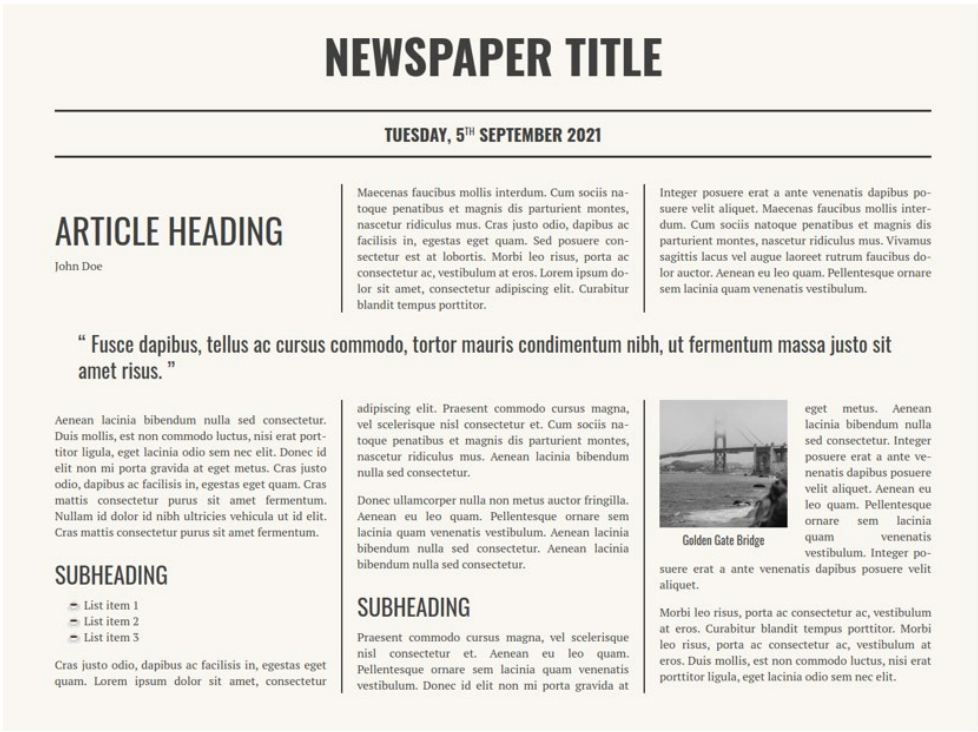

Figure 4.1 The end result we want to achieve

To create the content columns we will use the CSS multi-column module. Along the way we will also look at how we can manage the space between the columns, how to span elements across multiple columns and how to control where the content breaks onto a new column.

Part of the newspaper uses a list of items, which has some default styles provided to us by the user agent stylesheet. We will look at how to use a new module called the CSS Counters module which allows us to customize how our list-items counters (the numbers and bullets) are styled.

Another concept we will cover in this chapter is how to style images. This will include the use of the filter property in conjunction with functions to alter the image's look. We will also look at solutions for broken images and how to make them "fail gracefully". When we say "fail gracefully", or sometimes known as graceful degradation, we are putting in place fallbacks if the thing we are trying to load is having an issue.

The code for our project can be found in the chapter 4 folder of the GitHub repository: https://github.com/michaelgearon/Tiny-CSS-Projects/tree/main/chapter-04

Our starting HTML consists of the following elements (Listing 4.1). Inside the body tag is the title of the newspaper and print date followed by an article. The article has a heading, author name, a quote, 2 subheadings, a list, some paragraphs, and an image.

Listing 4.1 Starting HTML

```
<html lang="en-US">
<body>
  <h1>Newspaper Title</h1>                                    #A
  <time datetime="2021-09-07">                                #B
    Tuesday, 5<sup>th</sup> September 2021                    #B
  </time>                                                     #B
  <article>                                                   #C
    <h2>Article heading</h2>                                  #D
    <div class="author">John Doe</div>                        #E
    <p>Maecenas faucibus mollis interdum. Cum sociis nato...</p>
    <p>Integer posuere erat a ante venenatis dapibus posu...</p>
    <blockquote>                                              #F
      Fusce dapibus, tellus ac cursus commodo, tortor ma...   #F
    </blockquote>                                             #F
    <p>Aenean lacinia bibendum nulla sed consectetur. Dui...</p>
    <h3>Subheading</h3>                                       #G
    <ul>                                                      #H
      <li>List item 1</li>                                    #H
      ...                                                     #H
    </ul>                                                     #H
    <p>Cras justo odio, dapibus ac facilisis in, egestas ...</p>
    <p>Donec ullamcorper nulla non metus auctor fringilla...</p>
    <h3>Subheading</h3>                                       #I
    <img src="./image.jpg" alt="" />                          #J
    <p>Praesent commodo cursus magna, vel scelerisque nisl...</p>
    <p>Morbi leo risus, porta ac consectetur ac, vestibulu...</p>
  </article>                                                  #K
</body>
</html>
```

#A Newspaper title
#B Print date
#C Start of the article
#D Article heading
#E Article author
#F Quote
#G First subheading
#H List
#I Second subheading
#J Image
#K End of the article

Figure 4.2 shows our starting point. The styles applied to the HTML are the defaults provided by the browser. No author styles have been applied to the page yet.

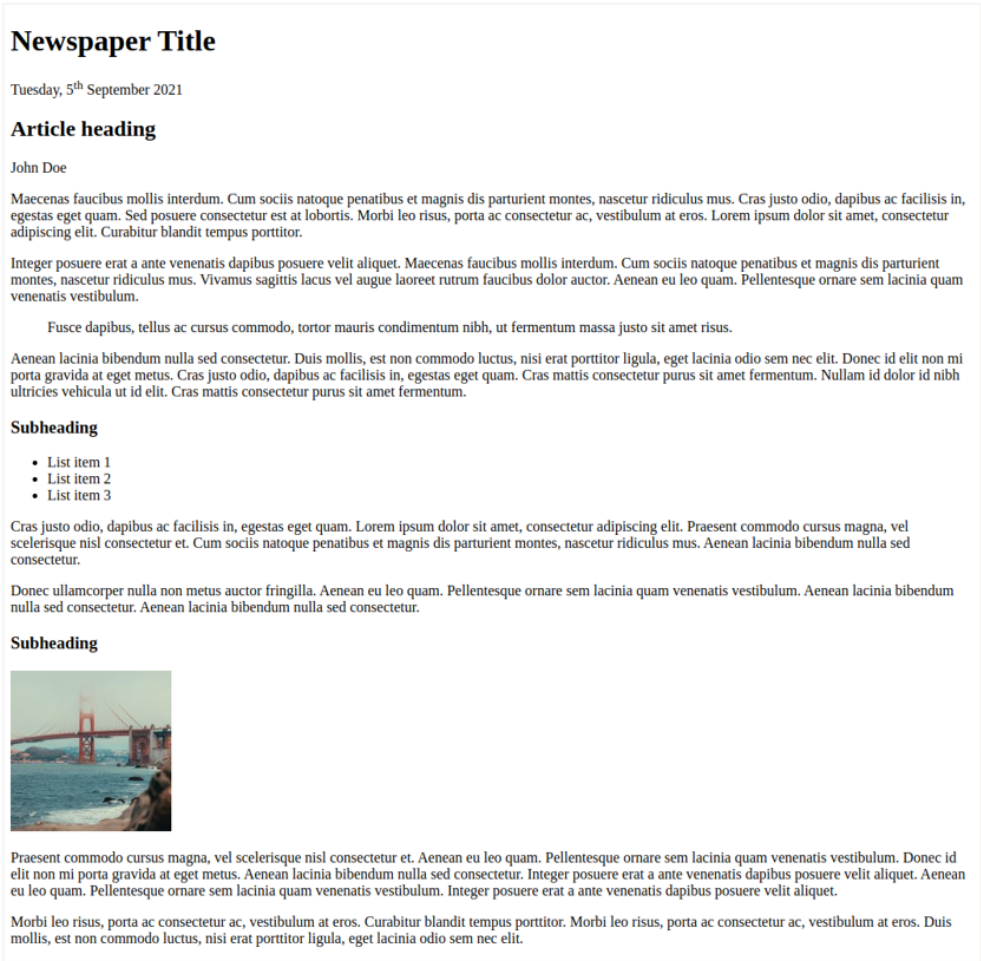

Figure 4.2 Starting point

Before we worry about layout, let's define our theme.

4.1 Setting up our theme

The theme sets the tone for the page and generally consists of colors, fonts, borders, and sometimes padding. Our theme will stay the same regardless of screen size or layout. Oftentimes, the theme of a website is tightly coupled to its logo and brand colors.

We will set some defaults on the `<body>` tag that can be inherited by its descendants. As a general rule, styles that revolve around typography, (color, font-family…) can be inherited by

most elements (exceptions are some form elements, which we cover in Chapter 10). It is worth mentioning that font-size is not inherited. By setting inheritable properties on the parent, the styles will trickle down to the descendents, preventing us from needing to apply them to every element.

4.1.1 Importing fonts from Google Fonts

We apply a background color, font, and text color (Listing 4.2). Notice that before the body rule, we import our chosen font-family from Google Fonts. We tend to use Google Fonts as it's freely available, without needing to create an account or pay to use the fonts. PT Serif is not a font we can safely expect a user to already have loaded on their computer, we therefore have to import it for the browser to know what the glyphs (letters, numbers, and symbols) should look like. We also provide a default of `serif` as a fallback should the import fail.

Web safe fonts

There are very few web safe fonts (fonts we can assume most devices will have access to). The generally accepted list is: Arial, Verdana, Helvetica, Tahoma, Trebuchet MS, Times New Roman, Georgia, Garamond, Courier New, Brush Script MT[1].

Regardless of the font family we choose, it is good practice to always provide a fallback value (Serif, Sans-serif, Monospace, Cursive, or Fantasy)

Although we will do the bulk of the layout later in the chapter, we also add some left and right padding on our body now to move our text away from the edge.

Listing 4.2 Defining some theme styles

```
@import url('https://fonts.googleapis.com/css2?family=PT+Serif&display=swap');  #A

body {
  background-color: #f9f7f1;
  font-family: 'PT Serif', serif; #B
  color: #404040;
  padding: 0 24px;
}
```

#A Importing PT Serif from Google Fonts
#B Applying PT serif to our content and providing a fallback

Figure 4.3 shows our updated page. We notice that all of the elements in the body have inherited the color and font-family.

[1] W3 Schools. "CSS Web Safe Fonts." W3Schools, https://www.w3schools.com/cssref/css_websafe_fonts.asp. Accessed 2 June 2022.

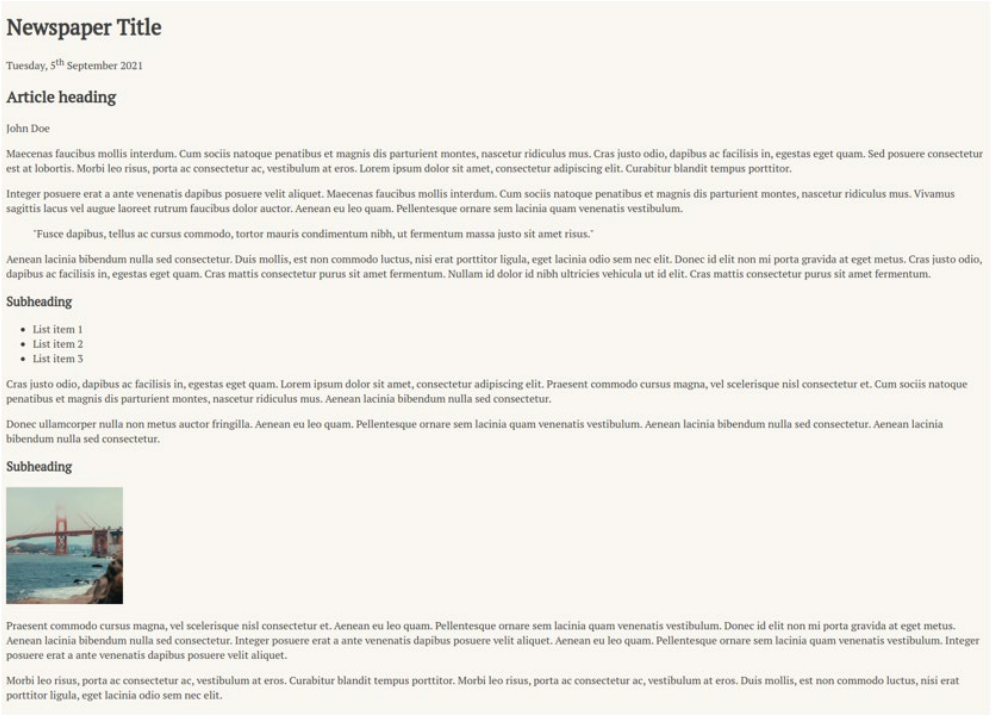

Figure 4.3 Theme styles applied to the body being inherited by descendents

Next we will style the title, headings, and subheadings. Let's start with the newspaper title which is the `<h1>` in the HTML. We want to change the font-family to use a typeface called "Oswald", increase the text size, make it bold, transform the font to use all capital letters, set the line height, and center the text. Like PT Serif, Osward is not a standard font we can expect most users' devices to know about. We will import it much like we did PT Serif.

To import "Osward" from Google Fonts we can either add a second @import at the top of our file or for better performance we can combine the two imports into one import statement. The ability to combine the two imports is specific to Google Fonts. Not all Content Delivery Networks (CDN) will provide this ability.

Notice in Listing 4.3 that in our import after the name of the font we see `:wght@400;700`. This indicates which Oswald font weights we want to import.

Listing 4.3 Styling the newspaper title

```
@import
       url('https://fonts.googleapis.com/css2?family=Oswald:wght@400;700&family=PT+Serif&di
       splay=swap'); #A

h1 {
  font-weight: 700; #B
  font-size: 4rem;
  font-family:'Oswald', sans-serif;
  line-height: 1;
  text-transform: uppercase;
  text-align: center;
}
```

#A updated import that includes both Oswald and PT Serif
#B Equivalent to using a value of bold

Our updated title can be seen in Figure 4.4.

Figure 4.4 Styled title

4.1.2 Font-weight property

The `font-weight` property can either take a number value between 100 and 900 or a keyword value (normal, bold, lighter, or bolder). `normal` is equivalent to `400` and `bold` to `700`. Lighter and bolder change the element's font weight based on the font-weight of the parent element.

Table 4.1 Font weight values and their common weight name

Value	Common weight name
100	Thin (Hairline)
200	Extra Light (Ultra Light)
300	Light
400	Normal (Regular)
500	Medium
600	Semi Bold (Demi Bold)
700	Bold
800	Extra Bold (Ultra Bold)
900	Black (Heavy)
950	Extra Black (Ultra Black)

If we do not import the weight that matches the one we set in the rule, the browser will apply the closest weight it does have access to. Therefore, had we only imported Oswald with a weight of 400 and applied a font-weight value of bold to our element, the browser would have displayed our text with a weight of 400 because that is the only value it has to work with.

4.1.3 Font shorthand property

Using the `font` shorthand property we can combine most of the styles in our rule. The font property requires us to provide a font-family and size, and then optionally the style, variant, weight, stretch, and line-height using the following syntax: `font: font-style font-variant font-weight font-stretch font-size/line-height font-family`. Listing 4.4 shows our updated rule using `font`.

Listing 4.4 Title styles using the `font` shorthand property

```
h1 {
  font: 700 4rem/1 'Oswald', sans-serif;
  text-transform: uppercase;
  text-align: center;
}
```

Let's apply the concepts we just covered regarding importing fonts, font-weight, and the font shorthand property to style the article title and subheadings.

4.1.4 Creating a visual hierarchy

To create visual hierarchy in the page we will set the article heading <h2> to be smaller than our newspaper's title <h1> but larger than the subheadings within the article <h3>. Generally speaking, the larger an element, the more important it will be perceived to be; we therefore use size to make our headers stand out. By using a different font-family than we do on the main bodies of text and making all of our letters uppercase we further their distinction.

Creating a visual hierarchy is important as it allows the user to glance at the screen and immediately recognize elements of interest. It also helps segment information into groups, making the information easier to process and understand. Listing 4.5 shows our header rules.

We will keep the same font family and uppercase lettering and adjust the sizing. We will also remove the browser provided bottom margin of both article headers to keep them closer to the text they precede.

Listing 4.5 Article header rules

```
h2 { #A
  font: 3rem/.95 'Oswald', sans-serif;
  text-transform: uppercase;
  margin-bottom: 16px;
}

h3 { #B
  font: 2rem/.95 'Oswald', sans-serif;
  text-transform: uppercase;
  margin-bottom: 12px;
}
```

#A Article heading
#B Article subheadings

Our articles headers now look as seen in Figure 4.5.

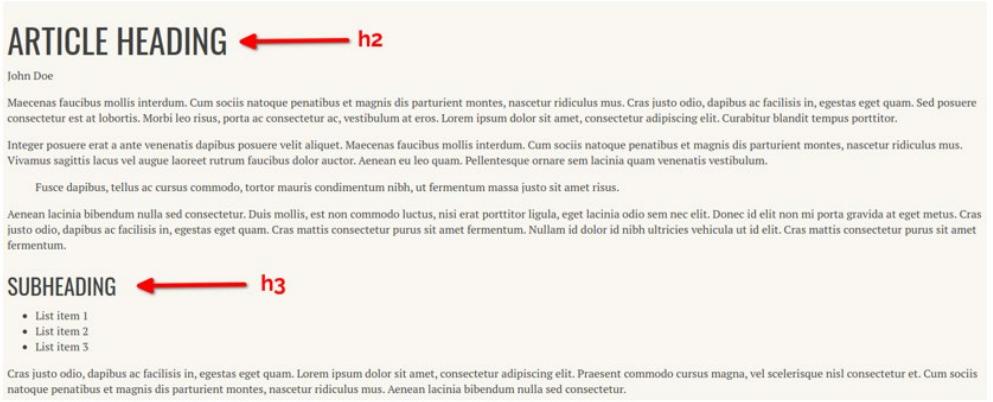

Figure 4.5 Styled article headings

4.1.5 Inline versus block elements

Let's continue to make important elements stand out from the rest of the content, starting with the publication date which is found in our HTML in a `<time>` element. The time element syntactically denotes a specific period in time, and takes an optional `datetime` attribute which translates the date into a machine readable format for search engines. Our time element looks as follows: `<time datetime="2021-09-07">Tuesday, 5<sup>th</sup> September 2021</time>`. Figure 4.6 shows the look we want to achieve.

Figure 4.6 Styled publication date

Starting with the typography, we center the text and use the Oswald font-family, set the `font-size` to `1.5rem`, and make the text uppercase and bold. We then change the text size of the "th" found in the superscript element (`<sup>`) to a slightly smaller font size and normal weight to decrease its prominence.

We then add the top and bottom borders to be 3 pixels thick, solid, very dark gray lines. After adding the borders, we add some top and bottom padding so that we can have some breathing room between the text and the lines.

The (date) time element `<time>` is an inline-level element, meaning that it will only take up the exact amount of space it needs for its content the same way a span or anchor tag does.

In contrast block-level elements (like the div, paragraph, or list) place themselves on a new line and take the full width of their available space. To achieve the design seen in Figure 4.6 we want our `<time>` element to behave as if it was a block-level element so that the text will place itself in the middle of the screen and to make the borders take the full width of the page. To change the element's default behavior we will use the `display` property and give it a value of `block`. Figures 4.7 and 4.8 show the time element before and after adding the `display` property. In Figure 4.6 (before adding the `display` property) the element is exhibiting its default behavior as an inline-level element. In Figure 4.7 (after adding the display property) the element behaves like a block-level element, taking the full width of the screen.

Figure 4.7 The time element exhibiting inline behavior

Figure 4.8 The time element exhibiting block behavior

Styling the publication date in this manner will serve 2 purposes, making it stand out, and creating a visual divide between the newspaper information (the date and newspaper title) and the article itself (everything below the date). The rules we wrote to achieve our design are found below in Listing 4.6.

Listing 4.6 Styling the publication date

```
time {
  font: 700 1.5rem 'Oswald', sans-serif;  #A
  text-align: center;                     #A
  text-transform: uppercase;              #A

  border-top: 3px solid #333333;          #B
  border-bottom: 3px solid #333333;       #B
  padding: 12px 0;                        #B

  display: block;                         #C
}
time sup {                                #D
  font-size: .875rem;
  font-weight: normal;
}
```

#A Typography
#B Handles the borders and padding
#C Makes the element behave like a block-level element
#D Styles the "th"

4.1.6 Quotes

The last bit of text we want to feature is the blockquote found after the second paragraph in the article. Sticking with our theme and like all of the other elements we want to make standout, we will make the font size bigger and bolder. We will also adjust the line height. Finally we add margin to the element. Isolating an element away from other content around it makes it easier to spot. By adding top and bottom margin, we add space between it and the paragraphs above and below creating white space around the element. By adding left and right margins, we change its alignment, effectively indenting it. The added white space creates the isolation we were just mentioning.

Let's also add quotation marks to our blockquote. To add the quotation marks at the beginning and end of our quote we could simply go into the HTML and add them, or we can do it programmatically via CSS.

The `quotes` property allows us to define custom quotation marks. We can pass it the symbols we want to use as our double and single quote glyphs. Not all languages use the same symboles. In American English we use "... " and '... ', in French however we use « … » and ‹ … ›. Using the quote property we can customize which symbols we want to use. If we don't provide a value for `quotes`, the browser's default behavior is to use what is customary for the language set on the document. This property however, only defines the symbols, it does not add them.

To add them we use the `content` property values `open-quote` and `close-quote` in conjunction with the `::before` and `::after` pseudo elements as seen in Listing 4.7. The pseudo elements allow us to insert content via the `content` property before and after the element they are applied to respectively. The `open-quote` and `close-quote` keywords represent opening and closing quotation marks as they are defined by the `quotes` property.

Since we did not add a `quotes` declaration to our `blockquote` rule, the browser will use what is conventional for the document's language which we have set to `en-US` in the language (`lang`) attribute on the `html` tag. The value of `en-US` specifies that our document is written in American English, therefore the symbols that the browser renders are " and " as we can see in Figure 4.9.

Listing 4.7 Styling the bloquote

```
blockquote {
  font: 1.8rem/1.25 'Oswald', sans-serif;
  margin: 1.5rem 2rem;
}
blockquote::before { content: open-quote; }
blockquote::after { content: close-quote; }
```

With our quote now styled, let's turn our attention to the bulleted list found in the middle of the article.

Figure 4.9 Styled title, heading, subheadings, and blockquote

4.2 CSS counters

Our article contains an unordered (bulleted) list. Currently, each list item has the default bullet before each list item. We can alter what our bullet looks like by using the `list-style-type` property. By default, we can choose from disc (•), circle (○), square (▪), and numbers or letters in a number of different languages, alphabets, and number formats. But let's say we wanted our bullet to be an emoji, specifically the hot beverage emoji (☕). We would have to create a custom list style.

To create our custom list type, we will use the `counter-style` at-rule. We have used at-rules before in chapter 3 when we created keyframes. In this case, instead of defining how an animation will behave, we will define how our list looks and behaves. The at-rule is called counter style because it specifically addresses the built counting mechanism for list items in CSS. Under the covers regardless of the list being ordered or unordered, the browser keeps track of the position of the item in the list. It counts the items.

Like with keyframes (which we named in order to reference them inside of our animation property) we will name our counter-style so that we can reference it using the `list-style` property and apply it to our list. Let's name our list-style "emoji". Our at-rule will therefore look as follows: `@counter-style emoji { }`. Next, we will define the behavior our list-style needs to have inside of our at-rule. We will use three properties: `symbols`, `system`, and `suffix`.

4.2.1 Symbols

The symbols property defines what will be used to create the bullet style. To define our emoji as the symbol we want to use we can either use the emoji directly or use its unicode value. Unicode is a character encoding standard that specifies how a 16 bit binary value is represented as a string. In other words, it's the code representation of our emoji. The actual emoji image is determined by the operating system and browser which is why we see variations in how emojis look between iOS and Android for example. The unicode value tells the machine what to render. We use look-up tables such as the one found at https://unicode.org/emoji/charts/full-emoji-list.html to find this value for our emoji. 🍮 is listed as having the following code: `U+2615`. To tell our CSS that we are using a unicode value we will replace the `U+` with a backslash (`\`). Using the unicode value our declaration will therefore look as follows: `symbols: "\2615"`. If we just use the emoji, our declaration will be `symbols: 🍮;`

Next we need to define our system.

4.2.2 System

We are styling a list. Regardless of type (ordered or unordered), under the covers, the browser keeps track of the list item it is styling based on its position inside the list. The first item's integer value is 1, the second 2 and so on. What the `system` property value does is define the algorithm used to convert that integer value into the visual representation we see on the screen.

We are going to use the `cyclic` value. Earlier we only provided one emoji in our `symbols` declaration, but we could have included multiple different emojis using a space delimited list. A `cyclic` value tells the browser to loop through these values and when it runs out, to start back at the beginning. Since we only have one value, it will apply the 🍮 to the first list item and run out of symbols. Having run out getting to the second list item, it will start back at the beginning of the list applying the 🍮 once again but to the second list item this time. Then it will run again moving on to the third list item and the cycle continues.

Finally we will set a suffix.

4.2.3 Suffix

The `suffix` property defines what comes between the bullet (our emoji) and the contents of the list item. By default it's a period. We want to replace the period with plain white space

between our emoji and list item content. We will therefore set our suffix property value to " " (a blank space).

4.2.4 Putting it all together

With our counter-style defined, we can now apply it to our list. Remember that we named the counter-style rule "emoji"; we will apply the name as the `list-style` property value for our list as seen in listing 4.8.

Listing 4.8 Styling the bloquote

```
@counter-style emoji {  #A
  symbols: "\2615";     #B
  system: cyclic;
  suffix: " ";
}

article ul {
  list-style: emoji;     #C
}
```

#A at-rule defining the custom list-style's behavior
#B ☕
#C Applying the custom list-style to the article's lists

Figure 4.10 shows our newly styled list.

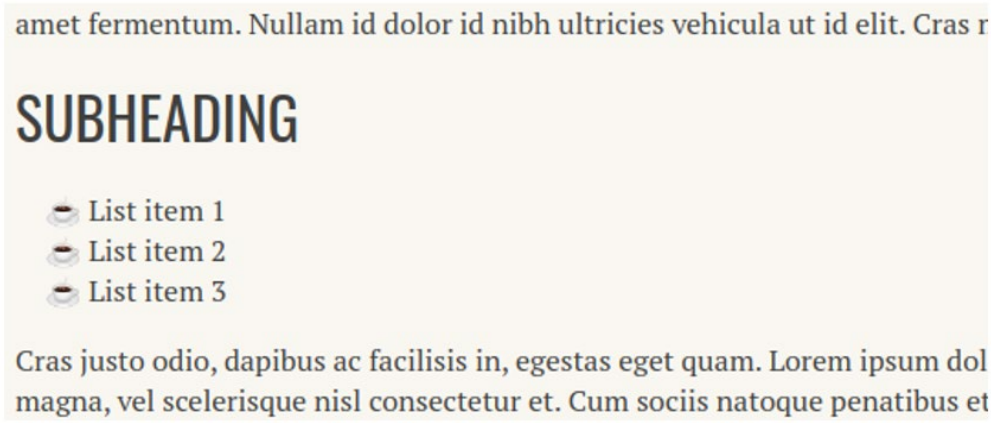

Figure 4.10 List styled using ☕ as counters

Let's continue going down the page and style the image next.

4.3 Styling images

Historically, newspapers were printed in black and white, colored ink in newsprint is a fairly new thing when we consider the history of print. To give our design a bit of a retro vibe, we will therefore make our image grayscale. We will first look at how to alter our image using filters. Unlike print, on the web we need to worry about resources not loading or links being broken, we will therefore also look at how to make the image fail gracefully should it fail to load. Finally we will add a caption to accompany the image.

4.3.1 The filter property

Like photo editors or on social media websites like Instagram we can apply filters to images with CSS. We can alter colors, blur, and add drop shadows. Figure 4.11 shows examples of some of the things we can do to our images using filters in CSS. Checkout the code sample in codepen to see it in action: https://codepen.io/michaelgearon/pen/porovxJ.

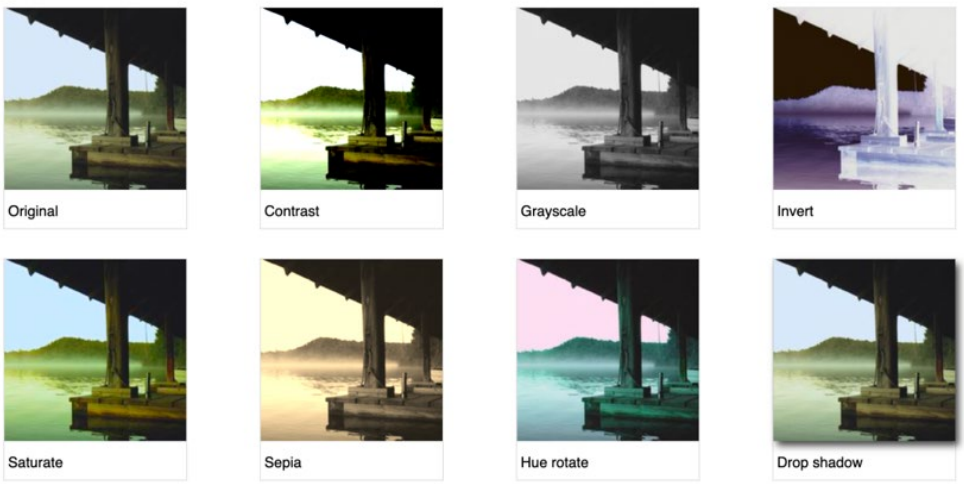

Figure 4.11 Examples of images altered using the filter property

If we think about photography (before the digital era), when we used film and had to go to a shop and have it developed, we applied filters by adding a translucent disk over our lens, which altered the light coming into the camera box and onto the film. By altering the nature of the light, we altered the image being produced. For example, if we use a red filter while taking a picture, only the red colored wavelength is allowed through, therefore our picture would be tinted red. Polarized sunglasses are another example of a filter which alters the light coming through a lens.

We can still use physical filters with digital cameras. However, in many cases, filters are applied digitally after the picture has already been taken.

In CSS to apply a filter to the image, we use the `filter` property and then a function which defines the behavior we want the filter to have. A list of the available functions can be found at https://developer.mozilla.org/en-US/docs/Web/CSS/filter#functions. We will use the `grayscale()` function to make our picture appear to be a black and white photo.

The grayscale() function takes a percentage which represents how much we want to reduce the amount of color in the image. We want to remove all of the color, so we will pass in a value of 100%. Our rule therefore looks as follows: `img { filter: grayscale(100%) }`. Figure 4.12 shows the filter applied to our image.

Figure 4.12 Grayscale Image

One consideration to make before using filters is the impact on website performance. Some of the filter functions like grayscale are relatively simple for the browser to process, but functions like drop-shadow and blur can be resource-intensive. If we find that we are applying many filters to a large number of images, it is worth considering the impact of the filters on the overall page performance and whether we should be preprocessing the image ahead of time rather than applying the change via CSS.

4.3.2 Handling broken images

Even with the most thorough of diligence and best testing practices, broken image links can happen. Let's add some fallbacks in order to ensure that should our image fail to load (regardless of the reason) we maintain a positive experience for our users.

First let's purposely break our link. In the HTML we will replace the path to the image to an image file that does not exist in our project like so: `<img src="./my-broken-image.jpg" alt="my broken link" />`. The image will not display as broken as seen in figure 4.13.

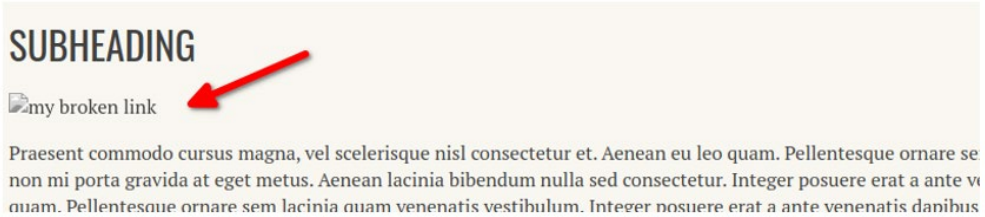

Figure 4.13 Broken link with alt text

When the link is broken, we will hide the image. There won't be anything there, but it will be less unsightly than the broken image icon. Because there is no way to detect that an image is broken in CSS, we will need to use a little bit of JavaScript to help us know when to hide the image. We will use the `onerror` JavaScript event handler to trigger a change in styles as follows: `<img src="..." alt="..." onerror="this.style.display='none'" />`. The bit of code that is of interest to us here is the `onerror` attribute. When an error occurs, the JavaScript inside of the `onerror` attribute triggers and set's the image's `display` property to `none`, hiding the image. We can see that in Figure 4.14, our broken image is now missing.

Figure 4.14 Broken image is missing

The `onerror` code will only trigger when the image fails to load, so let's fix our resource path to our image but keep the error handling: `<img src="./image.jpg" alt="" onerror="this.style.display='none'" />`. Our image is now restored (Figure 4.15), but has a safeguard in case it should fail.

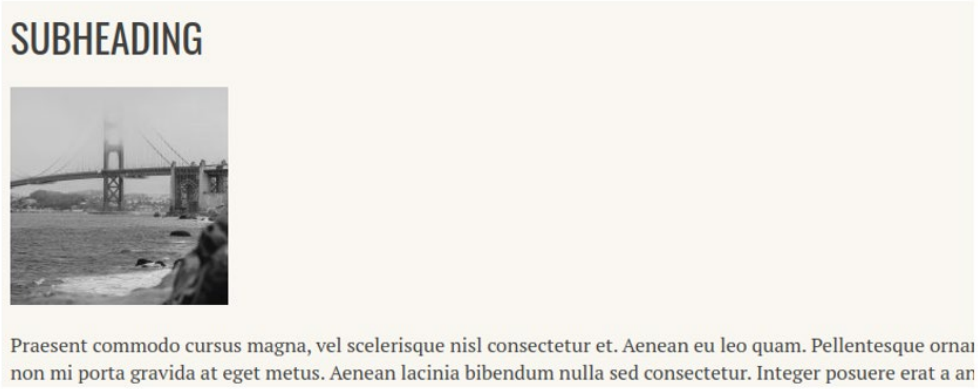

Figure 4.15 Restored image with fallback

Next let's add a caption to your image.

4.3.3 Formatting captions

The Image does not currently have a caption, so we are going to add one using the `figure` and `figcaption` HTML elements. Then we will style it.

`<figure>` and `<figcaption>` go hand in hand. The figure contains the image and then the optional `figcaption`. Often seen in books and other publishing material, diagrams, charts and images will have text underneath the element describing it or relating it to the text. Semantically, the benefit of grouping the image and the caption together is that it programmatically links the image with its caption. From a styling perspective, having the elements together in a parent element allows us to position the element and its caption together as a unit.

Listing 4.9 shows how to change the HTML to add the figure and caption.

Listing 4.9 Adding a figure and figcaption to the HTML

```
<figure>  #A
  <img src="./image.jpg" alt="" onerror="this.style.display='none'" />  #B
  <figcaption>Golden Gate Bridge</figcaption> #C
</figure> #D
```

#A Start of the figure
#B Our image
#C Our image caption
#D End of the figure

Let's style the figure and the caption, starting by removing the browser provided margins (seen in Figure 4.16) that are currently being applied to the figure.

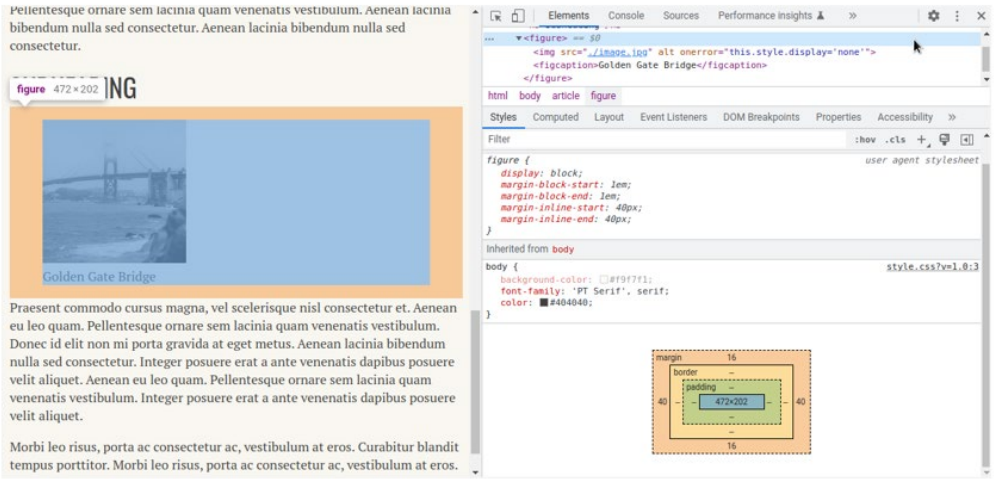

Figure 4.16 Figure browser provided styles

We will then reinstate a bottom margin so that our caption is kept separate from the paragraph below it. Finally we will center the image and caption.

We will also style the caption's text to use the Oswald font-family (the one we used for all the headers) so that it will be visually differentiated from the article text.

Listing 4.10 shows the CSS used to style the figure and caption.

Listing 4.10 Figure and figcaption styles

```
figure {
  margin: 0 0 12px 0; #A
  text-align: center;
}
figcaption {
  font-family: 'Oswald', sans-serif;
}
```

#A Padding shorthand property: top, left, and right padding set to 0 and bottom set to 12px.

At this point we look good on narrow screens but we still need to create our columns on wide screens. Figure 4.17 shows the progress we have made on our project thus far.

Figure 4.17 Progress thus far including styled figure and image caption

Next we will look at how to create a multi-column layout using the multi-column layout module.

4.4 Using the CSS multi-column layout module

The CSS multi-column module is fairly new and compared to similar modules like the grid and flexbox modules the multi-column layout module is perhaps a lesser-known option as a way to present content but no less useful.

The purpose of this module is to allow content to naturally flow between multiple columns. It works very similarly to how we create multiple column layouts in a Word or Google doc. We assign columns to a section of content and the content naturally flows from one column to another.

Since we only want our content placed in columns on wider screens, we will use a media query to conditionally apply our columns only after the window reaches a particular size.

4.4.1 Media queries

Media queries are a type of at-rule. Like `@counter-style` which we used earlier in this chapter, it will start with an @ symbol followed by the identifier `media`. Then we set the instruction for when the rules found inside of the media query should apply. We will want to place the content in columns once our window width is greater than or equal to 955px. Our media query will therefore be: `@media(min-width: 955px) {}`. The diagram found in Figure 4.18 breaks down the individual pieces of the query.

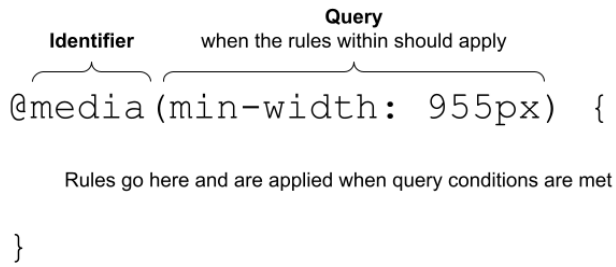

Figure 4.18 Media query breakdown

Inside of the media query we will now define our columns.

4.4.2 Defining and styling columns

There are 2 ways we can define how the columns are created:

- We can dictate a column width, and the browser will create as many columns of that width it can in the available space.
- We can dictate how many columns we want, and the browser will fit that number of equally sized columns in the available space.

We will go with the second option since we already know we want to create 3 columns. We therefore specifically target the article and using the `column-count` property set our quantity to 3 as shown in Listing 4.11.

Listing 4.11 Conditionally break article into 3 columns based on screen width

```
@media(min-width: 955px) { #A
  article {
    column-count: 3;       #B
  }
}
```

#A Media query
#B Setting how many columns we want

Figure 4.19 shows our article layed out into three columns using the above CSS.

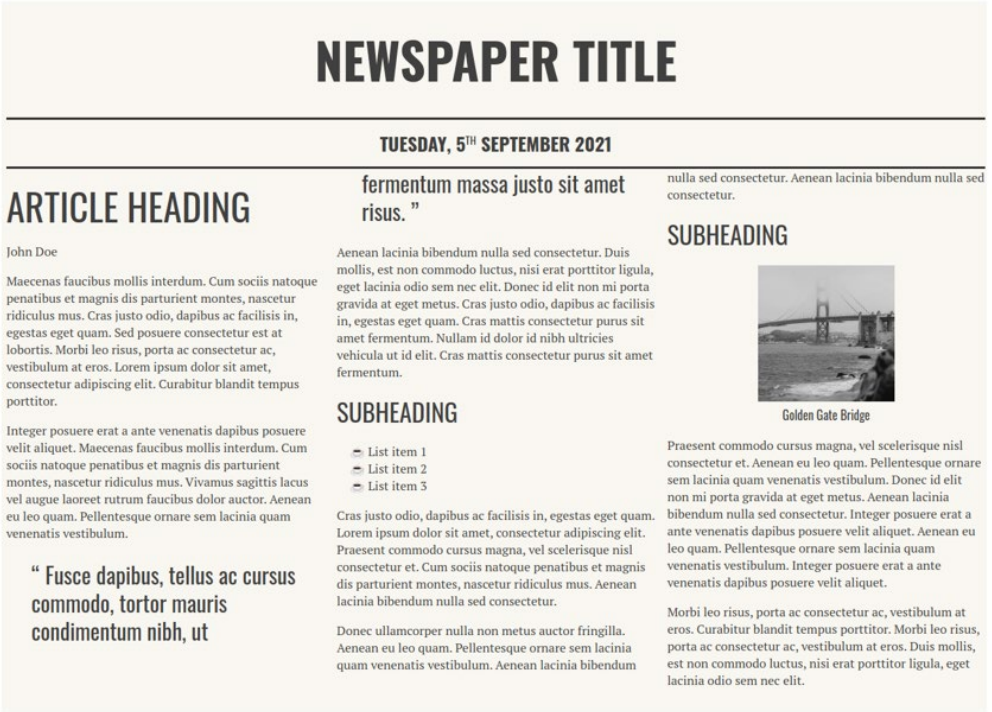

Figure 4.19 3 column layout

Next we will adjust the spacing between our columns and add vertical lines between them. Let's start with the vertical lines.

4.4.3 Using the column-rule property

To create a clear separation between our columns we will add a vertical line in between each column using the `column-rule` property. Like borders and outlines, we will need to set a line type, width, and color. To keep our line work consistent we will use the same color and style of line as the ones we set for the borders above and below the date at the top of the page. We will make the lines slightly narrower, however.

The lines at the top of the screen separate content types (title, date, article), here we are within the same content type. We add the lines to make visual separation of the columns easier; we don't want to break up the content. We therefore want the lines less prominent, so we will make them thinner.

To create the lines we add `column-rule: 2px solid #333333;` to our existing article rule inside of the media query. Our article now looks as seen in Figure 4.20.

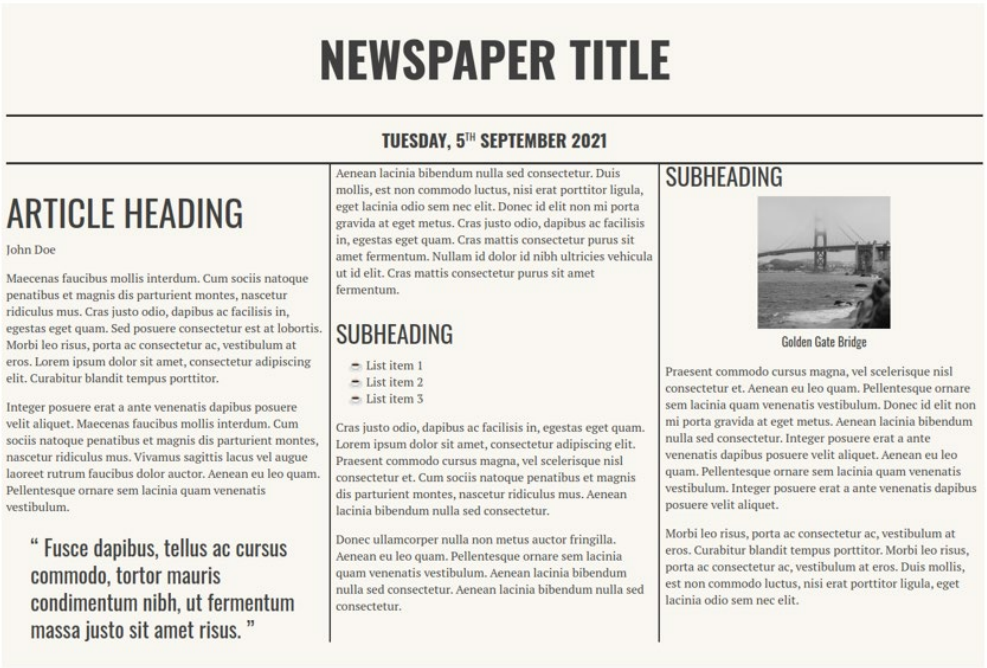

Figure 4.20 Columns with added vertical lines

With our lines in place we can see that we have some crowding between the article itself and the date and that we could use a bit more space between our lines and our text.

4.4.4 Adjusting spacing with the column-gap property

To adjust the spacing between the article and the date, we will add 36px of margin to the top of the article. When working out a value to use for spacing it is sometimes judging it and seeing what looks right on the page, we want to create enough spacing that each item has its own space and is clear but not spacing it too far apart that it looks separated.

> **TIP** In design there are the Gestalt Principles which are a collection of principles of human perception that describe how humans group similar elements. 1 of 7 laws is the law of proximity. This principle talks about how things that are close together appear to be more related than things that are spaced farther apart.

With the space between the article and the date handled, let's turn our attention to the space between the columns. To add a gap between our vertical lines and our text we use `column-`

`gap` property. The `column-gap` property defines the amount of white space we want to have between our columns; we will set ours to 42px;

We continue to add these styles inside of the media query as seen in listing 4.12 because we only want them to apply when our layout is columned. We do not want these style changes to apply to narrower screens.

Listing 4.12 Updated media query and article rule

```
@media (min-width: 955px) {
  article {
    column-count: 3;
    column-rule: 2px solid #333333;
    column-gap: 42px;
    margin-top: 36px;
  }
}
```

With these adjustments made (Figure 4.21), let's turn our attention to the quote.

Figure 4.21 Layout with adjusted spacing

Earlier in this chapter, we styled the blockquote so that it would stand out but now that we have a multi column format, it gets a little lost in the other visual elements on the page. Let's make it span multiple columns to really make it pop.

4.4.5 Making content span multiple columns

We can make elements span across multiple columns by using the `column-span` property. Our choices of values are `all` or `none`. Since we want the quote to go across the entire page, we will choose `all`. Inside of our media query will therefore add the following rule: `blockquote { column-span: all }` resulting in the layout found in Figure 4.22.

Notice that the flow of the content has changed. We added arrows to show the new flow introduced because of making the quote span across the screen. Instead of the entirety of the article flowing from top left to bottom right, evenly distributed across the columns, by adding the `column-span: all` on the quote, content that is before the quote, now flows top left to right across the page above the quote. The content after the quote will do the same below it. As a result of spanning content, we have changed the flow of the text through our columns.

Figure 4.22 Content reflow due to spanning the blockquote across the columns

When we look at the content flow we notice that the caption and the image have been split across 2 different columns which isn't ideal. Let's prevent that from happening.

4.4.6 Controlling content breaks

To prevent the image and its caption from ending up in different columns we can use the `break-inside` property with the keyword value `avoid` which we set on the `<figure>` element. With this declaration we inform the browser that when it's generating the columns the contents of the element should stay together as a unit and should not be split across multiple columns. In other words, the image and figcaption should remain together. The rule we add to the media query is: `figure { break-inside: avoid }`. Figure 4.23 shows the resulting output.

Figure 4.23 Keeping the image and caption together

4.5 Finishing touches

With our content flowing the way we want it across the columns let's polish up some final details. One of the hallmarks of newspaper layouts is that the text is often justified.

4.5.1 Text justification and hyphens

Justification refers to the alignment of the lines inside of a body of text as described in Figure 4.24. The text being justified means that the lines of text will start and end at the same spot forming a box rather than having ragged ends like when the text is left aligned.

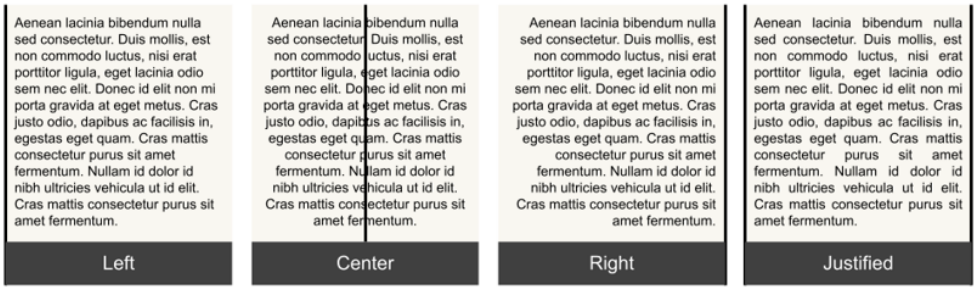

Figure 4.24 Text justification

Let's go ahead and justify our paragraph text. To do this we will use the `text-align` property and give it a value of `justify`. To make the lines equal in length, extra space is distributed across the line. We can tune how the space is redistributed using the `text-justify` property. If we don't set a `text-justify` a value, the browser will choose what it thinks is best for the situation. We have a fluid design, it grows and shrinks with the window size. What is best may be different based on the window size; we will therefore let the browser decide what will work best.

We will, however, add some hyphens. By default, browsers will not hyphenate words when it comes to the end of a line, it will simply continue to the next line. We can alter this behavior by setting the `hyphens` property to `auto`. Allowing the browser to hyphenate words at the end of lines will help diminish the amount of whitespace that is needed between our words to justify the text.

Listing 4.13 shows our paragraph rule. We continue to include our updates inside of our media query, as these changes are only relevant once we switch to the columns layout.

Listing 4.13 Justifying paragraph text

```
@media (min-width: 955px) {
  ...
  p {
    text-align: justify;
    hyphens: auto;
  }
}
```

Our paragraphs now look as seen in Figure 4.25.

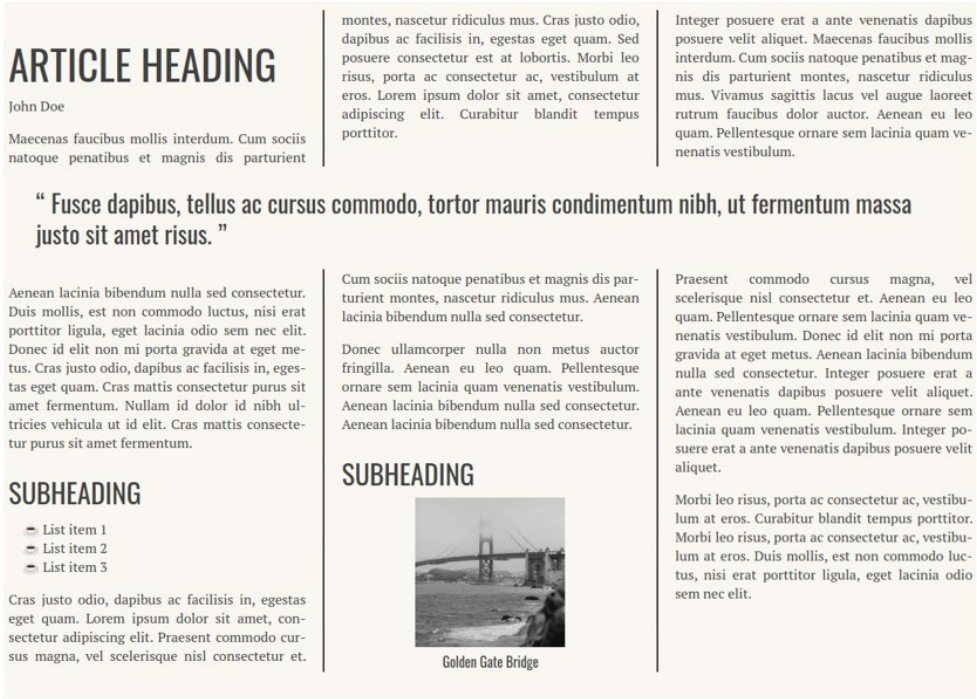

Figure 4.25 Justified and hyphenated paragraph text

As we look at our layout, we notice that the image looks a little odd and out of place at the bottom of the second column. Let’s fix that.

4.5.2 Wrapping the text around the image

To reconnect the image with the subsequent text, we will push the image and its caption to the left and have the text wrap around the image using the `float` property. Applying the `float` property to an element pushes it to either the left or the right, allowing text and inline elements to wrap around it.

This is where having the image and caption as a unit inside of a figure tag comes in really handy for styling. Since they are both contained inside of the figure, we will apply the float to the figure, which will neatly wrap the text around both the image and the caption.

Listing 4.14 shows how we float the figure. Notice that we added a right margin to the figure. Since we are floating the figure to the left, it places itself on the left side of the column, allowing the text to wrap around it in the left over space to the right as seen in Figure 4.26. The right margin creates a space between the image and the text so that the text does not come right up against the edge of the image.

Listing 4.14 Floating the figure

```
@media (min-width: 955px) {
  ...
  figure {
    float: left;
    margin-right: 24px;
  }
}
```

Figure 4.26 Floated image

There are a lot more really cool things we can do with floating images covered in chapter 7. For now though, let's focus on our newspaper page. The last thing we will address is handling how the page behaves if we are looking at the content in a really wide window.

4.5.3 Max-width and margin auto

Figure 4.26 shows us that our layout starts to degrade as the window gets really wide. The wider the window the worse it will get. More and more users have extra wide screens and we therefore need to consider what would happen if they have the window maximized taking up the entire screen. To handle this use case we will use the same trick we used with the loader in chapter 2. We will set a maximum width for our layout and then set its left and right margins to `auto`, which will center the container when the size of the window is greater than our maximum width.

For our page our container is the body, so we will give our body a `max-width` of 1200 pixels and set our left and right margins to auto. We will also need to move the background color from being set on the body to being set on the html element otherwise, when our screen size is greater than 1200 pixels we will end up with a white band to the left and right of our page.

These changes won't go inside of the media query. We will edit the styles we set on the body at the beginning of this chapter and add an html rule to set the background color. Our changes are as follows (Listing 4.15):

Listing 4.15 Floating the figure

```
html { background-color: #f9f7f1 }

body {
  background-color: #f9f7f1; #A
  font-family: 'PT Serif', serif;
  color: #404040;
  padding: 0 24px;
  max-width: 1200px;  #B
  margin: 0 auto;     #C
}
```

#A We move the background color from the body rule to the html rule
#B Sets the maximum width our page can become
#C Centers the page when the window is greater than 1200px

With these final changes we have a page that looks works for both mobile and desktop users. Figure 4.27 shows our finished layout.

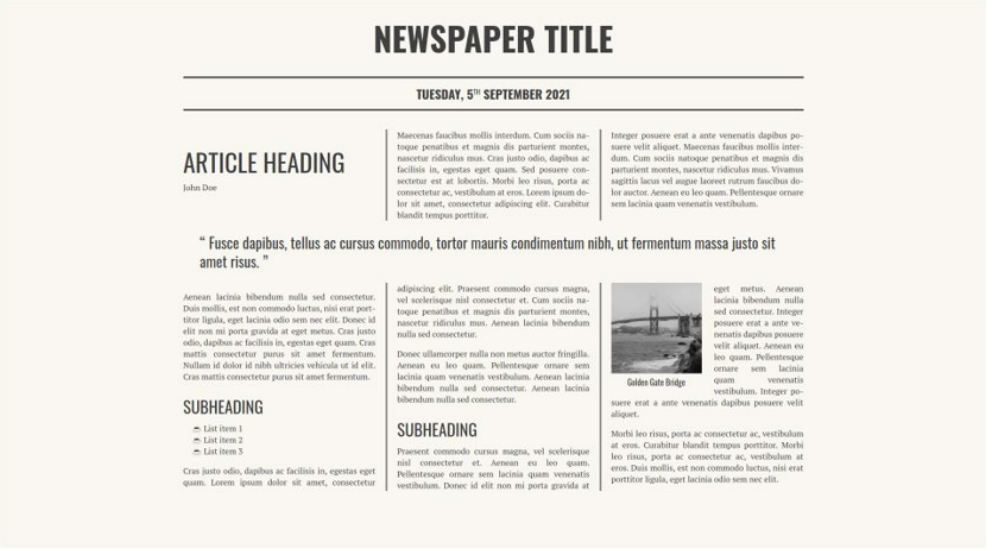

Figure 4.28 Finished layout

4.6 Summary

- A theme is the general look and feel we will maintain throughout an application
- We may need to import our fonts if we use something that is not considered websafe
- Creating a visual hierarchy will help our users orient themselves to the page and identify important information
- We can control which symbols the browser uses when instructed to display quotation marks
- We can customize the way our lists display their bullets using the counter-style at-rule
- Filters allow us to alter the appearance of an image
- We can create multi column layouts using the multi-column layout module
- We can make content span all of the columns when creating multi-column layouts
- We make the browser use hyphens to break words at the end of lines
- Float allows us to wrap text around an element

6

Creating a profile card

This chapter covers

- CSS custom properties
- Creating a background using radial-gradient
- Setting image size
- Positioning elements using flexbox

In this chapter we will create a profile card. In web designs, cards are a visual element that contains information on a single topic. We are going to apply this concept to someone's profile information, essentially creating a digital business card. This type of layout can often be seen on social media or blog sites to give readers a quick overview of who authored the content. It will sometimes have links to a detailed profile page or opportunities to interact with the person the profile belongs to.

By the end of the chapter, our profile card will look as follows (Figure 6.1):

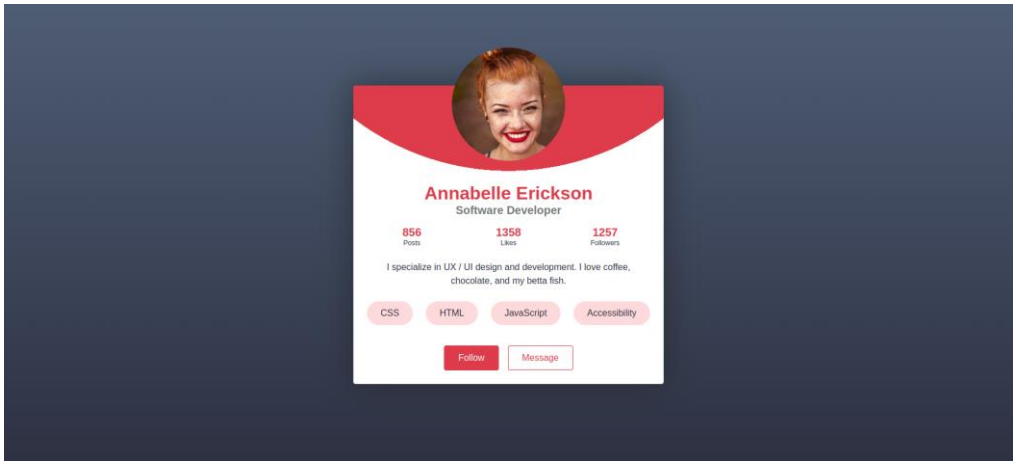

Figure 6.1 Final Output

To create this layout, we will do a lot of work revolving around positioning, specifically leveraging flexbox to help us align and center elements. We will also look at how to make a rectangular image fit into a circle without distorting the image.

Let's dive right in and take a look at our starting HTML (Listing 6.1) which can be found in the gitbub repo at https://github.com/michaelgearon/Tiny-CSS-Projects/tree/main/chapter-06.

We have a `div` with a class of `card` which contains all of the elements being presented in the profile card. To set our blog post information we use a description list.

Description list

A description list contains groups of terms including a description term (dt) and any number of descriptions (dd). Description lists are often used for glossaries or to display metadata. Since we are pairing terms (posts, likes, and followers) with their counts (the number), this a great use case for using a description list.

Our technologies (CSS, HTML…) are presented in a list.

Listing 6.1 Project HTML

```
<body>
  <div class="card">  #A
    <img class="portrait" src="./img/portrait.jpg" alt=""> #B
    <h1>Annabelle Erickson</h1> #C
    <div class="title">Software Developer</div> #D
    <dl>                              #E
      <div>                           #E
        <dt>Posts</dt>                #E
        <dd>856</dd>                  #E
      </div>                          #E
      <div>                           #E
        <dt>Likes</dt>                #E
        <dd>1358</dd>                 #E
      </div>                          #E
      <div>                           #E
        <dt>Followers</dt>            #E
        <dd>1257</dd>                 #E
      </div>                          #E
    </dl>                             #E
    <p class="summary">               #F
      I specialize in UX / UI design and development. I love coffee, chocolate, and my
       betta fish. #F
    </p>    #F
    <ul class="technologies">         #G
      <li>CSS</li>                    #G
      <li>HTML</li>                   #G
      <li>JavaScript</li>             #G
      <li>Accessibility</li>          #G
    </ul>                             #G
    <div class="actions">             #H
      <button type="button" class="follow">Follow</button>   #H
      <a href="" class="message">Message</a>                  #H
    </div>                                                    #H
  </div> #I
</body>
```

#A Start of the card
#B Profile Image
#C Profile holder's name
#D Profile holder's job title
#E Post information
#F Personal summary / about
#G Technologies
#H Actions

As we begin styling our card, our page looks as follows (Figure 6.2).

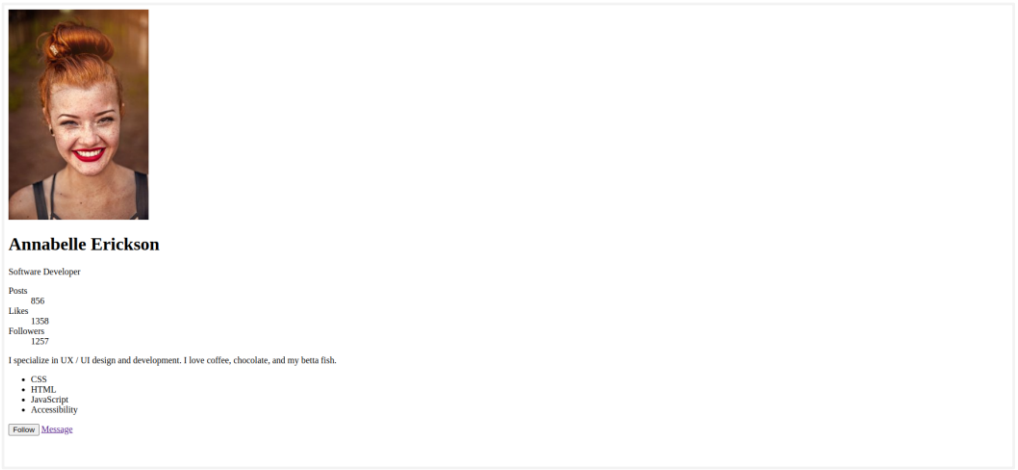

Figure 6.2 Starting point

6.1 CSS custom properties

In our layout, specifically when we style the profile image and colored portion at the top of the card under the image, we are going to need to use the image size value for a number of calculations. In other languages like JavaScript, when we have a value we are going to be referencing multiple times we use variables. In CSS we have custom properties also sometimes referred to as CSS variables.

To create a custom property we prefix the variable name with 2 dashes (`--`) immediately followed by the variable name. We assign the value in the same way we do any other property with a colon (`:`) followed by the value. A CSS variable declaration would therefore looks as follows: `--myVariableName: myValue;`

Like any other declaration, we need to define our variables inside of a rule. For our project we are going to define our colors and image size and declare them inside of a `body` rule as seen in listing 6.2. Since we define our variables on the `body`, the `body` element and any its descendants will have access to the variables.

Notice that we can assign different types of values to our variables. We assign colors, such as in our `--primary` variable (probably one of the most common uses for CSS custom properties), but we also define a size (`--imageSize`), a font family, and a gradient (`--page-background`).

To reference the variable and use it as part of a declaration, we use the following syntax: `var(--variableName)`, therefore to assign our text color for example would declare: `color: var(--text-color);`

Listing 6.2 Defining CSS custom properties

```
body {
  --primary: #de3c4b; #A
  --primary-contrast: white;
  --secondary: #717777; #B
  --font: Helvetica, Arial, sans-serif;
  --text-color: #2D3142; #C
  --card-background: white;
  --page-background: linear-gradient(#4F5D75, #2D3142);
  --imageSize: 200px;

  background: var(--page-background);
  font-family: var(--font);
  color: var(--text-color);
}
```

#A dark salmon
#B gray
#C very dark blue-gray

> **NOTE: LINEAR GRADIENT** Our linear gradient will go from top to bottom fading from a dark blue to very dark blue. For a more in depth explanation of linear gradients checkout Chapter 3 Section 3.7.2.

With our background and font color and family applied (Figure 6.3) we notice that our background repeats at the bottom of the page.

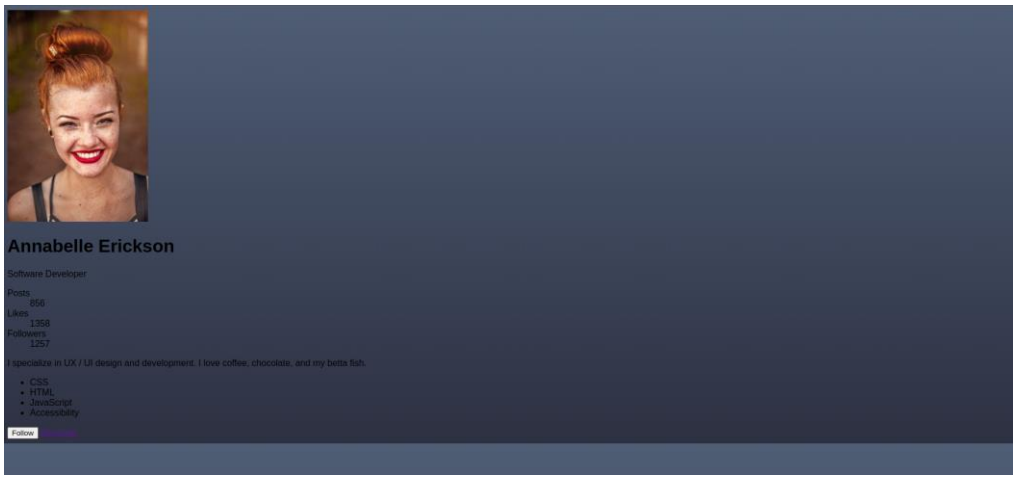

Figure 6.3 Adding the background to the body

6.2 Full height backgrounds

A linear gradient is a type of image. When we apply an image as a background to an element in CSS, if the image is smaller than the element, the image will repeat or tile. In this particular case we don't want the image to repeat.

We have 2 ways we can fix this:

- We can tell the background we don't want it to repeat using `background-repeat: no-repeat`; However since our body element is only as tall as its contents, if the window is taller than the content we will be left with an unsightly white bar at the bottom of the page which is not ideal.
- Our second option (the one we will use) is to make both the html and body take the full height of the screen rather than have it size itself to contents within.

We therefore add the rule found in Listing 6.3 to our stylesheet.

We reset to margin and padding to 0 because we want to ensure we go edge to edge inside of the window.

To set the height, we actually use `min-height` because should the content length be greater than the height of the window, we still want the user to have access to the content and for the background to be behind it. By using `min-height` we instruct the browser to make the `body` and `html` elements at a minimum the height of the window. If the contents would force the elements to be taller, then the browser will use the height of the content.

The value we set for `min-height` is `100vh`. Viewport height (`vh`) is a unit that is based on the height of the viewport itself. It is percentage based so assigning a value of `100vh` to `min-height` means we want the element to have, at minimum, a height equal to 100% of the viewport height.

Listing 6.3 Making the background full height

```
html, body {
  margin: 0;
  padding: 0;
  min-height: 100vh;
}
```

Now that we have our background set (Figure 6.4), let's style the card.

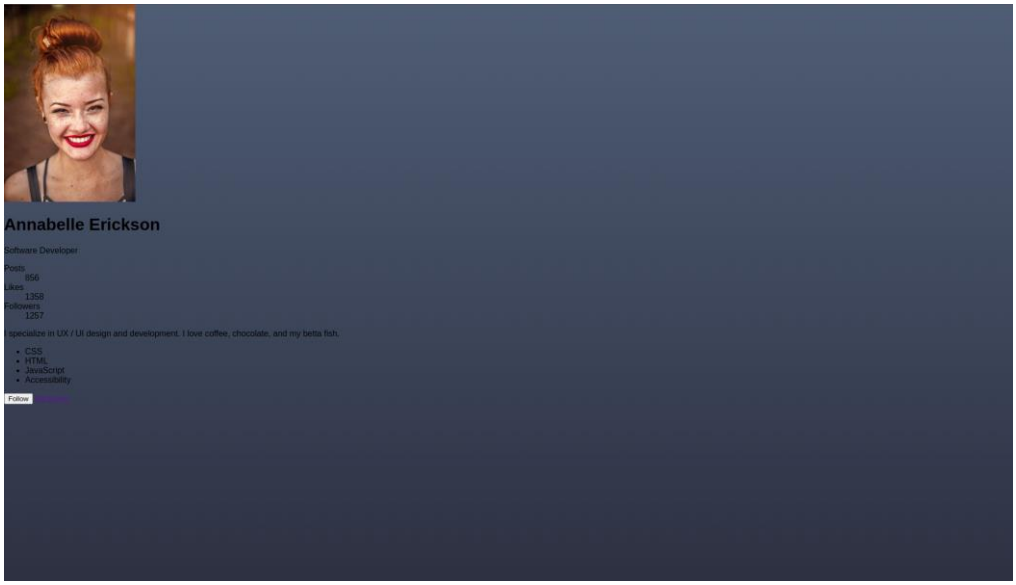

Figure 6.4 Full screen gradient background

6.3 Styling and centering the card using flexbox

Let's start with styling the card itself. We will give it a white background and shadow to give our layout some depth. Notice that instead of using the color value for the background, we use our background variable.

We are also going to set the width of the card to `75vw`. Viewport Width (`vw`) is the horizontal counterpart to the viewport height (`vh`) unit we used earlier. It is also percentage based, so by setting our width to `75vw` we are setting the width of the card to be 75% of the total width of the browser window.

We are then going to further constrain the width of the card to a maximum of 500 pixels wide. By using both the `width` and `max-width` properties, we allow for the card to shrink when the screen size is narrow but constrain the card from becoming too wide and unruly on larger screens.

Lastly we curve the corners of the card using border radius to soften the design. Listing 6.4 shows our card rule.

Listing 6.4 Styling the card

```
.card {
  background-color: var(--card-background);
  box-shadow: 0px 0px 55px rgba(38, 40, 45, .75);
  width: 75vw;
  max-width: 500px;
  border-radius: 4px;
}
```

Figure 6.5 shows the styles applied to our project. With some basic styles added to the card (we will add to these a little bit further into the chapter), let's place the card in the middle (both vertical and horizontal) of the screen.

Figure 6.5 Starting to style the card

6.3.1 Using flexbox to center elements

To center the card in the exact middle of the screen we are going to use flexbox. Flexbox allows us to place elements across a single axis either vertically or horizontally.

The `display` property with a value of `flex` is used to dictate that elements should be placed on the screen using flexbox. The declaration is set on the parent of the elements that will be flexed.

In our project the element being positioned is the card, its parent is the `body` tag, therefore we will add the `display: flex` declaration to our `body` rule.

We then define how we want the elements within the `body` to behave. In our case we have 1 child (the card) and we want it centered. To center the card horizontally we add a `justify-content: center` declaration to the body rule. This property allows us to dictate how elements are distributed across our axis. Figure 6.6 breaks down the options available to us.

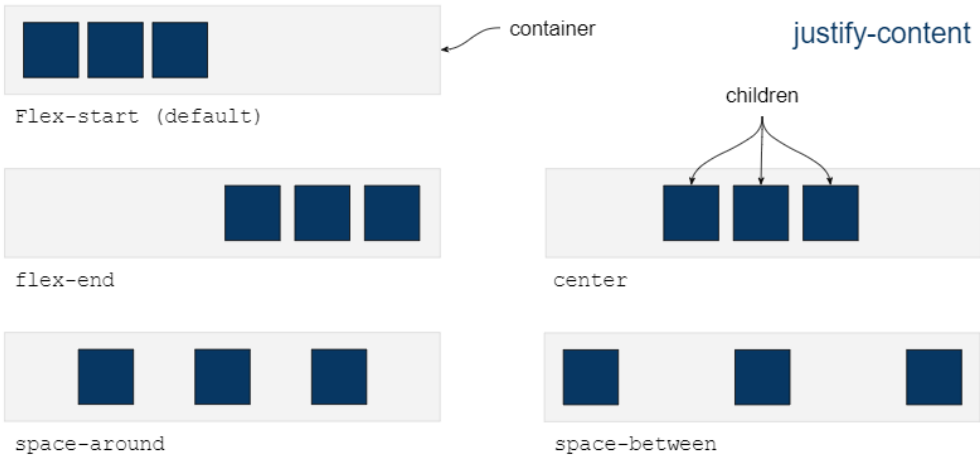

Figure 6.6 `justify-content` **property values**

We also want to center the card vertically. For the vertical positioning we will use `align-items: center`. The align-items property gives us the ability to dictate how elements should be positioned relative to one another and to the container as seen in Figure 6.7.

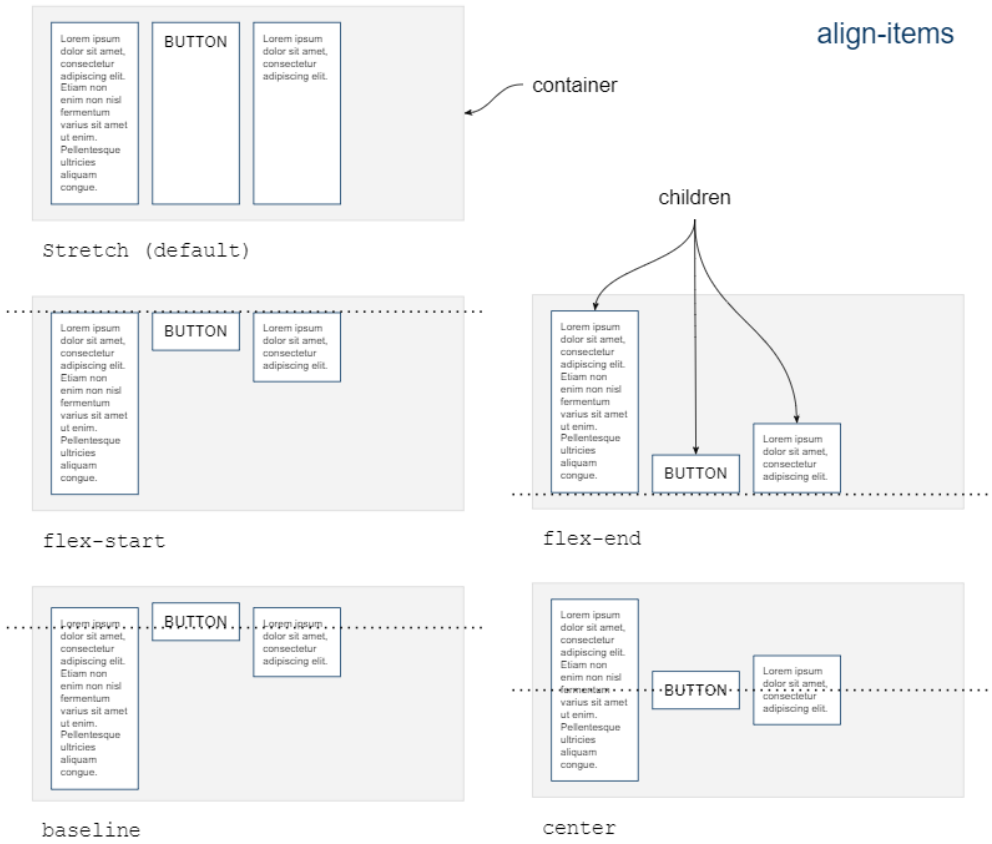

Figure 6.7 align-items property values

Listing 6.5 shows our updated body rule. Remember the parent of the element being positioned is the one on which we apply flexbox related declarations.

Listing 6.4 Centering the card

```
body {
  ...
  display: flex;
  justify-content: center; #A
  align-items: center; #B
}
```

#A centers the card horizontally
#B centers the card vertically

Now that our card is centered (Figure 6.8), let's focus on the contents of the card starting with the profile picture.

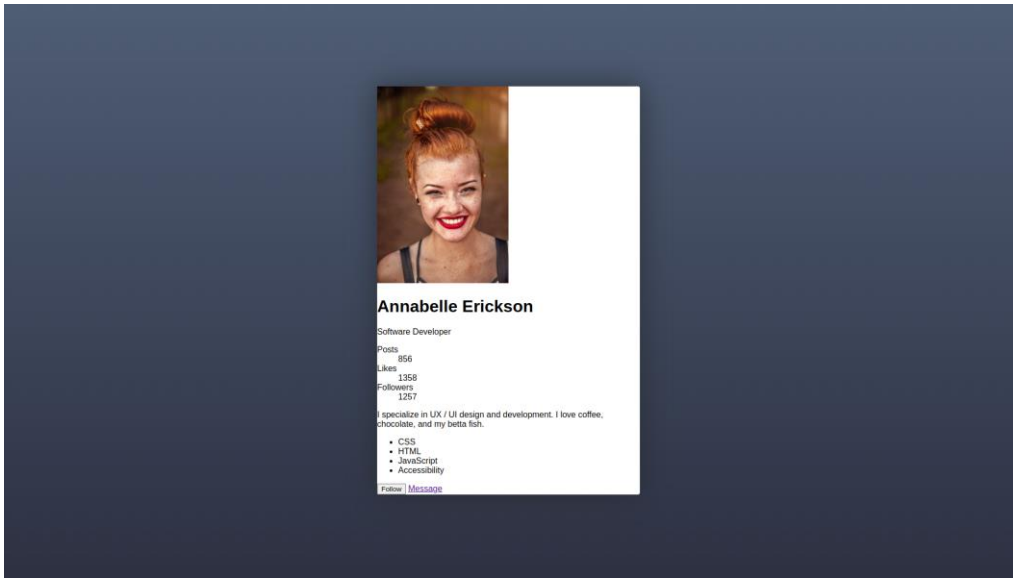

Figure 6.8 Centered card

6.4 Styling and positioning the profile picture

We currently have a rectangular image. We want to make the image circular. We also want to center it in the card and have it stick out the top of the card a little bit.

Let's start by making it into a circle.

6.4.1 Object-fit

A circle's height is equal to its width so as we can see in Figure 6.9, if we set the height and width of the picture to equal our image size variable, the picture will distort.

Figure 6.9 Distorted profile picture

To prevent the image from distorting we must also dictate how the image behaves in relation to the size it is given. To do this we will use the `object-fit` property. By setting the `object-fit`'s value to `cover`, we instruct the image to maintain its initial aspect ratio but fit itself to fill the space available. In our case this means that we will lose a little bit of the top and bottom of the image due to the image being taller than it is wide.

By default when using `object-fit`, the image will be centered and if some of the image is clipped, it will be the edges, which works great for our current use case and picture. However if we wanted to adjust the position of the image within its allotted size and clip only from the bottom, for example, we would also add an `object-position` declaration.

To make our image a 200px wide circle we use the CSS seen in Listing 6.5. Remember that we set the image size as a CSS custom property in the body, so set the width and height of the image equal to the `--imageSize` variable. We add the `object-fit` declaration to prevent the image from distorting and finally we give it a 50% `border-radius` to make it a circle.

Listing 6.5 Centering the card

```
body {
  ...
  --imageSize: 200px;
       }

img {
  width: var(--imageSize);
  height: var(--imageSize);
  object-fit: cover; #A
  border-radius: 50%;  #B
}
```

#A Prevents the distortion
#B Makes the image a circle

Our image now looks as seen in Figure 6.10.

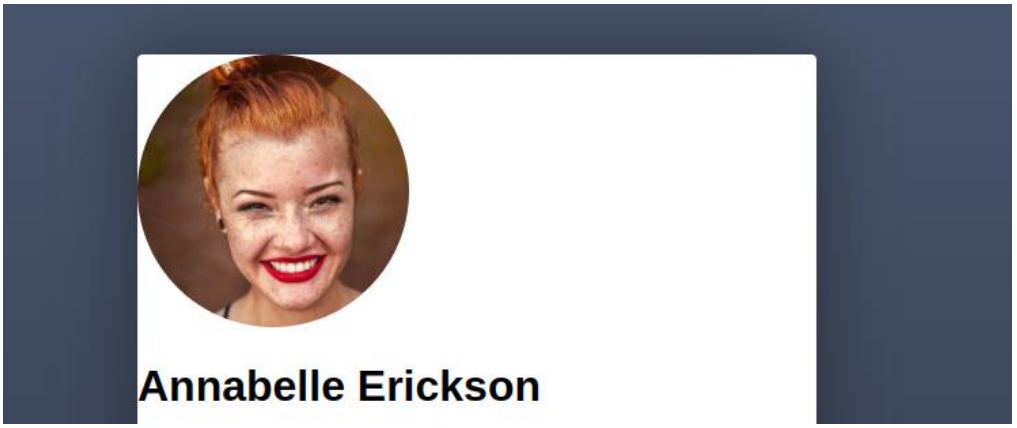

Figure 6.10 Circle profile picture

Now we need to position our picture.

6.4.2 Negative margins

To position our image to stick out above the card we are going to use a negative margin. To move an element down and away from content above it, we can add a positive margin-top value to the element. However if we add a negative margin, instead of the element being pushed down, it will be pulled upwards. We are going to leverage this in conjunction with text centering to position the image.

Looking back at the final design Figure 6.11, we notice that all of the text is centered.

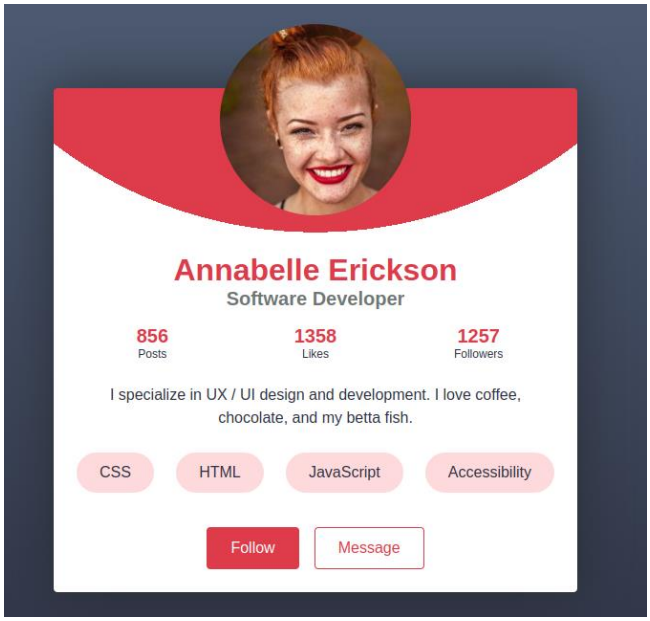

Figure 6.11 Final design

Since all of the text is centered, let's add a `text-align: center` declaration to the card rule. Images by default are inline elements, so we notice that by centering the text, the image also gets centered (Figure 6.12).

Figure 6.12 Centered text

Now all that is left is to add the negative top margin to move the image upward. We want ⅓ of the image to stick out from the top. We will therefore use the `calc()` function to do the math for us. Our function looks as follows: `calc(-1 * var(--imageSize) / 3);` We divide

the image size by 3 to get ⅓ of the height of the image and then multiply by -1 to make it negative. Our margin will therefore make ⅓ of the image stick out from the top of the card as seen in Figure 6.13.

Figure 6.13 Positioned Image

The last thing we need to do is give our card itself some margin. Due to the negative margin we added to the image, if we have a very short screen (Figure 6.14) we see that the top of the image disappears off screen.

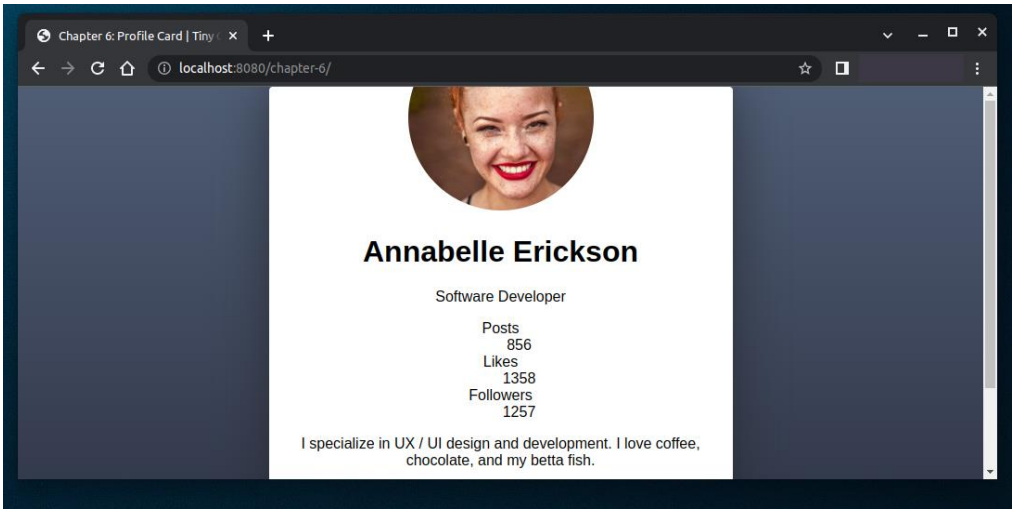

Figure 6.14 Clipping off the top of the image when window height is small

To prevent cutting off part of the picture when the window is not very tall, we want to add some vertical margin to the card itself greater than or equal to the amount of the picture that is sticking out of the card.

To calculate the amount sticking out we used `calc(-1 * var(--imageSize) / 3);`. For our card margin we are going to use a similar concept. We are going to take ⅓ of the image

height and then add 24 pixels to move the card and image away from the edge. Our final function will be: `calc(var(--imageSize) / 3 + 24px)`.

Listing 6.6 shows the CSS we added to position the image.

Listing 6.6 Positioning the Image

```
.card {
  ...
  text-align: center;
  margin: calc(var(--imageSize) / 3 + 24px) 24px; #A
}

img {
  width: var(--imageSize);
  height: var(--imageSize);
  object-fit: cover;
  border-radius: 50%;
  margin-top: calc(-1 * var(--imageSize) / 3); #B
}
```

#A Vertical margin of ⅓ image size + 24px and horizontal of 24px
#B Negative top margin to make the image stick out of the card

With our image positioned and margins added so that the top of the image does not get cut off on small screens (Figure 6.15), let's turn our attention to the curved red background under the picture.

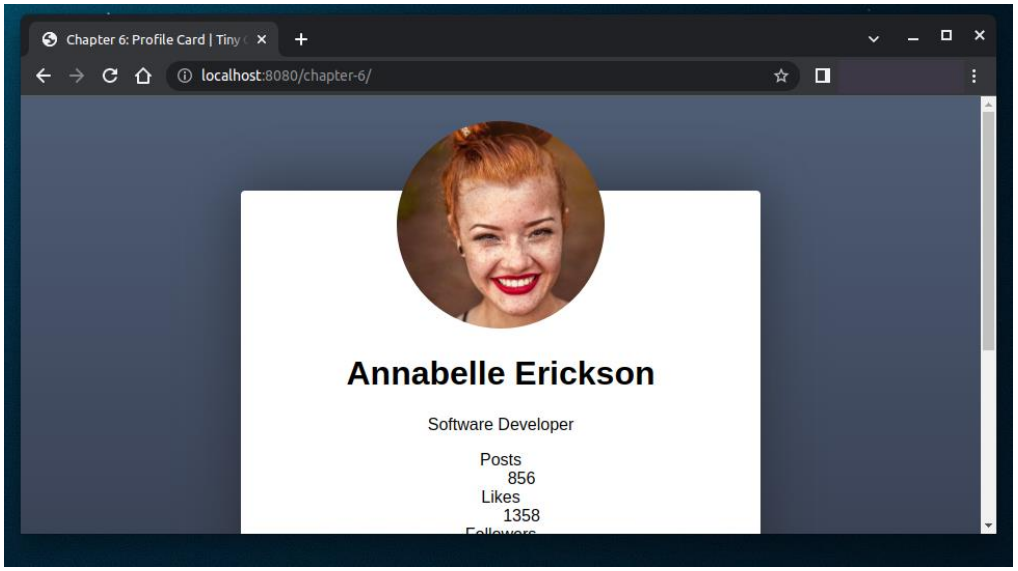

Figure 6.15 Added card margin

6.5 Background size and positioning

To create the background behind the image we are going to follow the steps diagrammed in Figure 6.16.

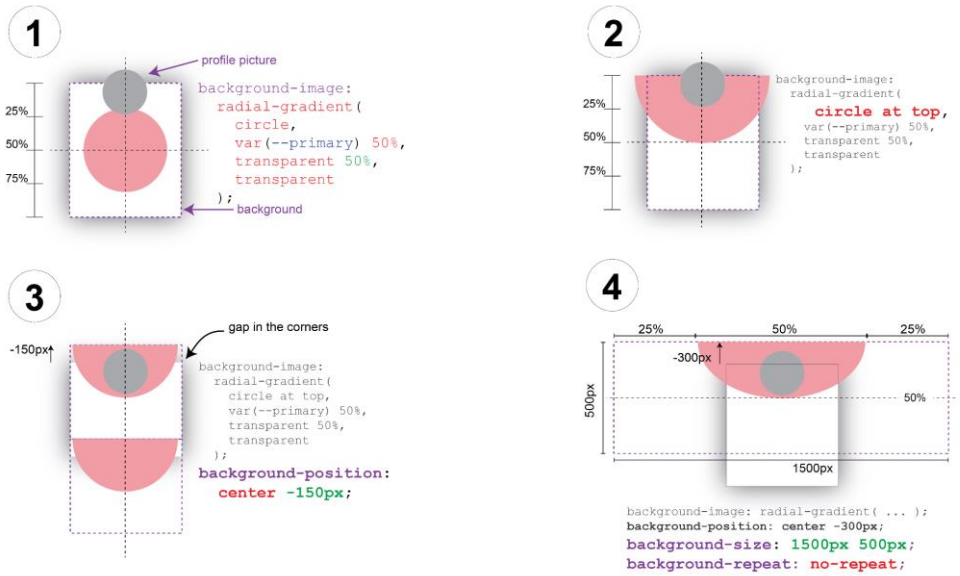

Figure 6.16 Creating the image background

First, we add a background-image consisting of a radial-gradient.

> **NOTE: COMBINING BACKGROUND COLOR AND IMAGE** We can add both a background color and a background image to the same element. We assign the color to the `background-color` property and the image to the `background-image` property. Or we can apply both in the `background` shorthand property as follows: `background: white url(path-to-image);`

The radial gradient takes an ending shape (circle or ellipse) and then where we want each color to start and stop to form the gradient. We define ours as `radial-gradient(circle, var(--primary) 50%, transparent 50%, transparent);`. Our primary color is a dark salmon so our gradient will create a circle that up until it reaches 50% of its container will be dark salmon. At 50% of the container size it immediately shifts the color to transparent. Since it's an immediate shift in color there is no fade, so we get a nice clean circle.

By default, radial gradients emanate from the center of their container. In step 2 we add `circle at top` to the beginning of our `radial-gradient` function to shift the origin of the circle from the center of the background to the top. Our updated radial-gradient function

therefore now looks as follows: `radial-gradient(circle at top, var(--primary) 50%, transparent 50%, transparent);`

Next we want to move the circle upwards so that the bottom of the circle comes just under the image. Step 3 shows us that if we move the background up -150 pixels and our card is rather short (our profile does not have a lot of content) we will end up with a gap on each of the top corners between our circle and the edge of the card which we don't want. To prevent this from happening we are going to make the background image 3 times wider than the maximum card size ($3 \times 500 = 1500$).

> **NOTE: NEGATIVE BACKGROUND POSITION** Unlike the image, where by adding negative margin, the visibly extends out of the card, negative background positions do not render the overflow image portions visible. Excess image from an oversized background or changed positioning will simply be hidden.

When we create a background-image using gradient, the background image produced will grow and shrink with the container, we are therefore also going to give the background a set height so that no matter how much content is in the card, the shape of our background will be predictable.

After changing the dimensions of the background we also increase by how much we move the background up to snuggly end just below the profile image

Finally as mentioned earlier in the chapter, background images repeat by default. By moving the image up, we leave room for the background to tile. We only want the one semi circle so we set the background-repeat to no-repeat.

All put together, our card background is defined as follows (Listing 6.7):

Listing 6.7 Positioning the Image

```
.card {
  background-color: var(--card-background);
  ...
  background-image: radial-gradient(circle at top, var(--primary) 50%, transparent 50%,
       transparent); #A
  background-size: 1500px 500px; #B
  background-position: center -300px; #C
  background-repeat: no-repeat; #D
}
```

#A Creates a semi-circle whose flat side is the top of the card
#B Sets the dimensions of the background image to 1500px wide and 500px tall
#C Positions the background be horizontally centered and start 300px above the card
#D Prevents the background from tiling

Figure 6.17 shows the background added to the card.

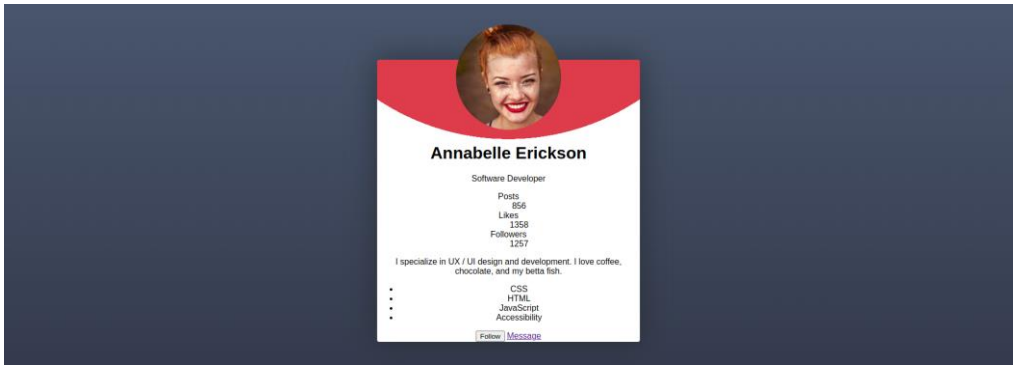

Figure 6.17 Finished background image

With the top of our card starting to look pretty good let's focus on the rest of the content.

6.6 Styling the content

Our card does not currently have any padding, which means that if our name was longer it could potentially touch edge to edge on our card. Since in most cases, a card would be something we would create as a component or template to reuse across multiple users, let's add some left and right padding to ensure our text does not run into the edge of the card. We will also add some bottom padding to move the links and bottom away from the bottom edge of the card.

Listing 6.8 shows our updated card rule and Figure 6.18 shows the new output. We use the shorthand property which defines 3 values. It states the top padding is 0, left and right are 24px, and bottom is 24px. We specifically do not add padding to the top as it would push the image down and we would have to readjust our image positioning.

Listing 6.8 Added padding to the card

```
.card {
  ...
  padding: 0 24px 24px;
}
```

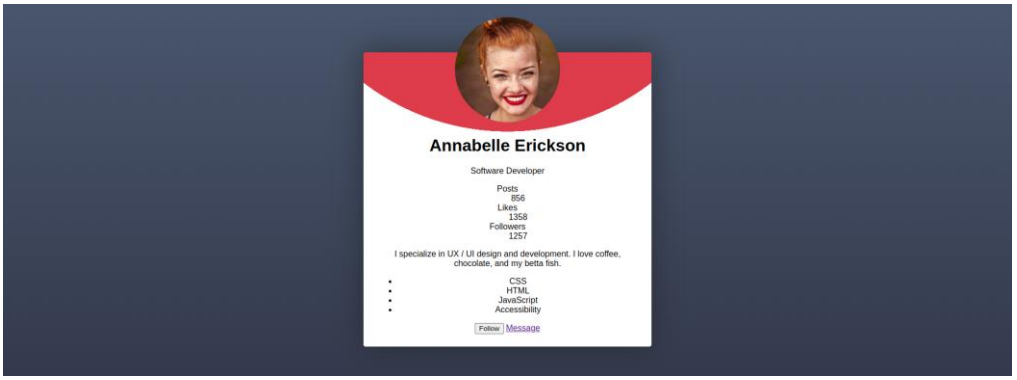

Figure 6.18 Added card padding

6.6.1 Styling the name and job title

Going down the card, our first piece of content is the name. As an `h1` it has some default styles provided by the browser which include some margin. We are going to edit the margin to increase the amount of room between the header and the image and remove the bottom margin so that the job title can come directly below the name. We will also change the color to salmon and set the font size to `2rem`.

Rem Unit

Rem is a relative unit based on the font size of the root element, in our case HTML. For most browsers, the default is 16px. We did not set a font size on the HTML element in our project, therefore when we set the h1 font-size to 2rem, the output size will be 32px.

The benefit of using relative font sizes is accessibility. They help ensure that the text can scale gracefully regardless of the user's settings or device.

For the job title we will increase the size and weight and change the font color to use our secondary color which is gray.

Listing 6.9 shows our new rules and Figure 6.19 shows our output.

Listing 6.9 Styling the name

```
h1 {                        #A
  font-size: 2rem;          #A
  margin: 36px 0 0;         #A
  color: var(--primary);    #A
}                           #A

.title {                     #B
  font-size: 1.25rem;        #B
  font-weight: bold;         #B
  color: var(--secondary); #B
}                           #B
```

#A Styles for the name
#B Styles for the job title

Figure 6.19 Styled Name and Job title

Next we are going to style the post, like, and follower information.

6.6.2 Space-around and gap

In our HTML, the description list (dl) contains the post, like, and follower counts (Listing 6.10). Each grouping is contained within a div, so we will apply a display value of flex to the definition list in order to horizontally align all three groups. We will then set the justify-content property to space-around in order to spread them out across the card.

Listing 6.10 Description list HTML

```
<dl>
  <div>
    <dt>Posts</dt>
    <dd>856</dd>
  </div>
  <div>
    <dt>Likes</dt>
    <dd>1358</dd>
  </div>
  <div>
    <dt>Followers</dt>
    <dd>1257</dd>
  </div>
</dl>
```

The space-around value distributes the elements evenly across our axis by providing an equal amount of space between each element and half as much on each end. Figure 6.20 shows a diagram how the spacing is applied.

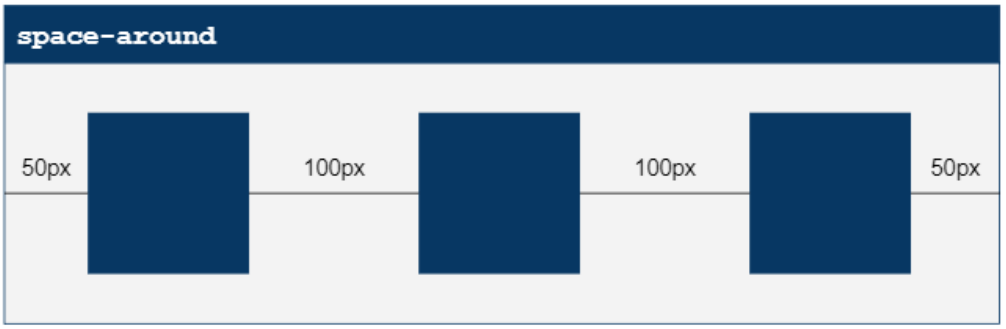

Figure 6.20 space-around property

Listing 6.11 shows our styles for the description list. Notice we also included a `gap: 12px` declaration. This ensures that the minimum amount of space between our elements will be 12 pixels. We could have given our divs inside of the description list a margin; however, those would have also affected the outer edges. Gap only affects the space between elements.

Listing 6.11 Styling the name

```
dl {
  display: flex;
  justify-content: space-around;
  gap: 12px;
}
```

As seen in Figure 6.21, our profile stats are now in a row and evenly spaced across the card.

Figure 6.21 Aligned profile stats

The numbers, however, are offset. This comes from the description which has some margins that come from the browser defaults. Let's get rid of those and style the text to be bold, bigger, and salmon using the CSS from Listing 6.12.

Listing 6.12 description details rule

```
dd {
  margin: 0;
  font-size: 1.25rem;
  font-weight: bold;
  color: var(--primary);
}
```

With the margin removed (Figure 6.22) we notice that the likes are not exactly centered to the middle of our card.

Figure 6.22 Description list alignment

The reason that the likes are not centered in the middle of the card is because each of the three elements do not have exactly the same width. When the elements are distributed, the browser calculates the total amount of space each element needs, and then redistributes the

leftovers equally. Therefore, since the `div` containing followers is larger than the `div` containing posts, likes does not land in the middle.

6.6.3 Flex-basis and flex-shrink

To center the likes, we will assign all three `div`s the same width. Instead of using the `width` property, however, we will use `flex-basis` and set its value to `33%`. `flex-basis` sets the initial size the browser should use when calculating the amount of space the element needs. We will also set `flex-shrink` to `1`.

`flex-shrink` dictates whether an element is allowed to shrink to a smaller size than the size assigned by the `flex-basis` value if there is not enough room in the container for the element. If the `flex-shrink` value is `0`, the size is not adjusted; any positive value allows for resizing.

We set our `flex-basis` to `33%` but remember: we also set a `gap` of 12 pixels between each of our elements. Therefore, the `flex-basis` size we set is too wide for the container once the `gap` is taken into consideration. By allowing the elements to shrink, we tell the browser to start its positioning calculations with each `div` taking up 33% of the width of the container and to evenly shrink them to fit into the available space. This prevents us from having to do math of figuring out exactly how wide the `div`s should be and still have them equally sized.

To write our rule (Listing 6.13), we therefore target the divs that are immediate children of the definition list (`dl`) using a child combinator (`>`) and apply the `flex-basis` and `flex-shrink` declarations.

Listing 6.13 Centering the likes

```
dl > div {
  flex-basis: 33%;
  flex-shrink: 1;
}
```

With our likes centered (Figure 6.23), let's turn our attention to the definition terms (`dt`).

Figure 6.23 Centered Likes

6.6.4 Flex-direction

In our original design, we have the description details (the numbers) above the description terms. To flip them visually we are going to use the `flex-direction` property. We asserted that flexbox can place elements across a singular axis. So far we have done our work across the horizontal or x axis.

To move the details above the terms we are going to use flexbox on the vertical or y axis. To change which axis we want flexbox to operate on we use the `flex-direction` property. By default it has a value of `row`, which makes flexbox operate on the x axis. By changing the value to `column`, we make it operate on the y axis.

Furthermore, the `flex-direction` property also allows us to dictate how the elements should be ordered. Setting the value to `column-reverse` tells the browser that we want to operate on the y axis, and we want the elements to be placed in reverse HTML order, making the description details (`dd`) appear first and the description term (`dt`) second.

As before, we want to set the behavior on the parent, in this case the `div`. We will therefore add to our previously created `div` rule as seen in Listing 6.14 to reorder the elements. We also decrease the size of the description term (`dt`) to emphasize the number over its term.

Listing 6.14 Reversing content order

```
dl > div {
  flex-basis: 33%;
  flex-shrink: 1;
  display: flex;
  flex-direction: column-reverse;
}
dt { font-size: .75rem; }
```

> **NOTE: CONTENT ORDER** For accessibility reasons we want to make sure that the order in which our HTML is written follows the order in which it is displayed on the screen. A user who has their computer read the contents of the page to them as they are visually following along would be easily disoriented or confused if the content that is being read to them does not match what they are looking at. Caution must therefore be used when using properties such as flex-direction to reorder content.

Figure 6.24 shows our styled description list (`dl`).

Figure 6.24 Styled description List

Continuing down the card, let's turn our attention to the summary paragraph below the profile stats.

6.6.5 Styling the paragraph

The paragraph already looks pretty good. The only thing we are going to do to it is add some vertical margin for breathing room and increase the line height for better legibility as seen in listing 6.15.

Notice that the line height does not take a unit. By not setting a unit we allow the line height to scale with the font size. If we had set it to a 12 pixel value for example, the line height would remain 12 pixels no matter the font size. So if the font size was radically increased, our letters would vertically overlap. It is always safest not to declare a unit.

Listing 6.15 Paragraph rule

```
p.summary {
  margin: 24px 0;
  line-height: 1.5;
}
```

With our paragraph taken care off (Figure 6.25), let's style the list of technologies.

Figure 6.25 Styled summary paragraph

6.6.6 Flex-wrap

The first thing we are going to do is style the list elements themselves. We will use a design pattern sometimes referred to as a pill or chip where the element has a background color and rounded edges. Our CSS will look as seen in listing 6.16. We also include some padding in order for the text not to come right up against the edge of the chip.

Listing 6.16 Styling the list elements

```
ul.technologies li {
  padding: 12px 24px;
  border-radius: 24px;
  background: #ffdadd;
}
```

With the individual elements styled (Figure 6.26) we can now focus on the list's layout.

Figure 6.26 Styled List Items

The first thing we are going to do is remove the bullets using `list-style: none`. We will then remove all padding and set the margins to 24px vertically and 0 horizontally.

To position the items we will use flexbox, add a `gap` of 12px and set the `justify-content` property value to `space-between`. `space-between` works very similarly to `space-around` except that it does not add space to the beginning and end of the container as seen in Figure 6.27.

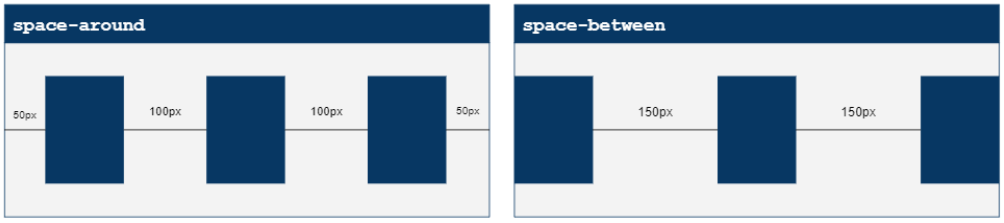

Figure 6.27 space-around versus space-between

Our rule to layout our chips will looks as follows (Listing 6.17):

Listing 6.17 Styling the list of technologies

```
ul.technologies {
  list-style: none;
  padding: 0;
  margin: 24px 0;
  display: flex;
  justify-content: space-between;
  gap: 12px;
}
```

However, we notice that when we reduce the screen width (Figure 6.28), our last chip extends beyond our card.

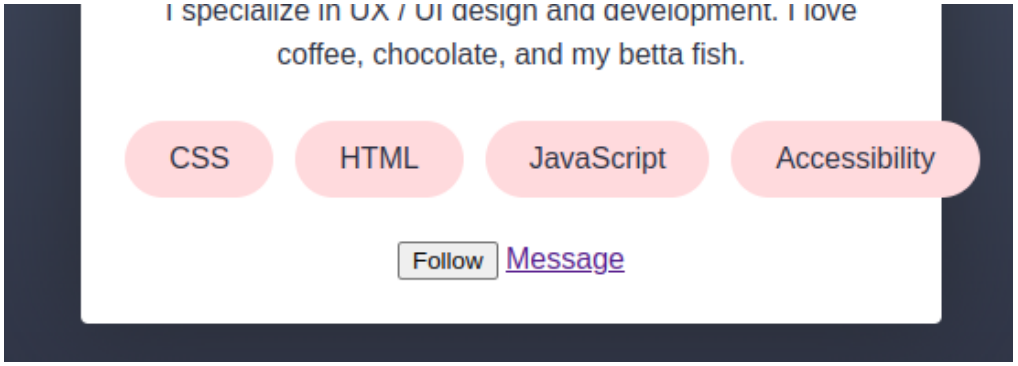

Figure 6.28 Chip extending beyond width of the card

On narrow screens our list is wider than our card. To prevent the content from overflowing the card, we can use the `flex-wrap` property.

By default, flexed items will display in a straight line even if the container is too small as we are experiencing with our list of technologies. To force the last element onto a new line when we run out of room, we can set the `flex-wrap` property to `wrap`. This will tell the browser to start a new line of items below when it runs out of room.

Like `flex-direction`, `flex-wrap` can also change the order in which the elements are displayed, but we won't need to change that here. Our updated rule is shown in Listing 6.18.

Listing 6.18 Adding flex-wrap

```
ul.technologies {
  list-style: none;
  padding: 0;
  margin: 24px 0;
  display: flex;
  justify-content: space-between;
  gap: 12px;
  flex-wrap: wrap;
}
```

Notice the gap between the CSS and Accessibility chips in Figure 6.29 even though our list element does not have any margin. Our list has a `gap` property value of `12px`, this means that not only will it ensure we have a minimum of 12 pixels horizontally between our items, when we wrap, it will also add a 12 pixel gap between them vertically.

Figure 6.29 Wrapping the chips on narrow screens

6.6.7 Styling the actions

The last thing we need to style in our profile card are the 2 actions the user can take at the bottom of the card: to message or to follow the profile owner.

Even though they are semantically different, one is a link and the other is a button, we are going to style both of them to look like buttons. Let's start with some basics that will apply to both elements. We create one rule with selectors for both elements in order to ensure that both element types are visually consistent. We then create individual rules for where they diverge.

We also create a focus-visible rule which will be applied to all elements using the universal selector (*) and the pseudo-class :focus-visible so that when a user navigates to our links and buttons via keyboard, they are presented with a dotted outline around the element and can clearly see what they are about to select.

Listing 6.18 Adding flex-wrap

```
.actions a, .actions button {            #A
  padding: 12px 24px;
  border-radius: 4px;
  text-decoration: none; #B
  border: solid 1px var(--primary);
  font-size: 1rem;
  cursor: pointer;
}

.follow {
  background: var(--primary);
  color: var(--primary-contrast);
}

.message {
  background: var(--primary-contrast);
  color: var(--primary);
}

*:focus-visible {
  outline: dotted 1px var(--primary);
  outline-offset: 3px;
}
```

#A applies to both the link and the button
#B removes the underline

Figure 6.30 shows our styled link and button. However, as they are quite close together, we are going to want to add some space between the 2 buttons.

Figure 6.30 Styled actions

We want to keep the link and button centered but add some space between them. Let's use flex and gap one last time to position our action elements.

We are going to give the list a `display` property value of `flex` and add a `gap` of `16px`. To keep the 2 elements centered, we will use the `justify-content` property with a value of `center`. Finally, we will add some space between the list of technologies and our actions by giving the list a `margin-top` value of 36 pixels as seen in Listing 6.19.

Listing 6.19 Positioning the link and button

```
.actions {
  display: flex;
  gap: 16px;
  justify-content: center;
  margin-top: 36px;
}
```

With this last rule we have finished styling our profile card. The final product is shown in Figure 6.31.

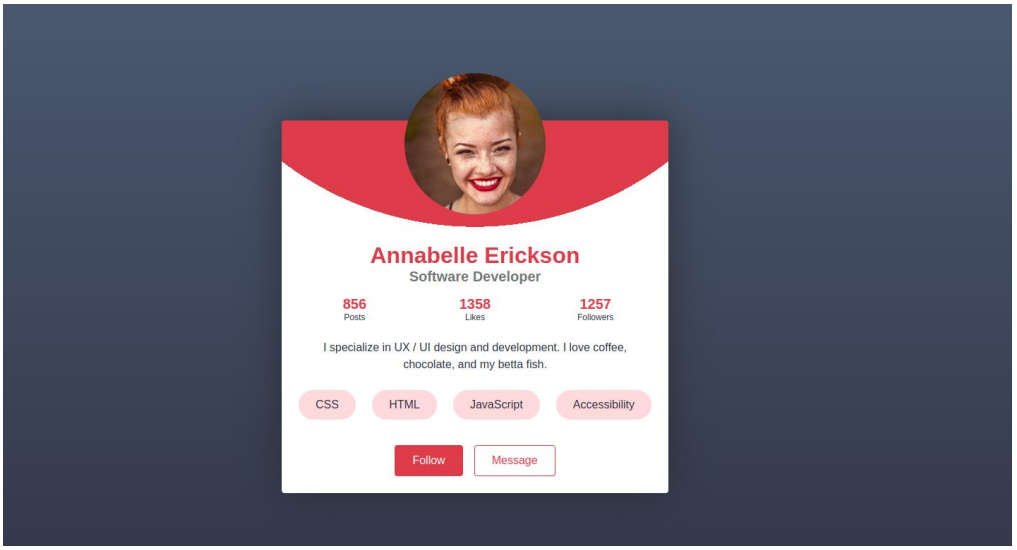

Figure 6.31 Finished Profile Card

6.7 Summary

- CSS custom properties are allow us to set variables that can be reused throughout our CSS
- Flexbox allows us to position elements on a single axis either horizontally or vertically
- `flex-direction` sets which axis flexbox will operate on
- Both `flex-direction` and `flex-wrap` can alter the order in which the elements are displayed
- The `align-items` property sets how the elements are aligned on the axis relative to one another
- The `justify-content` property dictates how the elements positioned and leftover space will be distributed within the element to which it is applied.
- `flex-basis` sets a starting element size for the browser to use when laying out flexed content

- `flex-shrink` dictates whether and how content can shrink when an element is being flexed
- We can prevent images from distorting when when using fixed heights and widths that do not match the image's aspect ratio by using the `object-fit` property