Master the Skill of

# Debugging CSS

Ahmad Shadeed

Tips and techniques on how to debug CSS the right way with easy and studied methods

Foreword by John Allsopp

**Editor:** Andrew Lobo

**Proofreader:** Geoffrey Crofte

# Foreword

I've been using, and a big advocate for CSS since before it was even a standard, nearly 25 years.

It's hard to convey what a profound change it brought in developing for the web when introduced, although its widespread adoption took years of the technology maturing in browsers, and of advocacy, changing the long established use of tables for page layout, font elements, and other hacks web developers came up with to design for the Web.

Since then, CSS has matured in ways its originators and early adopters could barely imagine and brings developers incredible power. But this power and complexity come at a cost.

When developing with CSS, I sometimes think of the story of "The Sorcerer's Apprentice" (originally a poem by German Romantic poet Goethe, but made famous by Mickey Mouse in the Disney film Fantasia). The Sorcerer's Apprentice gains access to his wizard master's powers, but unable to wield them correctly, causes mayhem.

Which sounds like a lot of developing with CSS to me!

The Cascade, Specificity, Inheritance are all powerful features of CSS, but also cause many of the problems we associate with the language.

Which is why I'm surprised it's taken so long for someone to really address the significant challenges of debugging CSS. And why I'm excited for Ahmad's new book, which addresses this important topic in detail.

I really recommend this to any web developer, it's long over due!

**John Allsopp — Web Directions**

# Table of Contents

# Chapter 1

## Introduction and Overview

1

Let's face it: The process of debugging CSS is not straightforward, because there is no direct or clear way to debug a CSS problem. In this book, you will learn how to sharpen your debugging CSS skills.

For traditional programming languages, such as Java, C, and PHP, the techniques of debugging have evolved over the years. That is not the case with CSS. Debugging CSS is not like debugging a programing language because you won't be alerted to errors at compilation time. You would get silent errors, which are not helpful.

Before debugging a CSS error, you need to spot it first. In some cases, you might receive a report from a colleague that there is a bug to be solved. Finding a CSS bug can be hard because there is no direct way to do it. Even for an experienced CSS developer, debugging and finding CSS issues can be hard and confusing.

This chapter will discuss:

- the history of debugging CSS,
- what has changed today,
- what debugging CSS means,
- the debugging mindset,
- why debugging needs time,
- an overview of this book's topics.

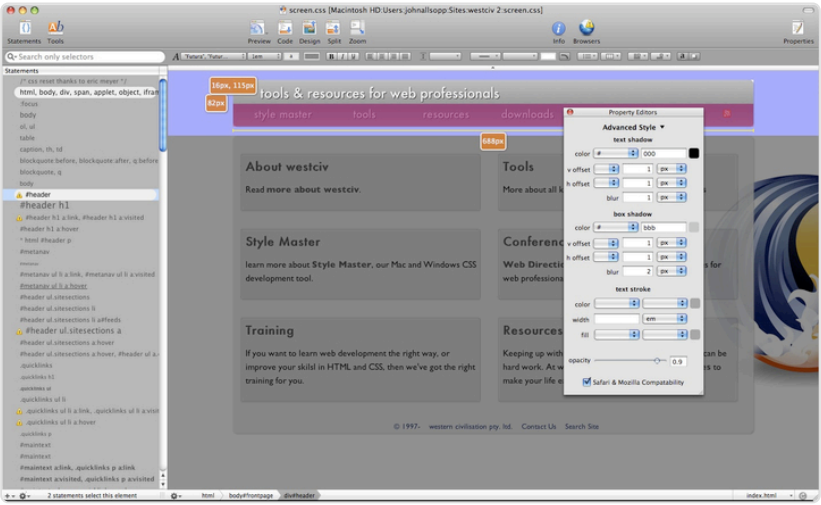
## The History of Debugging CSS

Because this book is about debugging and finding CSS issues, you should be aware of a bit of the history of how debugging tools for CSS have developed over the years.

## Style Master

You might be surprised to hear that the first CSS debugging tool was released in 1998 — 22 years ago! Its creators, John Allsopp and Maxine Sherrin, named it Style Master. As they described it:

> Style Master is the leading cross-platform CSS development tool. Much more than just a text editor, Style Master supports your workflow — including: creating style sheets based on your HTML; live CSS editing of PHP, ASP.NET, Ruby and other dynamically generated sites; editing CSS via ftp; and much, much more.

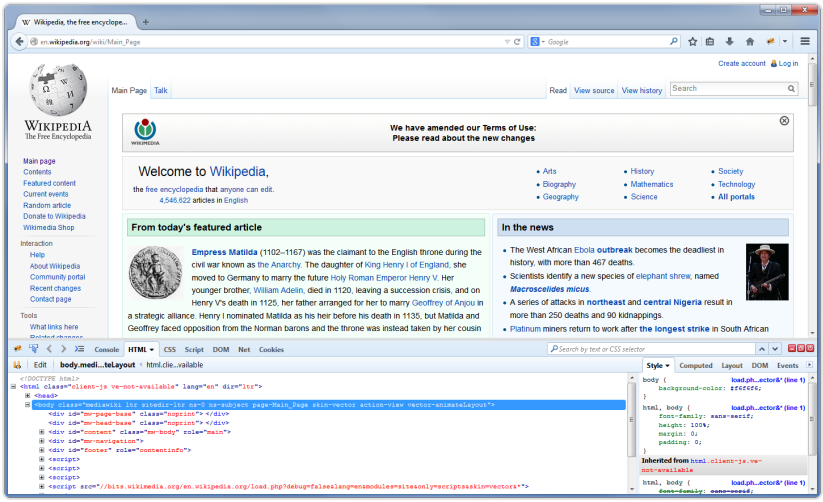



The goal of Style Master was to make working with CSS more efficient, more productive, and more enjoyable. As you can see, then, debugging CSS has been a topic of interest to developers for a long time.

## Firebug Browser Extension

In 2006, Joe Hewitt released the first version of the Firebug browser extension.

> Firebug is a discontinued free and open-source web browser extension for Mozilla Firefox that facilitated the live debugging, editing, and monitoring of any website's CSS, HTML, DOM, XHR, and JavaScript. — *[Wikipedia](https://en.wikipedia.org/wiki/Firebug_(software)*




The main features of Firebug were very similar to what we have in the developer tools (DevTools) of today's browsers:

- inspecting HTML and CSS,
- reviewing the JavaScript console,
- testing web performance.

Without the effort of the folks who created these great tools, Style Master and

Firebug, we might not have the DevTools we use today. We couldn't be more thankful to them.

## What Has Changed Today?

The scene today is dramatically different. Every browser these days has built-in DevTools that make it easy for a developer to inspect and edit the HTML, CSS, and JavaScript of a web page. In this book, we're interested in CSS.

Not to mention, when Style Master and Firebug were released so long ago, websites were very simple, and we had only one screen size to test on. Today, a website can be accessed by hundreds of smartwatches, mobile phones, tablets, laptops, and desktop computers. Debugging for all of these types of devices is not an easy task. You could fix something for mobile and break it unintentionally for the desktop.

It's not only about screen sizes. The size of web projects has gotten much bigger in the last 10 years. For example, the developers of a large-scale front-end project like Facebook or Twitter need a systematic way to test and debug. All of these changes to the work of web development are clear evidence that debugging must be taken care of from day one and that developers must learn it as a core skill.

## What Does Debugging CSS Mean?

> Debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems. — *Wikipedia*

We can use the same definition. Finding and resolving CSS bugs is an essential

skill. Regardless of the programming language you are used to working with, the debugging steps are almost the same for CSS. Later in this chapter, we'll go over a clear strategy for debugging that you can use right away.

When I refer to a "CSS bug" throughout this book, I mean a bug that was caused by the developer, not a bug implanted by the browser. But we will address both types.

## Why Debugging Should Be Taught

The fast development of browser DevTools makes it hard to catch up with all of the techniques and methods of debugging CSS. Not to mention the lack of an organized resource that is easy for a beginner to follow.

## The Debugging Mindset

According to Devon H. O'Dell, in his paper "The Debugging Mindset":

> Software developers spend 35-50 percent of their time validating and debugging software. The cost of debugging, testing, and verification is estimated to account for 50-75 percent of the total budget of software development projects, amounting to more than $100 billion annually.

If you need to fix a bug quickly, you might feel a bit stressed, which could lead you to rush a solution without a clear strategy. That could easily result in confusion and time wasted on things that don't matter.

Any programming language has logical and illogical errors. Take JavaScript. When there is an error in your JavaScript, you can see by checking the browser's console. At least you will have evidence that there is an error, with

the reason for why it happened.

CSS is completely different. You won't get any kind of alert when you've made an error. That alone can make the simplest CSS bug very hard to fix, if you don't think clearly and follow a sensible strategy. A CSS bug could be caused either by the developer, as when a CSS property is improperly used, or by inconsistencies between web browsers.

Moreover, you might be the one responsible for testing a website and uncovering the bugs. So, we're dealing not only with fixing bugs, but with finding them as well.

## Identifying CSS Bugs

Before a customer or someone on your team discovers something broken on the website, you can do some testing and try to **break** the design intentionally. In the fifth chapter, you will learn some methods of intentionally breaking a CSS layout.

## Explaining a Bug to Someone

Have you ever spent hours trying to solve a CSS issue, only to explain it to a friend or colleague and gotten that spark of an idea of how to fix it? That is the effect of explaining a bug to a friend. You got stuck because you didn't take enough time to think deeply about the problem.

When you find yourself going in circles like that, take a break and come back to it later. Fixing issues needs intense focus. If you're working on a solution while your mind is exhausted, you might unintentionally break something else. Avoid that with a break.

## Why Debugging Needs Time

In his excellent blog post, "You've Only Added Two Lines — Why Did That Take Two Days!", Matt Lacey presents some solid reasons for why debugging takes time. Let's go through them.

### An Issue Is Not Clear

Don't expect someone to report an issue in full detail. A vague description is fine. Before asking for more detail, try to understand the issue as fully as possible. Once you've done that, you'll need to reproduce the bug on your machine, and then you'll have a starting point.

### The Symptoms Are Easier to Treat Than the Cause

When working on an issue, it's important to investigate the cause of it, not only the symptoms. As Lacey says:

> If some code is throwing an error, you could just wrap it in a try..catch statement and suppress the error. No error, no problem. Right? Sorry, for me, making the problem invisible isn't the same as fixing it.

Making a bug invisible with a "quick fix" might introduce some unexpected side effects. You have a chance to fix the issue, not to create more problems!

### Focusing on One Path to the Problem

Some issues can be reproduced in multiple ways, not just the reported one. Finding those ways is not only useful to thoroughly solving the issue, but also can provide great insight into how the CSS is written, and whether there are

other spots in the code base where the same issue can be expected to crop up. This can be very helpful for fixing bugs before they reach users.

### Ignoring Side Effects

Fixing an issue is one thing; avoiding side effects from fixing it is another. That's why it's best to fix an issue with the least amount of CSS possible, and with a thorough understanding of possible side effects.

## Write Code That Is Easy To Debug

Poorly organized code can make debugging much harder. For a large web project, the CSS should be divided into components and partial files, which would then be complied with a CSS preprocessor such as Sass, LESS, or PostCSS.

If you decide to write all of you CSS to a single file, don't expect debugging to be easy. You will end up scrolling the large file up and down endlessly. This approach is confusing and not ideal. More bugs tend to crop up in a single-file CSS.

In the next chapter, we'll go through different types of CSS issues, get into details about debugging in the browser, and much more.

## Who Is This Book For?

This book is intended for designers and front-end and back-end developers who are interested in improving their skills in finding and fixing CSS bugs. You should have an intermediate level of knowledge of HTML and CSS.

In some sections, you will need to follow the steps of installing an npm package. Don't worry, that won't require extensive Node.js experience. You'll be able to follow along easily.

## Why I Wrote This Book?

The lack of resources for learning how to find and fix CSS bugs is the primary reason why I wrote this book. As soon as I began researching the topic, I discovered that this topic has been overlooked. There should be a guide that discusses in an easy and straightforward way all of the details related to debugging.

## An Overview of the Book Chapters

Here is an overview of the chapters you'll find in this book:

- Introduction to CSS bugs
- Debugging environments and tools: Browser DevTools, virtual machines, mobile browsers
- CSS properties that commonly lead to bugs
- Breaking a layout intentionally
- Browser inconsistencies and implementation bugs
- General tips and tricks

Now that we're done with introducing the book, let's start debugging CSS!

# Chapter 2

## Introduction to CSS Bugs

## What Is a Bug?

When something is different from what you expect, that is a bug. For example, an icon might not be aligned with its sibling elements, or an image might look weird because it's stretched (its width and height are not proportional to each other). In some cases, what you view as a bug might actually be what is expected. It could be a feature request, or someone has done it on purpose. For this reason, it's worth checking and asking the person about the bug in detail.

### Browsers Are Different

Web browsers are different, and not all browsers support everything in CSS. In the course of your work, you might encounter something that looks like a bug in browser X, whereas in browser Y, it works perfectly. That doesn't mean that browser X is rendering it incorrectly. Sometimes, an issue occurs because a browser vendor has implemented a feature according to the specification, whereas it works in another browser because that vendor didn't implement it incorrectly.

## How to Fix a CSS Bug

Let's walk through the basic steps of what to do when you find a CSS bug or someone on your team points out something that is broken on a page you've worked on.

### Check the CSS

First, check the CSS that is being used. Are you using some cutting-edge

property that is supported only in modern browsers? Or is it something old that would be expected to work in the browser showing the bug?

Is it working in the browser you're building with? Or perhaps someone else built the website, and you don't know whether it works? Well, you'll need to check whether the issue is reproducible in your browser.

## Check Browser Support

Go to Can I Use and check for browser support of the CSS property. If you see that the property is not supported in the browser where the bug appears or that the property is supported only with a vendor prefix, then that might be your answer. Make sure that vendor prefixes are added (if any) and that the browser you are testing supports the CSS property.

## Use the Browser's Developer Tools

Once you establish that the CSS property can be expected to work in the browser that is showing the bug, then it's time to dig into the browser's developer tools (DevTools). Before inspecting the element, you will need to determine what type of issue it is. For example:

- Is it a visual issue, such as a misaligned icon?
- Is it happening within a distinct section or across pages? (The issue could be related to the CSS layout.)

In the next section, we will dig into the details of the types of CSS issues we find and how to debug them using the browser's DevTools.

## CSS Bug Types

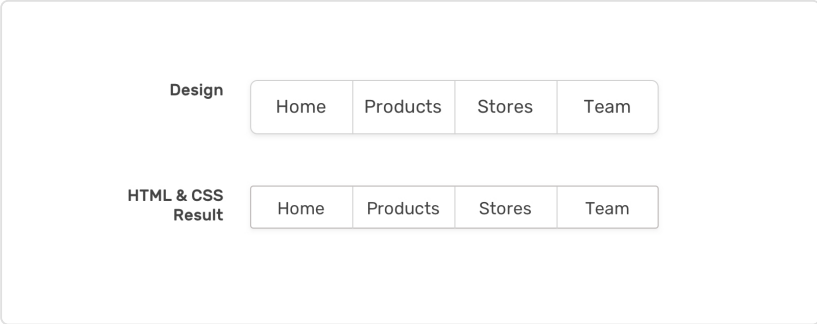
Categorizing bugs by type is helpful. For example, is the issue design-related or related to a syntax error? In this chapter, we will go through each type, along with a basic example.

### Visual Design Bug Types

When you implement a design in HTML and CSS, any obvious inconsistencies between the design and the code can be considered bugs. For instance, have you ever noticed an icon misaligned with its text label, or that the page container is either wider or narrower than the one the proposed in the design? All of these can be considered visual design issues that the developer did unintentionally.

The designer might not know CSS, in which case they would probably take screenshots of the issues and send them back to the developer with notes. If the developer has a design background, then they might be able to easily notice those inconsistencies reported by the designer.

Consider the following figure:




In the navigation design shown above, the first one is the original design, and the second one is the code implementation. The developer put effort into implementing it, but it's still far from the original design, for a couple of reasons:

- The height is shorter.
- The font size is smaller.
- The border radius is less.
- The border color is different.
- The shadow is too light.

We've already spotted five ways in which the implementation is not similar to the design. On a larger scale, a lot of components and sections will need to be crafted carefully to make the implementation look similar to the design. Not all developers notice these design details.

Moreover, issues with visual design include anything that poses an obstacle to the user without being an actual bug. Some examples are an inaccessible color, a confusing organization of content, a misaligned button, text that makes the layout look weird, and inconsistent behavior between website pages. All of these lead to visual design issues and, by extension, accessibility issues.

## Technical Bug Types

Not all issues are noticeable just by looking at a web page. Sometimes you're dealing with a syntax error or an incorrect value for a CSS property. Let's explore the causes of technical issues.

### Calling an Incorrect File Path

Many a developer have spent hours trying to figure out why some CSS is not working at all, only to realize that the cause is an incorrect path for a CSS file. It could happen because you're using a CSS preprocessor such as Sass or LESS, which will render a `.css` file. Sometimes, the rendered file's name is different from the source's. Always be sure that the linked CSS file is the correct one, especially if you have multiple CSS files.

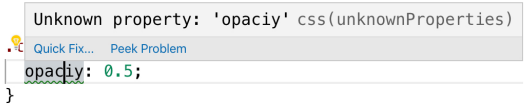
### Misnaming a Property

When you make a typo in a CSS property's name, the browser won't tell you that directly. CSS doesn't throw an error when something is wrong. You need to figure it out by using the browser's DevTools. If you inspect the element, the browser will show the invalid property with a warning triangle and a strike through the name.

I remember working on a simple demo for an article, and I scratched my head trying to figure out why something wasn't working? It turned out that I had a typo when declaring the `opacity` property.

```
.element {
    opaciy: 0.5;
}
```

The reason I didn't notice this trivial mistake is that I was so distracted and didn't think quietly about the reason for the bug. Most code editors will warn you when a property name is mistyped. Here is an example from Visual Studio Code:




A warning in VS code about using an unkown property

### Using an Invalid Value for a Property

Similar to the last issue, this one happens when you give an invalid value to a CSS property. The value could be a typo or one that doesn't work with the given property. Consider the following example:

```
.element {
    opacity: 50;
}
```

The `opacity` property accepts values from 0 to 1. The author here wants 50% opacity but expresses it as a percentage while forgetting the percentage sign. The browser would ignore this `opacity` property.

### Using a Property That Depends on Another

Not all properties work on their own. A property might depend on a certain rule, applied either to the element itself or to a parent or child. Consider this:

```
.element {
    z-index: 1;
}
```

The `z-index` property won't work, because it needs a `position` value other than `static`. The browser wouldn't mark this as invalid, and you'd need to guess why it doesn't work.

Let's consider when a rule must be applied to a parent or child:

```
.child {
    position: absolute;
}
```

We want the child element to be positioned absolutely to its parent. However, the parent doesn't have `position: relative`. This will cause the child to be positioned according to the closest parent that is relatively positioned or to the `body` element.

### Overriding One Property With Another

Sometimes there is no typo or mistake, but you are overriding one property with another. It's just how CSS works, but some developers might think a bug occurred. For example, CSS' minimum and maximum sizing properties can be confusing.

```
.element {
  width: 100px;
  min-width: 50%;
  max-width: 100%;
}
```

Here, the width of the element would be `50%` of its parent. If you haven't read the CSS specification carefully, you might think this is an issue, but it's not.

### Duplicating a Property

Sometimes you'll declare a property, and for some reason it doesn't have an effect on the element. You keep trying and testing with no result. Eventually, you realize that the property is duplicated, and you're editing the first declaration of it, which is being overridden by the second one.

```
.element {
     display: block;
     width: 50%;
     opacity: 1;
     border: 1px solid #ccc;
     opacity: 0;
}
```

The `opacity` property is defined twice here. This is a mistake, and it can happen for various reasons:

- Maybe you copied some styles to test them quickly and forgot to remove the duplicate.
- It could simply mean that you're tired and need to take a break.

Whatever the reason is, it's a bug.

### Incorrectly Typing a Class Name

Your CSS could be 100% correct and valid, but one typo in a class name could lead to styles not being applied to the element. As simple as this is, when we are working for eight hours a day, we tend to focus on big problems and might

overlook such a small mistake.

### Neglecting the Cascade

CSS stands for Cascading Style Sheets. As indicated by the name, a website's styles cascade, and their order matters. If you define a CSS rule for an element and then redefine it at the end of the CSS file, the latter will override the former.

```
.element { color: #000; }
/* 500 lines later… */
.element { color: #222; }
```

This is a very simple example of what can happen. You might face a trickier issue than this. Consider an element that should switch colors on mobile and desktop:

```
@media (min-width: 500px) {
  .element {
    background: #ccc;
  }
}

.element {
  background: #000;
}
```

The background color of `.element` would be `#000` because it comes after (and, thus, overrides) the rule in the media query.

### Forgetting to Bust the Cache

CSS caching happens on the server, not on the local machine. A common problem is pushing an update, and when you refresh the web page, the CSS updates don't appear. In this case, the CSS file might be cached, and you'll need to clear the browser's cache or rename the file after each push.

There are multiple solutions to this problem, the simplest being to add a query string:

```
<link rel="stylesheet" href="app.css?v=1.0.0">
```

And when you make a change, you would also change the version:

```
<link rel="stylesheet" href="app.css?v=1.0.1">
```

Then, the browser would download the latest CSS file. For more information, CSS-Tricks has a great article.

### Neglecting Performance

Using the wrong property for the job can easily impair performance. For example, when animating an element from left to right, the `left` property is a performance killer, because it forces the browser to repaint the layout with each pixel moved.

```
.element:hover {
    left: 100px;
}
```

A better solution would be to use the CSS `transform` property. It won't affect performance, and everything will run smoothly. A simple choice of property can significantly improve performance!

```
.element:hover {
    transform: translateX(100px);
}
```

### Ignoring Specificity

If a CSS rule is not working as expected, the reason could be that its specificity is higher than another's.

```
.title {
    color: #222;
}

.card .title {
    color: #000;
}
```

The specificity of `.card .title` is higher than that of `.title`. As a result, the former would be overridden. To fix this, we can add a variant class to the element, and apply the new color to that.

```
.card-title {
    color: #000;
}
```

Another possibility is using `!important`. Avoid using this in general because it makes maintaining CSS at scale much harder. Use it judiciously and only when needed.

## The Debugging Process

As we've seen, there are many categories of CSS issues. Some are visual, and others non-visual. Now that we've finished listing the common types, the next step is to figure out how to debug, using the various tools and techniques at our disposal.

### Getting Browser Information From Non-Technical People

Suppose that a user reports an issue on your website. As the front-end developer, you've checked your own browser, and everything is OK. So, the issue is appearing only in the user's browser. In such a case, what's the best way to ask a non-technical person for more details? Here are the steps you would normally take:

1. Ask for the browser's name.
2. Ask for the browser version, and explain how the user can get it (for example, "Click on the settings icon, then on 'About', and copy the number at the bottom).
3. Ask for a full-page screenshot. If the user does not know how to do that, recommend to them a browser extension that is easy to use.

A great tool for retrieving browser information is mybrowser.fyi by Andy Bell. The great part is that the user can share an auto-generated link of their browser's information. The following figure shows the visual result:


Once you have the browser's name and version and gotten a visual of the issue, you can start to debug. If you don't have the browser that you need to debug on, then you can either install it on your machine or use an online service, such as BrowserStack.

## Debugging Techniques

When it comes to testing a web page in order to debug CSS, there are a lot of techniques and tools we can use, the most common being these:

- browser's DevTools;
- mobile devices;
- mobile emulators (such as an iOS simulator);
- virtual machines (such as VirtualBox);
- online services (such as BrowserStack and CrossBrowserTesting).

## Wrapping Up

In this chapter, we've defined what a bug is, gone over the different types of CSS bugs, and summarized the debugging process. In the next chapter, we'll dig into the browser's DevTools and learn how to leverage them when fixing CSS issues.

# Chapter 3

## Debugging Environments and Tools

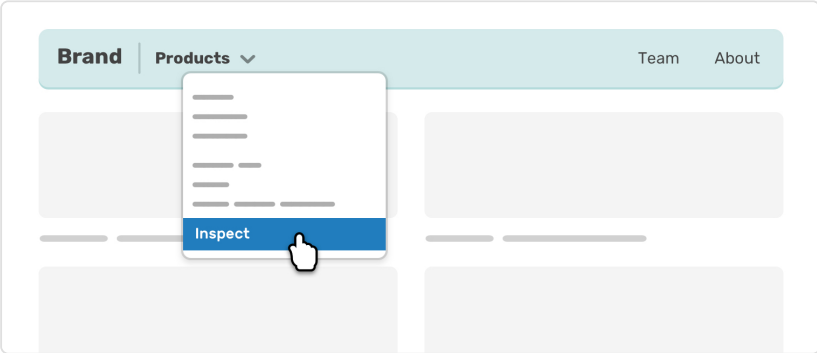
## 3. Debugging Environments and Tools

Every modern web browser has development tools, or DevTools, built in. In the history section, I explained a bit about the tools Style Master and Firebug. Browser DevTools are based on these projects. To open yours, right-click and select "Inspect element" from the menu. If you're a keyboard person, here are the shortcuts for each browser:

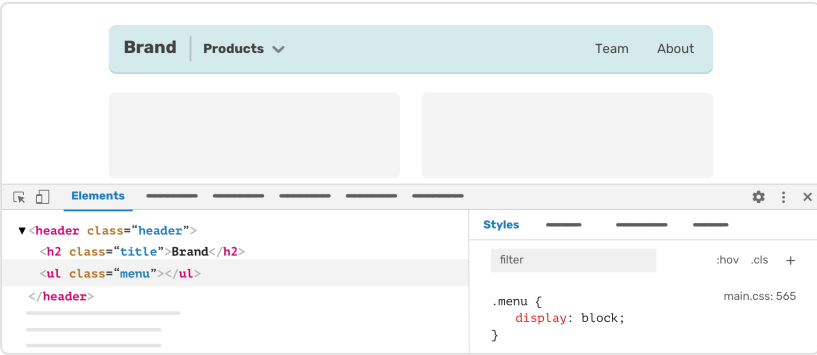
- Chrome: `⌥ + ⌘ + I` on a Mac, and `Ctrl + Shift + I` on Windows
- Firefox: `⌥ + ⌘ + C` on a Mac, and `Ctrl + Shift + I` on Windows
- Safari: `⌥ + ⌘ + I`
- Edge: `⌥ + ⌘ + I` on a Mac, and `Ctrl + Shift + I` on Windows

I will be using Google Chrome in this book, unless I mention another web browser.

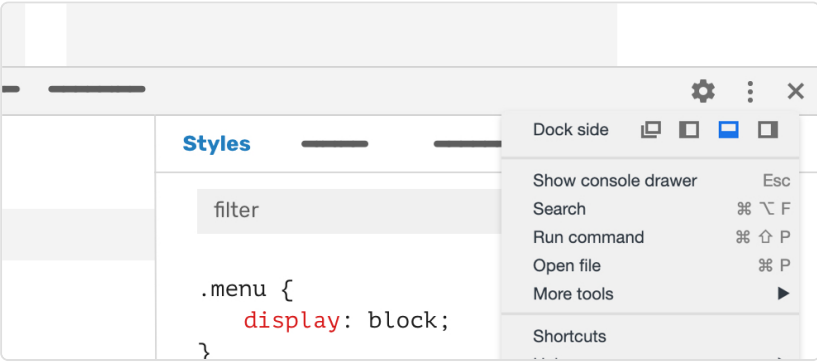
You can inspect any element and toggle its CSS properties. To select an element, right-click and choose "Inspect" from the menu.




When you select "Inspect", the browser's DevTools will open at the bottom of the screen. That's the default position for it. You can pin it to the right or left side of the screen by clicking on the dots icon in the top right.




With the dots clicked, a little dropdown menu will open. You can choose where to pin the DevTools. There is no right place; choose based on your preference. However, you will need to dock it to the right when you are testing at mobile and tablet sizes. This is how it looks:

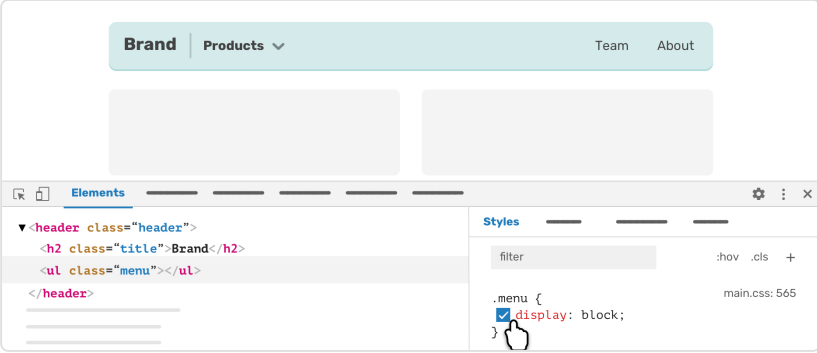



## Toggling a CSS Declaration

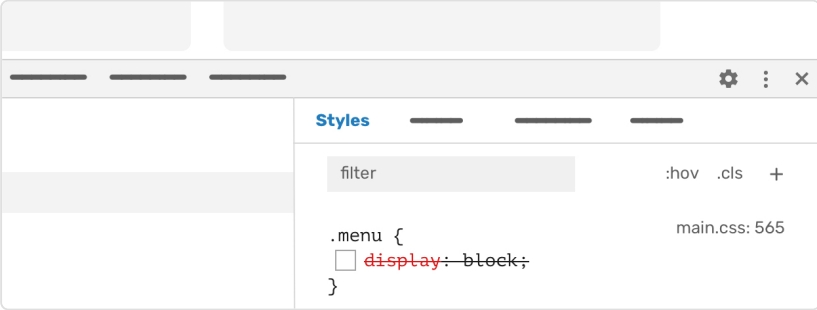
We've opened the DevTools and know how to access them. Let's inspect an

element and play with its CSS at a basic level. With an element inspected, we can toggle its styles with a checkbox (the checkbox is not visible by default).

With an element inspected, look over at the “Styles” tab. You’ll notice that when you hover over a CSS property, a checkbox appears before the CSS declaration. When this box is unchecked, the style will be disabled and won’t be applied to the element.




When you toggle a style off, the checkbox will be visible, to give you a visual hint that it is disabled.

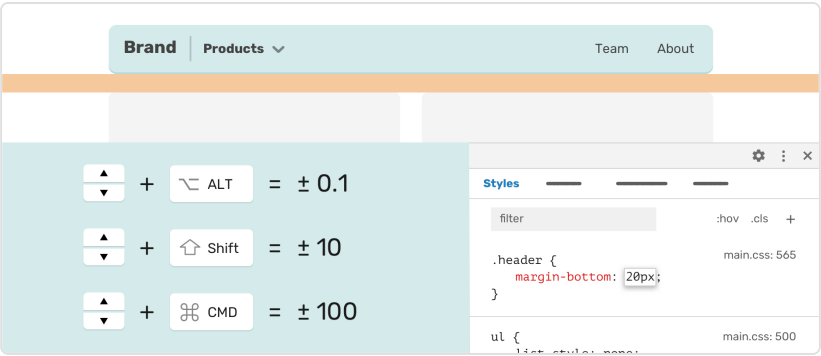


Turning a CSS declaration on and off is similar to commenting CSS. In fact, if you copy a CSS rule with one of its styles toggled off and paste it somewhere, the editor will disable the style by commenting it out with `/* */`. Here is how the CSS will look when copied:

```
.menu {
    /* display: block; */
}
```

## Using the Keyboard to Increment and Decrement Values

In the "Elements" panel, you can select a CSS declaration that has a number, and increment or decrement the value by using the up and down arrow keys. You can also type a value manually.




You can also hold the `Shift`, `Command`, or `Alt` keys with the up and down arrow keys to change numbers with set intervals:

- `Shift` + up/down: ±10
- `Command` + up/down: ±100
- `Alt` + up/down: ±0.1

This is faster than changing one number at a time with the up and down arrows.

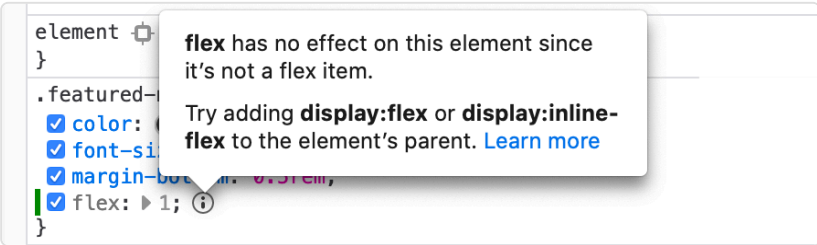
When you change a value and want to exit editing mode, you can do one of the following:

- Click on white space next to the CSS declaration.
- Press the `Escape` key.
- Press the `Enter` key (although this will move to the next declaration in the CSS rule).

## CSS Errors

Given the nature of CSS, debugging is harder when a typo is made or an incorrect value is used for a property. You won't know that a property has a mistyped name until you inspect the element that is showing the bug. The way CSS works is that the browser will parse all declarations and ignore the invalid ones. Compare this to JavaScript, in which an error will break the whole script, and opening the browser's console will make it obvious that something is wrong.

Thankfully, Firefox has a great feature that shows a warning when you use a CSS property that has no effect. At the time of writing, this feature is available only in Firefox.




Hopefully, more browsers will follow!

## DevTools Mobile Mode

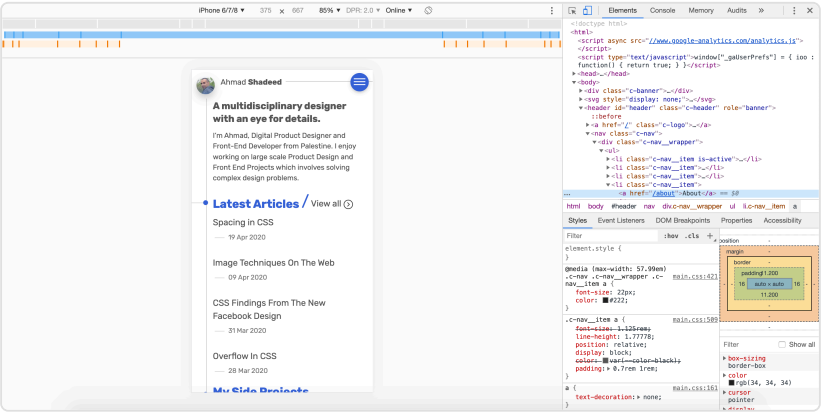
With the browser's DevTools, you can test different viewport sizes of the website you're working on. In this section, we will look at mobile testing topics related to modern browsers (Chrome, Firefox, Safari, Edge).

Suppose you get a message from a client or colleague saying, "Hey, the font size on page X is too small to read on mobile. Can we do something about it?"

From their message, we can determine that:

- the font size is too small to read,
- we need to test in a mobile viewport.

The first thing we'll need to do is fire up the DevTools in the browser, and switch to the device toolbar (in Chrome). You can access the device toolbar by clicking on the mobile icon in the top-left corner of the DevTools or by using the keyboard shortcut ( `Command + Shift + M` ). From there, we can start testing different sizes and, eventually, find the source of the issue.




Other browsers, such as Firefox and Safari, have a device mode but call it "responsive design mode." Here is how to access it:

- Firefox: Tools > Web Developer > Responsive Design Mode
- Safari: Develop > Enter Responsive Design Mode

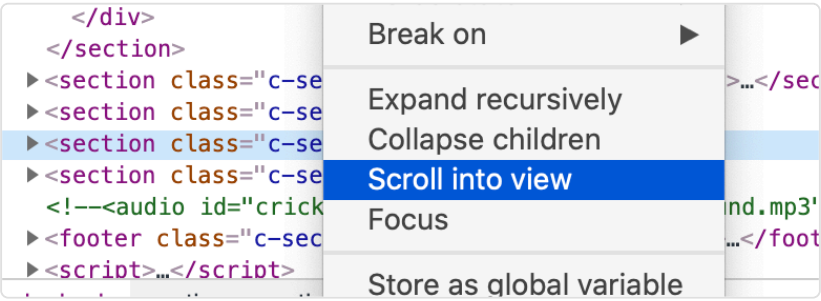
Let's go over some things to keep in mind while testing.

## Mobile Mode Doesn't Show a Horizontal Scrollbar

If an element has a width bigger than the viewport, then horizontal scrolling will take effect. Try to scroll randomly to the left or right. This can reveal any unwanted scrolling issues. Note: This book has a whole chapter on how to break a layout.

## Scroll Into View

While testing a website in mobile mode, the page will usually be very long, and it wouldn't be practical to have to keep scrolling to reach the element you want to inspect. Luckily, Chrome has a feature named "Scroll Into View", which scrolls the page to the section you've selected.




## Screenshotting Design Elements

There will be times when you need to take a screenshot of a page. The tools available online are not all great. Chrome and Firefox have a feature to take screenshots. I particularly like Firefox's feature, named "Screenshot Node", which simply takes a screenshot of the selected HTML element. It's very helpful and a time-saver.

For Chromium-based browsers (Chrome and Edge), the process is:

1. select the element;
2. hit `Shift + Command + P` (make sure no browser extension uses this command);

3. Type “capture node screenshot” and hit “Enter.”

At the time of writing, Chrome Canary 86 supports “capture node screenshot” in the inspector. It will be soon available officially in Chrome.

To take a screenshot in Firefox:

1. open Firefox’s DevTools,
2. right-click on an element,
3. select “Screenshot Node.”

## Device Pixel Ratio

The device pixel ratio (DPR) is the ratio between physical pixels and logical pixels. For example, the iPhone 5 reports a DPR of 2, because the physical resolution is double the logical resolution.

- Physical resolution: 960 × 640
- Logical resolution: 480 × 320

In mobile mode in Chrome’s DevTools, you will find a dropdown menu for the DPR. There are two basic types of screens: standard and “retina.” A `1x` image will look OK on a standard screen but will look pixelated on a retina screen.

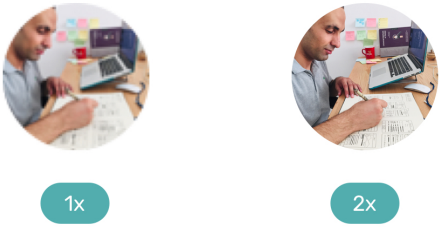
The DPR has three ratios: `1x`, `2x`, and `3x`. Chrome names it as “device pixel ratio”, while Firefox and Safari list the ratios mentioned. The benefit here is that we can test images and simulate how they look at different resolutions.

As Google Developers states:

> To simulate this effect on a standard display, set the DPR to 2 and scale

> the viewport by zooming. A 2x asset will continue to look sharp, while a 1x one will look pixelated.

If you have a standard screen and a `1x` image looks good to you, it's possible to simulate how it would look on a `2x` screen by setting the DPR to 2 or by choosing `2x` as an option and then zooming in once.




In general, use SVG wherever possible. This can't always be done, so if you use images, provide different resolutions for them. For example, you can use the HTML `<picture>` element to load different resolutions and sizes of the same image. The browser will then serve a resolution suitable to the screen's size.

## Switching the User Agent

According to Mozilla Developer Network (MDN):

> The User-Agent request header is a characteristic string that lets servers and network peers identify the application, operating system, vendor, and/or version of the requesting user agent.

The user agent enables the server to identify the browser that the visitor is

using. Each browser has its own user agent.

Each browser also allows you to test different user agents. If you're on Windows and using Chrome, you can switch the browser to "Safari on macOS." The web server will identify the user agent you're browsing with.

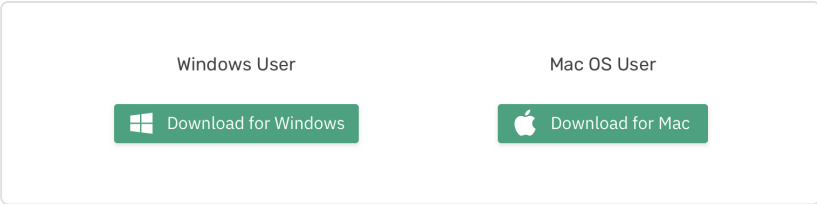
To debug and check the user agent of your current browser, open the DevTools' console and type the following:

```
console.log(navigator);
```

I'm using Chrome on macOS. The log shows this string:

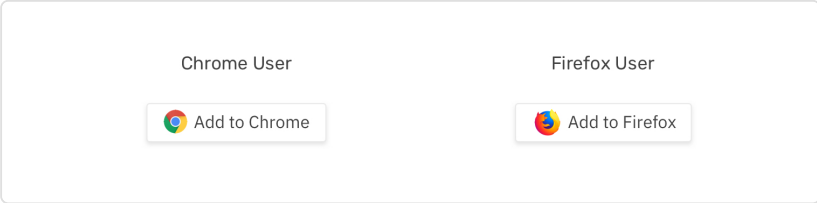
> Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163 Safari/537.36

Why debug this at all? Well, there are some important use cases. Consider this figure:




We have a download button for an application, which should change according to the user's operating system.

Another use case is a browser extension that is available in both Chrome and Firefox browsers:




The process of changing the user agent will depend on the browser you're using:

- Chrome: Network Conditions > Uncheck "Select Automatically" > select the user agent
- Safari: Develop > User Agent
- Firefox: I've found it's a bit complex, so I recommend using an extension instead.

## Debugging Media Queries

Media queries are the foundation of responsive web design. Without them, the web wouldn't look as it does today. To debug media queries, we need the power of the DevTools.

First, inspect the CSS you're debugging (in case you didn't write it). Does the code use `min-width` media queries more than `max-width`? Do you see any `max-width` media queries at all? This matters because of mobile-first design, which you've probably heard of. It entails writing CSS for small screens first, and then enhancing the experience for bigger screens such as tablets and desktop devices.


Here, we have a navigation toggle, shown by default on small screens. When the viewport is large enough to display the navigation items, the toggle disappears.

```
.nav__toggle {
    /* Shown by default */
}

.nav__menu {
    /* Hidden for mobile */
    visibility: hidden;
    opacity: 0;
}

@media (min-width: 900px) {
    .nav__toggle {
        display: none;
    }

    .nav__menu {
        visibility: visible;
        opacity: 1;
    }
}
```

However, if this wasn't built mobile-first, then the menu toggle would be hidden by default and shown via a `max-width` media query:

```
.nav__toggle {
    display: none;
}

@media (max-width: 900px) {
    .nav__toggle {
        display: block;
    }

    .nav__menu {
        visibility: hidden;
        opacity: 0;
    }
}
```

When debugging a project, you need to get your hands dirty with such details to know what you're dealing with. This will help you to fix issues more quickly and reduce unwanted side effects.

To view a media query in Chrome's DevTools, you need to select the element that is being affected by it:


Notice that when we select an element, we see the media query for it. The handy thing is that we can edit the media query and test right in the DevTools. The figure above shows a normal case, without any issue. Let's explore the most common bugs related to media queries.

## Don't Forget the Meta Viewport Tag

The `viewport` meta tag tells the browsers, "Hey, please take into consideration that this website should be responsive?" Add the following to the HTML's `head` element:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

## The Order of Media Queries Matters

Consistent ordering of media queries is important.

```
@media (min-width: 500px) {
  .nav__toggle {
    display: none;
  }
}

.nav__toggle {
  display: block;
}
```

Can you guess whether the `.nav__toggle` element will be visible in viewports wider than 500 pixels? The answer is yes, because the second declaration of `.nav__toggle` overrides the one in the media query.

The DevTools will show something similar to the figure above. The style in the media query would be struck through, meaning it's been canceled or overridden. The solution, then, is to order them correctly:

```
.nav__toggle {
  display: block;
}

@media (min-width: 500px) {
  .nav__toggle {
    display: none;
  }
}
```

## What If a Media Query Doesn't Work?

When someone is reporting a bug, saying that a media query is not working is not enough. However, we can check whether a media query is working with a simple test. Suppose we have this:

```
@media (min-width: 500px) {
    .element {
        background: #000;
    }
}
```

To test the media query, we can add a background color to the `body`:

```
@media (min-width: 500px) {
    body {
        background: red;
    }
}
```

Still not working? Then check whether:

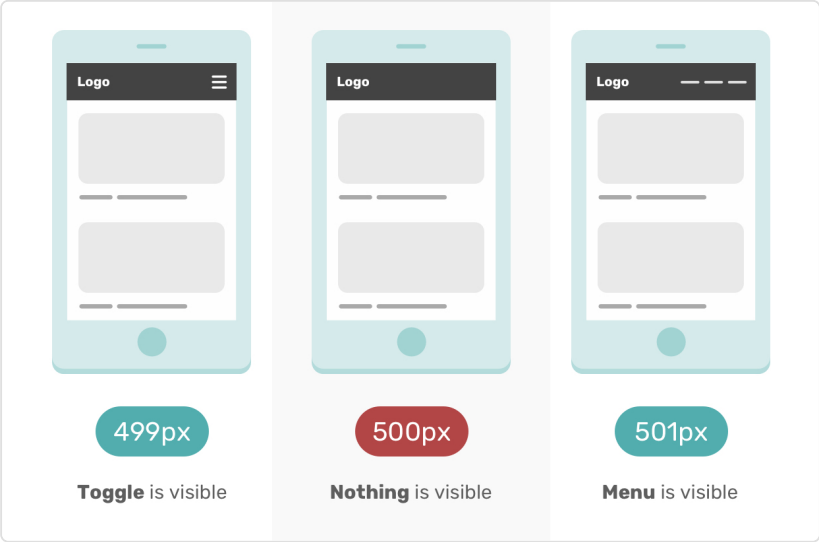
- the cached CSS is cleared,
- the CSS file is linked to correctly,
- the media query doesn't have a typo and is not missing the closing brace.

## Avoid Double-Breakpoint Media Queries

A common mistake is to use the same value in two media queries, one with a `min-width` and the other with a `max-width`. This typically happens with mobile navigation.

```
@media (max-width: 500px) {
  .nav {
    display: none;
  }
}

@media (min-width: 500px) {
  .nav__toggle {
    display: none;
  }
}
```

At a glance, these media queries might look good to you. However, 99% of the time, you'll forget to test an important breakpoint: `500px`, that 1-pixel gap between the two breakpoints. At this breakpoint, neither the navigation nor the toggle would be visible.

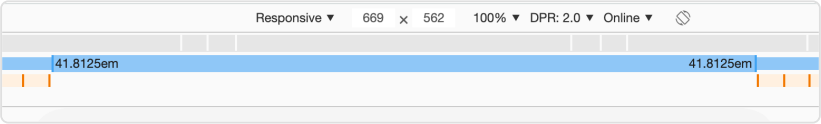



This 1 pixel is hard to debug without manually entering a value for a `500px` media query in mobile mode. To prevent this issue, avoid using the same value in two media queries.

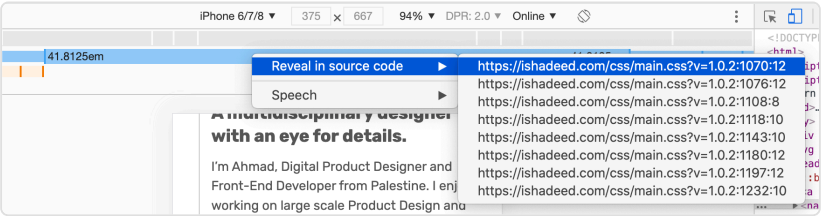
```
@media (max-width: 499px) {
  .nav {
    display: none;
  }
}

@media (min-width: 500px) {
  .nav__toggle {
    display: none;
  }
}
```

## List Media Queries

In Chrome, you can view a page according to the media queries defined in the CSS, rather than according to the list of devices available in the browser.



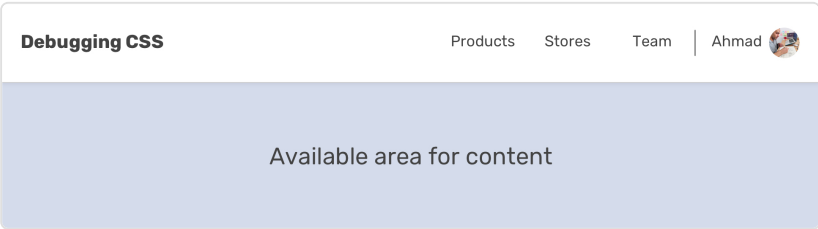

As you can see, we have two bars, the blue bar for `min-width` queries, and the orange for `max-width` queries. Having a broader view of all media queries is useful for testing multiple query sizes. Conveniently, we can reveal a media query in the source code. Right-click on a media query, select "Reveal in source code", and you'll be taken to the line of code for that media query.




## Vertical Media Queries Are Important

A common mistake with responsive web design is to test only by resizing the browser's width or by viewing multiple mobile sizes. However, decreasing and increasing the height of the viewport are equally important.

Just as we have media queries tailored to the width of the viewport, the same thing applies to height as well. Consider the following example:




After reducing the viewport's height, we find that the fixed header is taking up a lot of the screen's vertical space. Notice how small the area available for the content is. Users will be annoyed and won't be able to easily use the website. A simple solution would be to fix the header only when there is enough vertical space.

```
/* If the height is 800 pixel or more, the header should be fixed. */
@media (min-height: 800px) {
    .header {
        position: fixed;
        /* Other styles */
    }
}
```

### Don't Depend on Browser Resizing Alone

Resizing the browser to test responsiveness is not enough. For instance, Chrome's window narrows to a width of 500 pixels. This is not enough. You'll need to test below that (320 pixels, at least). Instead, test the website in the DevTools' device mode.

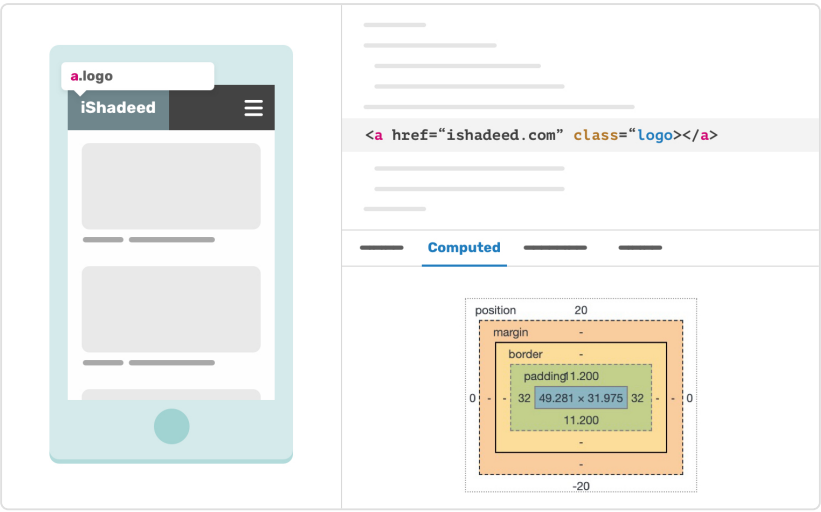
## Box Model

If we remember anything about the box model, it should be that every element

on a web page is a rectangular box containing one or more of the following: position, margin, border, padding, and content.

If padding and a border are applied to an element, they will be added to its content box, unless the `box-sizing` property for that element is set to `border-box`. Make this change to avoid any issues.

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

*, *::before, *::after {
  box-sizing: inherit;
}
```



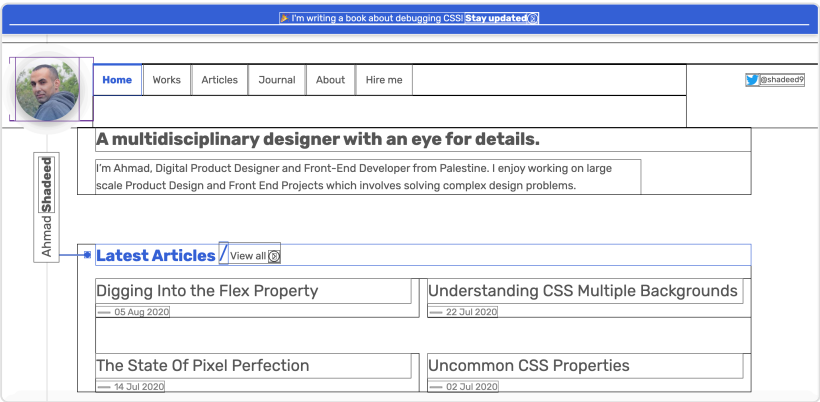

I inspected a website's logo to see its box model. Notice that you need to open the "Computed" tab to see the box model. All boxes are labeled, except the one

for width and height. When debugging an element, looking at the box model is extremely useful because it will show you all of the inner and outer spacings of an element.

Sometimes, we might build a web page without a CSS reset file, and we'll wonder why some elements have certain spacing. Looking at the box model is the best way to get an idea of what's happening.

## Everything in CSS Is a Box

Every single element on a web page is a square or a rectangle. Even if an element appears as a circle or has rounded edges, it is still a rectangular box. When debugging, keep that in mind.




The simplest way to see this for yourself is by going to your favorite website and applying the `outline` property to everything:

```
*, *:before, *:after {
    outline: solid 1px;
}
```

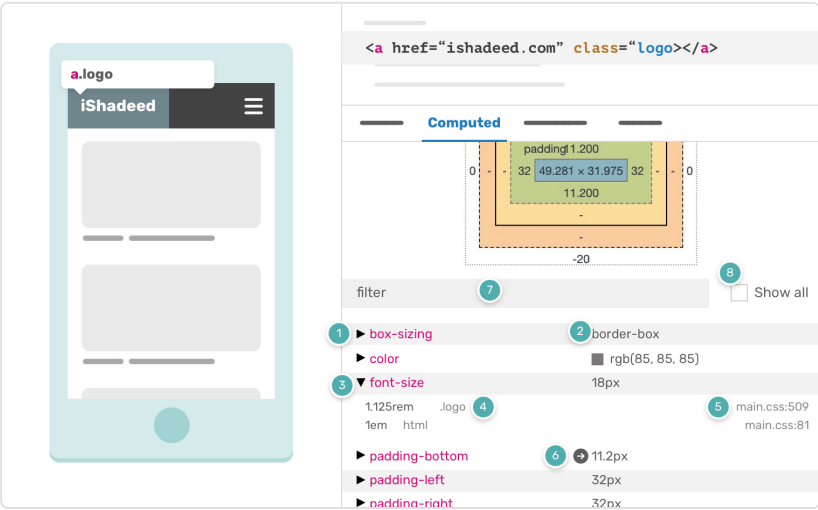
We've tagged every element on the page, including pseudo-elements ( `:before` and `:after` ). This way, we can see that the page is essentially some rectangles painted here and there.

## Computed CSS Values

In CSS, everything computes to a pixel value, regardless of whether you're using `rem`, `em`, `%`, or viewport units. Even the unit-less `line-height` property computes to a pixel value. In some cases, it's important to see the computed value of an element.

```
.element {
    width: 50%;
}
```

The width of `.element` is `50%`, but how do we see its computed value? Well, thanks to the DevTools, we can do that. Let's dig into the "Computed" tab.




You'll see that numbers have been assigned to various parts, to make it easier to explain this area of the DevTools.

1. This is the property's name. Usually, it's colored differently from the value.
2. This is the value of the CSS property.
3. We can expand a property to see its inherited styles. This element has `font-size: 1.125rem`, and it inherits a `1em` font size from the `html` element.
4. This is the pre-computed value, along with the element that the value is attached to.
5. This is the name of the CSS file, and the line number of the CSS rule.
6. When hovering over the value of a property, an arrow will appear. Clicking on it will take you to the source CSS file and the line number.
7. This filter helps with searching for a property. Note that you can only search by a property's longhand name. For example, searching for `grid-`

`gap` won't show anything, whereas searching for `grid-column-gap` would return a result.

8. By default, not all CSS properties are shown in the "Computed" tab. You will need to check this box to see them all.

## Grayed-Out Properties


You will notice that elements without an explicit height set might have a grayed-out `height` property.

```
.nav__item a {
    padding: 1rem 2rem;
}
```

The `<a>` element doesn't have a height set, but in reality, its height is the sum of the content and padding, which is an alternative to an explicit height.

This doesn't happen only for padding. The example below has two elements, one of which is empty.

```
<div class="wrapper">
    <div class="element">content</div>
    <div class="element"></div>
</div>
```

```
.wrapper {
    display: flex;
}

.element {
  width: 100px;
  padding: 1rem;
}
```

Flexbox stretches child items by default. As a result, the height of the items will be equal, even the empty one. If you inspect that element and check the computed value of its `height` property, you will notice that it's grayed out.

Tip: A property that is grayed out means that its value hasn't been set explicitly. Rather, it's being affected by other things, such as the element's content or padding or by being a flexbox child.

## Firefox's Style Editor


The style editor in Mozilla's Firefox browser is a kind of a design app in the browser. Here are some of the great things you can do with it:

1. Create new style sheets and append them to the document.
2. Import a CSS file.
3. List all of the CSS files for a document, with the ability to activate and deactivate them by clicking an eye icon (similar to showing and hiding layers in a design app).
4. Save a file from the list.
5. List all media queries in the selected CSS file. The active one will be highlighted in a dark color, and the inactive ones will be dimmed. You can jump to the part of the code has the media query.
6. Click a media query.

What I particularly like about this is that you can hide all of the CSS files, which is the equivalent of disabling CSS.

Also, being able to import a CSS file into a page is useful, opening up a lot of possibilities. Imagine that you're working on a layout for a web page and want to change a few things here and there but without losing your work. You can

import a new CSS file, copy the current CSS to it, and then edit it as much as you want. When you're done, you can switch between the old and new CSS to see the completely different layouts.

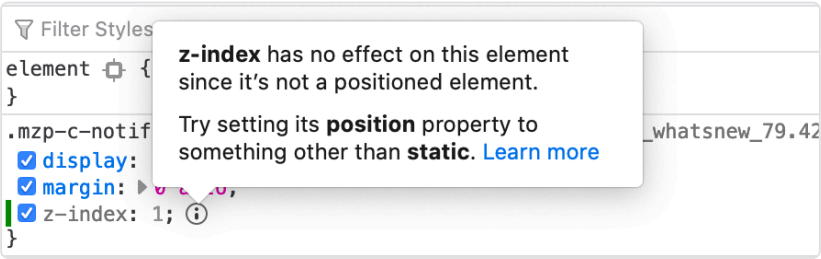
## CSS Properties That Don't Have an Effect

Some CSS properties have dependencies. Take the following:

```
.element {
    z-index: 1;
}
```

If you haven't changed the `position` of the element to anything other than `static`, then it won't affect the element at all. It's not easy to spot these issues while debugging because they don't break the layout. They are silent.

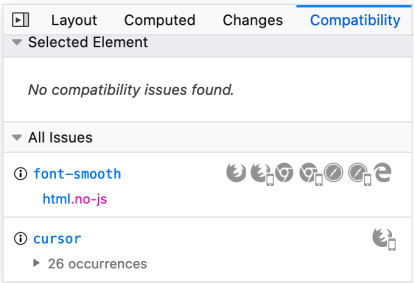
Firefox has a very useful feature for this, showing a warning when a CSS property doesn't affect the element it's being applied to.




This is very helpful and, at the time of writing, available only in Firefox.

## Compatibility Support in Firefox

While inspecting an element's CSS, you can see the browsers that support a particular feature, along with the browser versions. You can view details by hovering over one of them. I like this feature a lot because it gives you hints on which browsers to test more carefully.



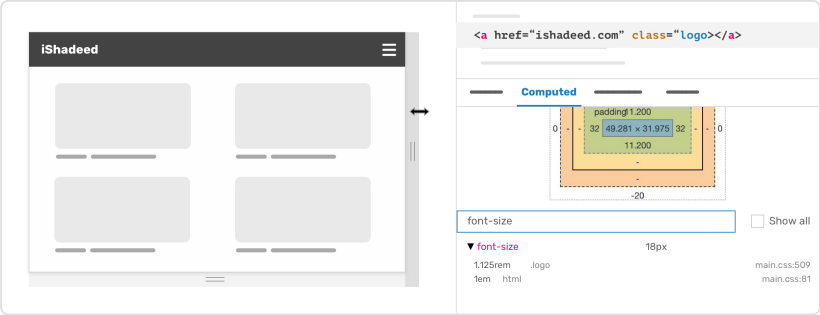

## Getting the Computed Value While Resizing the Browser

It's not enough to look at the computed value of an element. What's more useful is to filter a specific property that you need to check, and then resize the responsive view wrapper to see the value change.

```
.title {
    font-size: calc(14px + 2vw);
}
```

Here, we have a title with a base `14px` font size plus `2vw` (2% of the

viewport's width). Here is an explainer:




I searched for `font-size` and then started to resize the view on the left. This is a very helpful way to keep yourself aligned with what's happening in the background.

## Getting the Computed Value With JavaScript

By using JavaScript's `getComputedStyle` method, it's possible to get the value of a specific property. Consider the following example:

```
.element {
    font-size: 1.5rem;
}
```

We've set the font size using the `rem` unit. To get the computed pixel value, we would do this.

```
let elem = document.querySelector('.element'); /* [1] */
const style = getComputedStyle(elem); /* [2] */
const fontSize = style.fontSize; /* [3] */
```

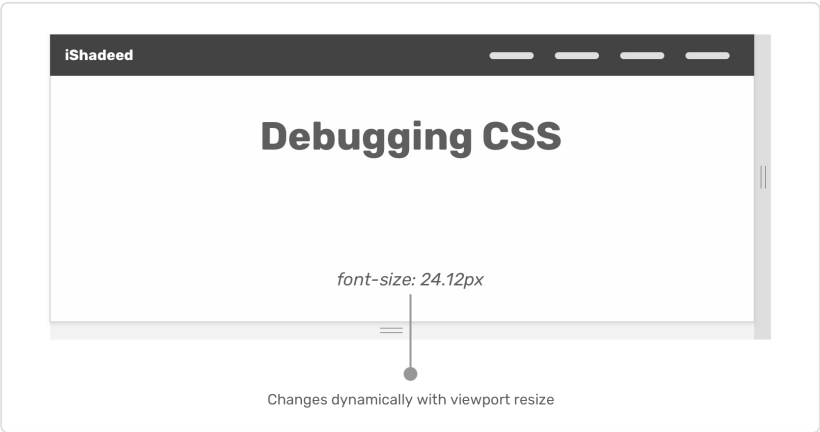
Here is what the code is doing:

1. It selects the element.
2. It stores the element's style in the `style` variable.
3. Now that the `style` variable is an object, holding all of the element's style, we can get the computed value of the property.

Cool! What if the property we want to check has a viewport- or percentage-based value (for example, `font-size: calc(14px + 2vw)`)? The value of that font size would change with every viewport resize.

```
let elem = document.querySelector('h1');

window.addEventListener('resize', checkOnResize);

function checkOnResize() {
  const style = getComputedStyle(elem);
  console.log(style.fontSize);
}

checkOnResize();
```

As you can see, this is the same concept, but with a `resize` event used this time. This can be very useful for tracking things, and you can even render the value on the page itself:




## Reordering HTML Elements

In Chrome's DevTools, you can click and drag an element to reorder it. This can be useful for changing the structure of an entire page or component. Once it's reordered, we can start testing various things, such as:

- the flexbox `order` property,
- the adjacent-sibling combinator ( `.item + .item` ),
- an element with `margin-bottom` or `margin-top`.

Let's dig in more and learn how reordering works.

1. Open up the DevTools.
2. Select the element you want to reorder.
3. Click and drag the element wherever you want.

This is how you would drag a `section` element along with its sibling:


We can also reorder child items. Suppose each section has a title and description. We can reorder them inside their parent.


A child element can be dragged in multiple ways:

- inside its parent (this will just reorder it at the same level),
- between other parent elements,
- inside another parent element.


## Editing Elements in the DevTools

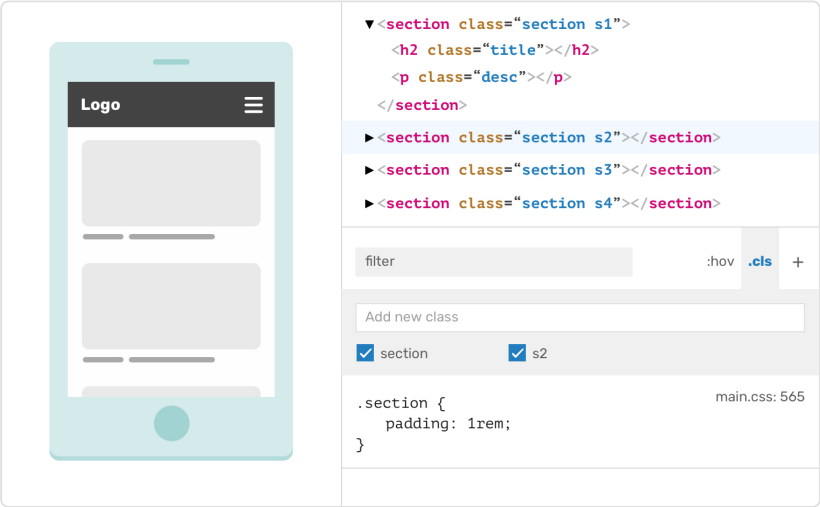
There are multiple ways to edit an HTML element in the DevTools. They're very useful in cases where you want to add a class or attribute or maybe delete the whole element. Here are the ways to do it:

- add or delete a CSS class,
- change the element type (for example, `<div>` to `<h2>`),
- add or remove an attribute,
- delete the element.

### CSS Classes

To add, edit, or remove a CSS class, you could double-click the class name of

the element, and it will become editable. But this is the less recommended way to do it. The better way is to select the element, and then click the `.cls` label with the DevTools opened. Being clicked, it will show all of the classes associated with the selected element, and we can add or remove them by checking and unchecking the boxes.




## Utility-Based CSS Websites

If the website you're working on was built with utility-based CSS, debugging its CSS in the browser would be different than debugging a website with regular class names.

Here is an element with a class name:

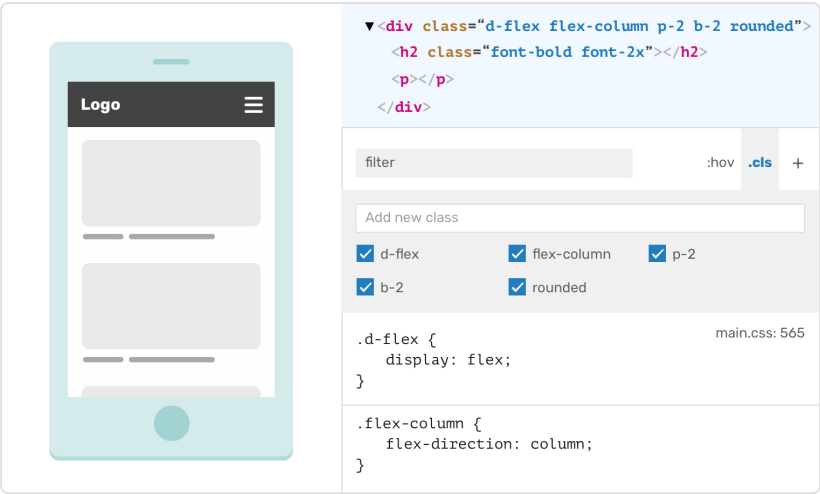
```
<div class="card"></div>
```

And here is the same element with utility-based CSS:

```
<div class="d-flex flex-column p-2 b-2 rounded hidden"></div>
```

When the whole website is built with utility classes, debugging will be a little different. Let's say we want to inspect an element and remove `display: flex` by unchecking the box in the DevTools. If we do this, any element that uses the `d-flex` class will break. The reason, of course, is that we've removed the `display` property from all of those other elements.

Using the `.cls` option would be better because it will list all of the CSS classes for that element:



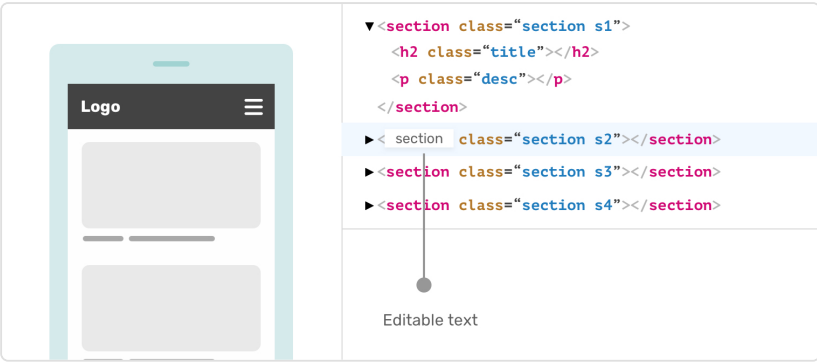

Another option would be to add inline CSS styles, which would override the

ones added in the CSS files. Or you could double-click on the element's class attribute and manually remove the class that you don't want.

## Changing an Element's Type

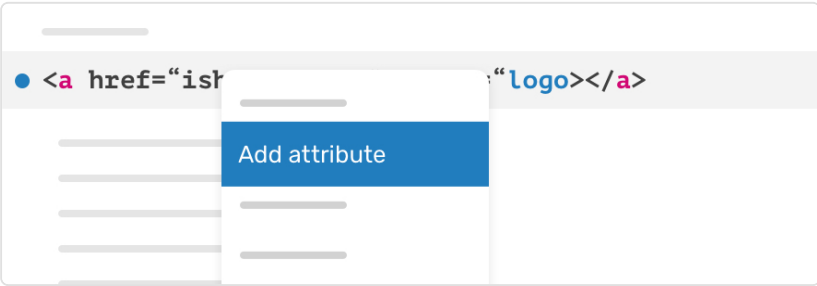
Say you have a `div` element but want to change its type without leaving the DevTools. Well, this is possible. To change it, double-click the element type and then edit the opening tag.

Note: There is no needed to edit the closing tag. The DevTools will do that automatically.



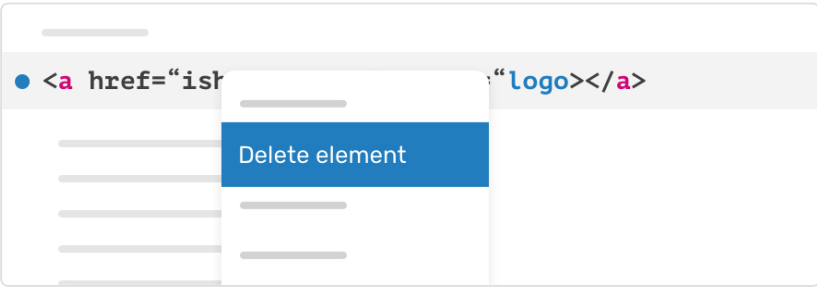

## Adding or Removing an Attribute

When you need to add an attribute, select the element, right-click, and select "Add Attribute." It's that easy. Note that you can also add it by double-clicking on the element itself.




## Deleting an Element

To delete an element, hit the `Function + Backspace` keys. This will work in all browsers and on all platforms. If you are using Chrome, hit the `Backspace` key only. The mouse is an alternative: Right-click the selected element, and choose "Delete" from the list.




## Keyboard Goodness

Some keyboard shortcuts are very useful and increase productivity:

- Navigate between elements with the up and down arrow keys.
- Hit the right arrow key to expand an element and the left arrow key to collapse it.
- When an element is selected, hit `Enter` to edit the CSS class name.

## The `H` Key

The fastest way to hide an element in the DevTools is by hitting the `H` key, which will add `visibility: hidden` to the element. The space taken up by the element will be preserved.

What's the benefit of hiding an element in this way? Here are a couple of uses:

- If you have a child of an element that doesn't appear as expected, we can use `H` to investigate it.
- If you need to screenshot an element or section, and you don't want all of its details to be in the image, simply use `H` to hide the unwanted elements.

## Forcing an Element's State

In CSS, an element can take one of four states (pseudo-classes): `:visited`, `:focus`, `:hover`, `:active`. Thankfully, we can debug all of them using the DevTools.

Before digging into how to debug them, let's review the pseudo-classes.

- `:visited` is the state when a link is clicked. When a user revisits a web page, that link should have a different state, so that the user can tell

they've visited it.

- `:focus` is the state that shows up when the user navigates the page by keyboard. A button or link should take a style that clearly indicates it is in focus.
- `:hover` is the state when an element is hovered over by the mouse.
- `:active` is the state when an element is being pressed from a click by the mouse.

In CSS, the order of pseudo-classes matters. It should be as follows:

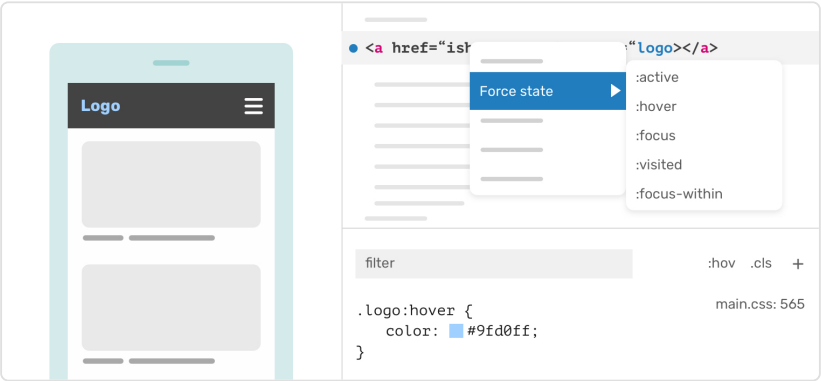
```
a:visited {
    color: pink;
}

a:focus {
    outline: solid 1px dotted;
}

a:hover {
    background: grey;
}

a:active {
    background: darkgrey;
}
```

If this order is not followed, some styles will get overridden. Order the styles correctly to avoid issues.

Let's get to the interesting part. How do we debug these pseudo-classes in the DevTools? Well, there are two ways.

## Select an Element

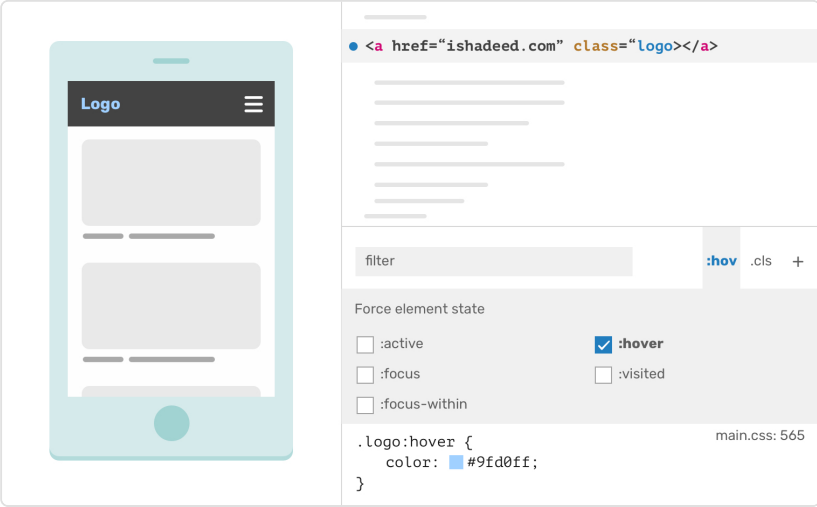
Right-click an element or click on the dots icon on the left, and then choose "Force State." From the options list, choose the state you want to activate. See the figure below:




Checking the box adds a blue dot to the element on the left side. This visual indicator shows that the element has a forced state.

## Use the Panel

Another way to force an element's state is by using the DevTools panel. Clicking on `:hov` will expand a list with all pseudo-classes. Each one has a checkbox, which makes it easy to activate or deactivate a state while testing.




## Toggle the State of an Element

We can also add a pseudo-class manually:

1. Select an element.
2. Click on the `+` button in the panel.
3. A new rule will be added in the styles panel. All you need to do now is edit it and add the pseudo-class you want.

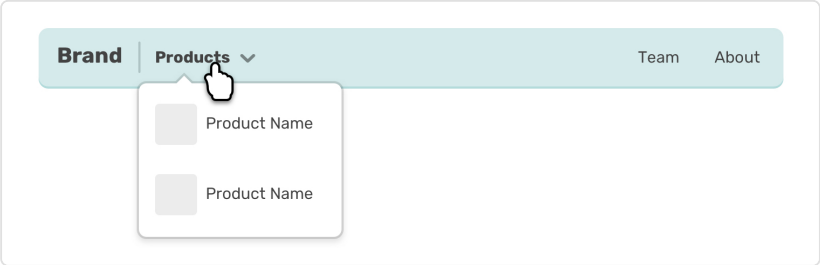

## Debug an Element Shown Via JavaScript

In some cases, hovering over an element will add a child to the DOM. Debugging these elements is tricky. They will be hidden in the inspector because you are not actively hovering over them.

The question is how to debug a hidden element? Well, there are a couple of ways.

### Is the Element in the HTML?

In this case, the element we want to debug is already in the HTML but hidden via CSS and only shown once its parent is hovered over. To debug this, the first thing we need to do is inspect the parent element. What's the parent, you ask? This should clarify:




In this example, we have a dropdown menu that is toggled on hover via JavaScript. To debug the dropdown itself, we would inspect the "Products" menu item, and the dropdown element should be inside it. From there, we can add `display: block` to the element and start the debugging process.

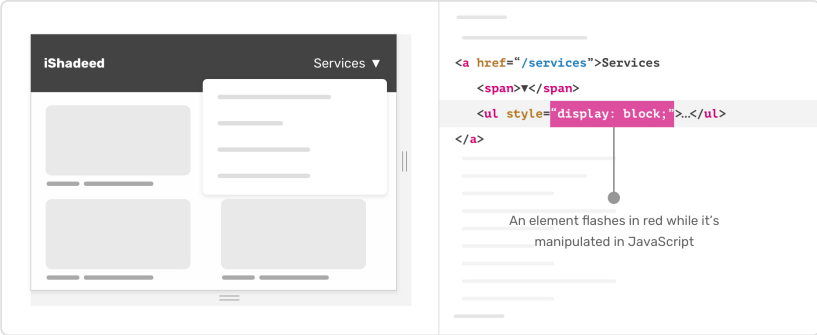
## Is the Element Added to the HTML on Hover?

This is more challenging. In this case, an element is added to the HTML only when its parent is hovered over, and it's removed completely from the HTML when the user stops hovering. We'll need help from the DevTools for this. To debug, we need to freeze the website once the thing we want to debug has become visible. The best way to do that is to pause the JavaScript's execution.

When JavaScript execution is paused, it's possible to follow the code that toggles the menu. This way, we can catch the element once it appears, and inspect it from there.

One important clue that indicates an element is being added to the DOM on hover is that its parent element flashes red. The flashing means that this DOM element is being edited through the addition or removal of a child item or maybe the modification of attributes.

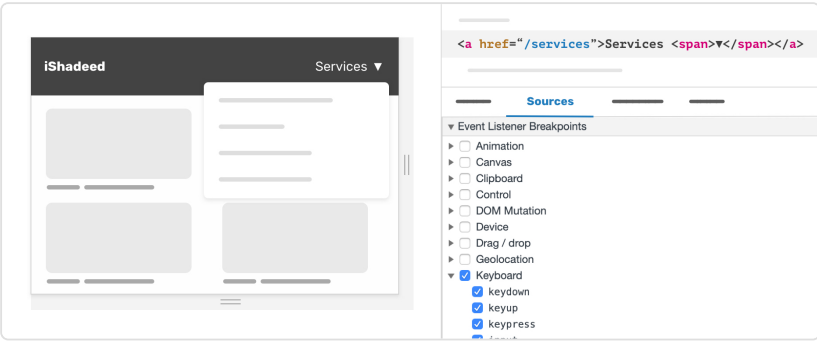
## 3. Debugging Environments and Tools




How do we pause JavaScript execution? Easy:

1. Go to the "Sources" panel.
2. Expand "Event Listener Breakpoints."
3. Add an event listener to "Keyboard."

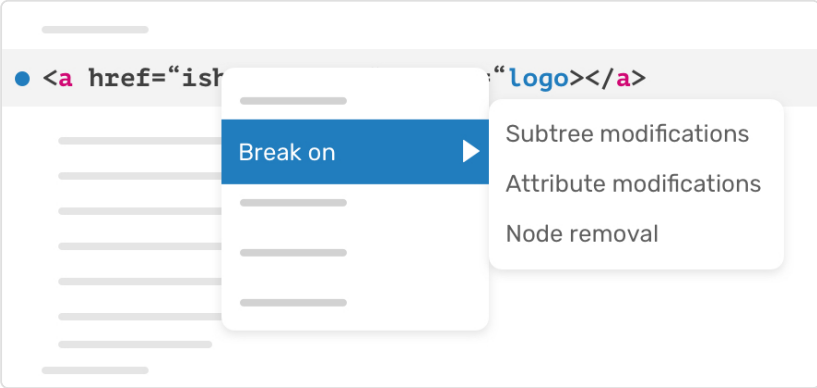
Now, hover over the element that you want to debug. Once you do, press any key on the keyboard, and you will notice that the application freezes, and the thing you want to inspect won't disappear. Feel free to dig in and inspect!



## Break JavaScript

In Chrome and Firefox's DevTools, you can break the execution of JavaScript with any of the following:

- subtree modification,
- attribute modification,
- node removal.




Let's get into each one.

### Subtree Modification

This targets child items of the selected parent. If any addition or deletion of an HTML element happens, this is considered a modification. In this case, the browser will add a breakpoint.

### Attribute Modification

This watches for any modifications to the attributes of the selected element, such as class names and HTML attributes. For example, a change to a class or style attribute would cause the browser to add a breakpoint, and a menu would then be shown via JavaScript.

### Node Removal

This is fairly obvious. Once an element is removed from the HTML, the JavaScript execution would be paused.

## Using the Debugger Keyword

Sometimes, a CSS bug will appear while a certain JavaScript function is running. For example, you might need to debug the mobile menu once it's toggled. In this case, the `debugger` keyword can be useful.

In the JavaScript for the toggle button, you would add the following:

```
function showNav() {
    debugger;
    // Add a breakpoint once this function is called.
}
```

Once this function is called, the browser will add a breakpoint. Then, you can start debugging.

Note that if the browser doesn't support the `debugger` keyword, this won't have an effect.

## Formatting the Source Code to Be Easier to Read

When you inspect an element and want to check its CSS file from the DevTools, you might find that the file has been minified. This makes it hard to read. Thankfully, we have the little "Format Code" feature, which quickly formats the minified code.

```
<a href="/services">Services <span>▼</span></a>
```

Sources

main.css* ×

```
509         left: 50%;
510         -webkit-transform: translateX(-50%);
511         transform: translateX(-50%);
512         width: 8px;
513         height: 7px;
514         opacity: 0;
515         border-radius: 50%;
516         background: #488D52;
517         -webkit-animation: base 0.25s 0.1s ease-out forwards;
518         animation: base 0.25s 0.1s ease-out forwards;
519     }
520
521     .c-nav .c-nav__wrapper .c-nav__item {
522         display: block;
523         opacity: 0;
```

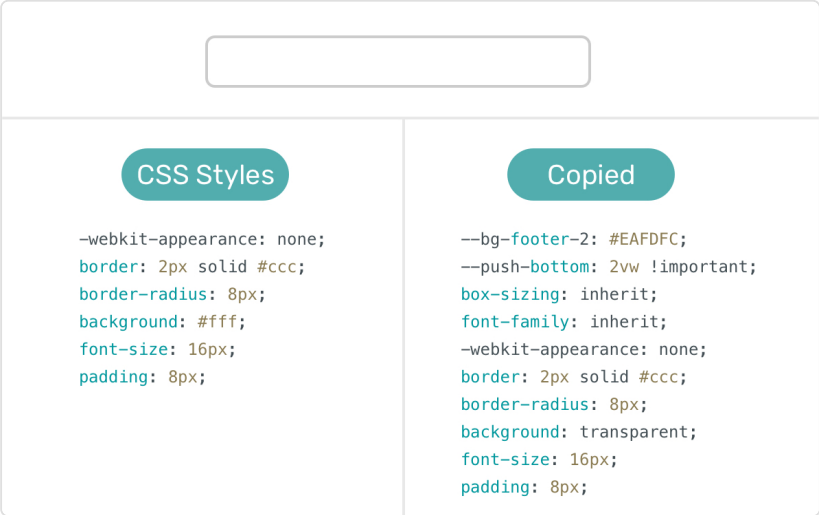
{} Line 523, Column 5 Cover

Notice the opening and closing braces icon. One click on it and all of the code will be formatted and easy to read.

## Copying an Element's HTML Along With Its CSS

The only browser that allows you to copy the CSS styles of an element is Chrome, even though it isn't perfect. The latest version at the time of writing, Chrome 81, has that feature. Here is how to do it:

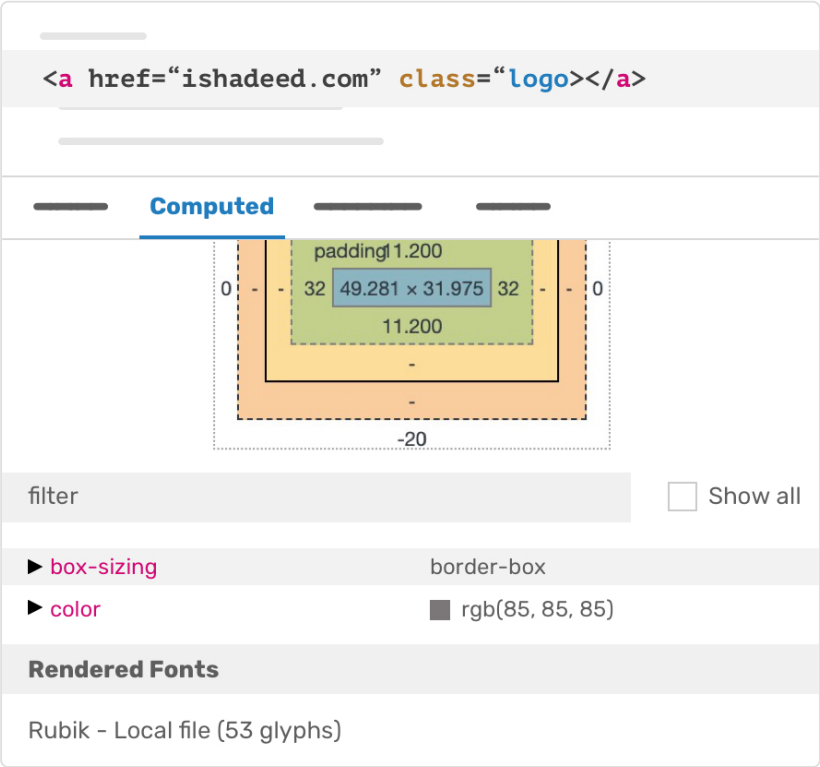
1. Right-click on a element.
2. Select "Copy."
3. Then, select "Copy styles." That's it!




In the figure above, notice the difference between the original CSS and the one copied from the browser's DevTools. The copied one has inherited styles such as `box-sizing` and `font-family`. Also, weirdly, it copied all of the CSS properties in the document!

## Rendered Fonts

Rendered fonts are the ones currently being used for a web page. To debug them, inspect any text element, such as a heading or paragraph. At the bottom of the "Computed" tab, there will be a section named "Rendered Fonts." There, you can check the font applied to the element.




Also, as mentioned in the "Computed" section, you can search for `font-`

`family` and see the computed value of it. In addition, you can expand the arrow next to it and see the CSS responsible for the addition of the font.

## Checking for Unused CSS

One useful feature in Chrome's DevTools enables you to check for unused CSS. It's called "Coverage." Here is how to use it:

1. Open up the DevTools.
2. Click on the dots icon, and select "More."
3. Open the "Coverage" panel and hit the reload icon.

Once it's reloaded, you will see something like the following:

```
1951
1952 .footer-row li:not(:last-child):after {
1953     display: -ms-inline-flexbox;
1954     display: -webkit-inline-box;
1955     display: -webkit-inline-flex;
1956     display: inline-flex;
1957     content: ".";
1958     margin-left: 4px;
1959     margin-right: 4px;
1960 }
1961
1962 .footer-row a {
1963     color: var(--color-brand-primary);
1964 }
1965
1966 .footer-row a.twitter {
1967     color: #1DA1F2;
1968 }
1969
1970 .footer-row a.behance {
1971     color: #1769FF;
```

{} Line 1706, Column 15 Coverage: 27.6 %

Console Network conditions Coverage × Animations Sensors

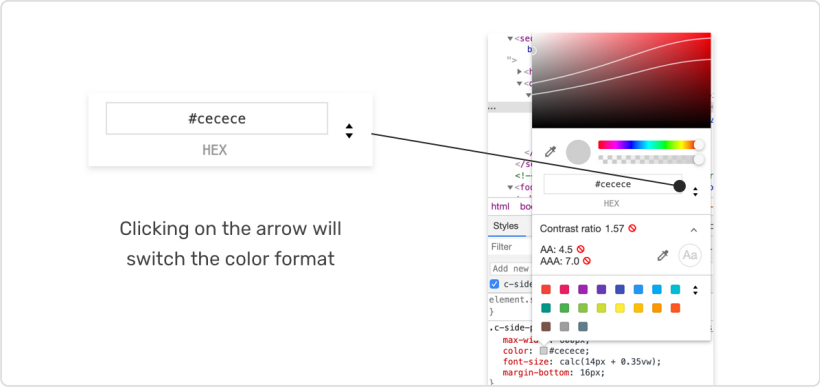
Per function URL filter All Content scripts

| URL | Type | Total B… | Unused Bytes | Usage Visualization |
|---|---|---|---|---|
| https://ishadeed.c… /main.css | CSS | 72 092 | 52 168 72.4 % | |
| https://www.goo…/analytics.js | JS (… | 45 229 | 21 595 47.7 % | |

The code blocks highlighted in red are the ones that are not used on the page, while the ones in blue are being used. Also, it shows you the percentage of the used CSS. That feature is extremely useful for refactoring CSS and checking whether you have unused styles.

## Color-Switching With the DevTools

Chrome's DevTools provide three types of color systems: hex, RGBa, HSLa. When you pick a color for an element, it's usually added as a hex color. What if you want the RGBa value of that color without having to use a converter tool? Well, that feature is available in the color inspector.




If you want a particular blue, and the design requires you to use it with `50%` opacity, add the color as a hex value to the element and, with the color inspector still open, click on the double-arrow icon on the right. This will switch between hex, RGBa, and HSLa, very handy for quickly coverting a color from one type to another.

## Copying CSS From the DevTools to the Source Code

When you edit the CSS of an element, you'll probably want to copy and paste it back in your code editor, instead of having to write it again. There is more than one way to do this.

### Copy Directly From the Inline Inspector

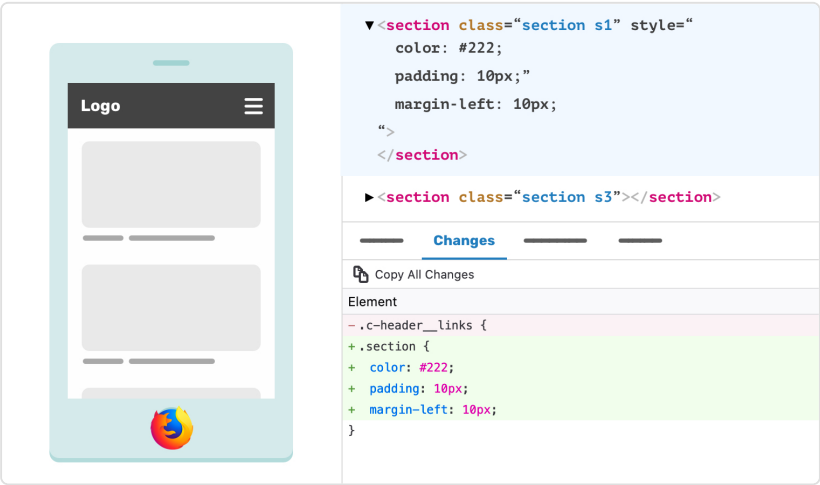
In the following figure, I've added some inline styles to an element. Notice how they're added to the `element.style` selector in the DevTools. This feature is the same for Chrome, Firefox, and Safari.


Now that we've added these inline styles, it's possible to copy and paste them into our code editor.

## Use the `changes` Feature in Firefox Browser

Firefox has a useful feature named "Changes" that shows the changes we've made in the DevTools. It's not unlike how version control shows the difference between two changes. Here is how to use it:

1. Inspect the element you want to edit.
2. Edit the styles.
3. Go to the "Changes" tab, and you will see the edits you've made.




## Debugging Source-Map Files

When using a preprocessor such as Sass, the rendered CSS file might contain instructions for a linked source-map file. When you inspect an element's CSS in the DevTools, you will notice that the CSS is in a file with the extension `.scss`. This can be confusing.

To remove that `.scss` file from the DevTools, you will have to turn off the source-map feature; or you can open the "Sources" panel in the browser and select the CSS file. Then, you will find something like this:

```
/*# sourceMappingURL=index.css.map */
```

This is an instruction that makes the browser load the Sass file. Removing it will hide the source-map file completely.

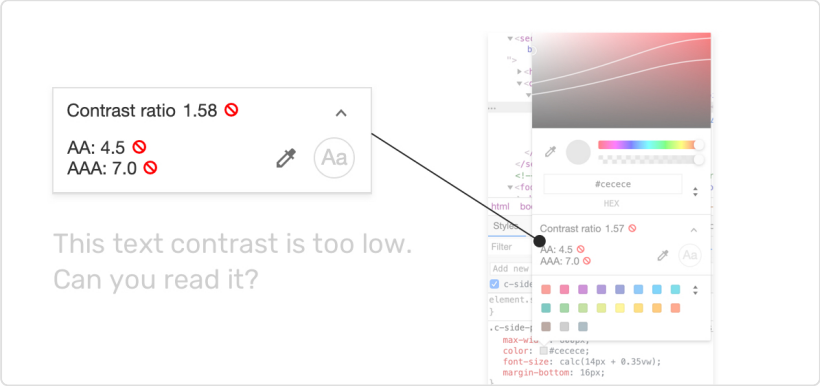
## Debugging Accessibility Issues Caused by CSS

Even though most accessibility issues are caused by misused HTML, CSS plays a role in accessibility, too. In this section, you will learn some things to keep in mind when debugging CSS.

### Give the Text Sufficient Color Contrast

A color that is too faint to read will be a problem for users. According to the Web Content Accessibility Guidelines (WCAG) 2.0, the foreground and background colors should have a 4.5:1 contrast ratio at Level AA and a 7:1 contrast ratio at Level AAA.

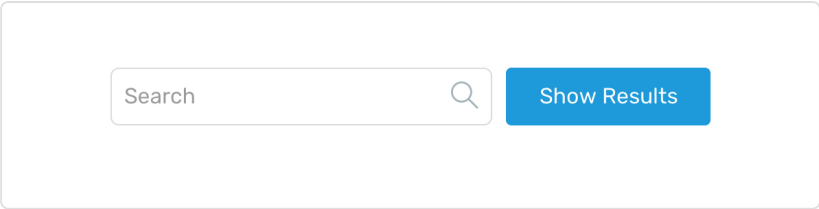
To achieve this, use colors that are well tested. Great tools are out there to make our job easier. To check whether a text color is accessible, inspect the element, and click on the little color square. You will see a contrast number.



## Think Twice Before Hiding With `display: none`

Using `display: none` incorrectly is a hindrance to accessibility. It can easily upset an experience. Suppose you have a search input and button, and the design calls for the `label` element to be hidden.

```
<label class="visually-hidden" for="search">Search</label>
<input type="search" id="search"/>
<button>Show results</button>
```




The `label` should be hidden visually but not with `display: none`. Why?

1. You wouldn't be able to tie the `label` to the input using the `for` attribute.
2. A screen reader wouldn't be able to read the input's label. If you're lucky, it might read the placeholder, if one has been added.

The correct way is to add a class of `visually-hidden` to the label. This will only hide it visually and, as a result, won't be an accessibility issue.

This snippet comes from The Accessibility Project:

```
.visually-hidden {
    position: absolute !important;
    height: 1px;
    width: 1px;
    overflow: hidden;
    clip: rect(1px 1px 1px 1px); /* IE6, IE7 */
    clip: rect(1px, 1px, 1px, 1px);
    white-space: nowrap; /* added line */
}
```

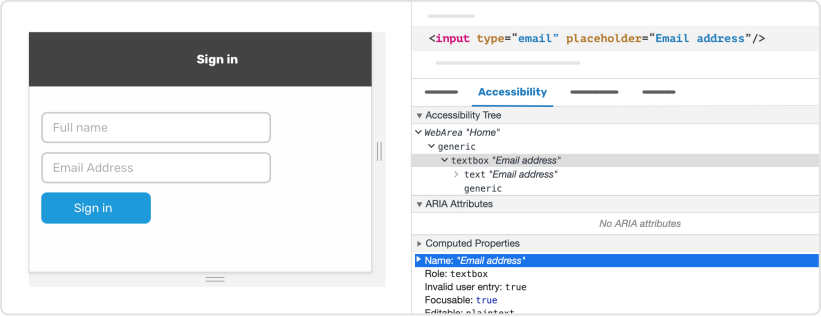
## Use the Accessibility Tree

The accessibility panel in the DevTools is a beautiful one. It give us clues on how an element will be exposed to screen readers. For instance, if we select an `input` field and check the accessibility tree, it will show us the label (if available) and the placeholder.

Fixing small issues related to this can have a huge impact, and you don't need to be an accessibility expert to do it. Here is a realistic example:

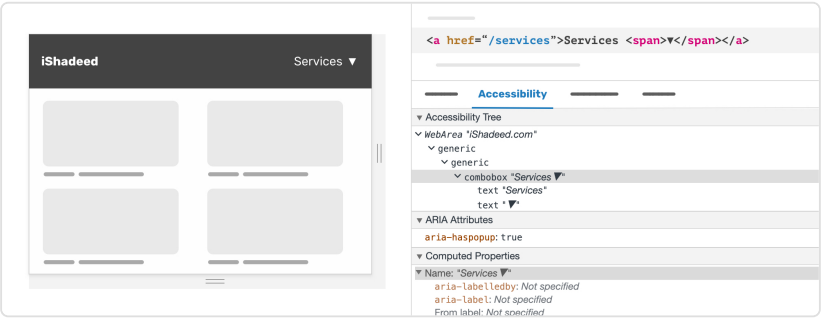
```
<label class="visually-hidden" for="email">Email address</label>
<input type="search" placeholder="Email address" id="email"/>
```

We have an email `input`, without a `label` associated with it. If we inspect the `input` and go to the accessibility panel, we will see something like this:




Notice that it says "textbox: Email address", and it reads what's inside the `input` s placeholder. Without it, the field would be empty, and that would be a problem. Make sure to debug using the accessibility tree when you're working with web forms.

Of course, it's not only about forms. There are elements that shouldn't be exposed to users of screen readers — for example, a menu item with an accompanying arrow.




The arrow is an HTML symbol inside a span. When inspected, it shows the text as "Services ▼", which is not correct. A screen reader would read this as: "Services down pointing black pointer." Very confusing. Debugging such

issues as early as possible is highly recommended. The solution is to use `aria-hidden=true` for the `span` element.

### Fix Unclickable Elements

Interaction with buttons and links is vital. When an element is expected to be clickable but is not, that is a problem. Misuse of a CSS property can prevent an element from being interactive. Consider this example:

```
.element {
    pointer-events: none;
}
```

CSS `pointer-events` prevent, for example, an event on a button from happening. In this case, when the user hovers over it:

- the cursor won't change,
- clicking on it does nothing.

A simple CSS property can prevent a button from being clickable. Misusing properties can ruin the experience, resulting in the loss of users.

## Debugging CSS Performance

Some CSS properties can cause performance issues when used incorrectly for animation. The properties that any browser can animate cheaply are:

- transforms ( `translate` , `scale` , `rotate` );
- opacity.

Using other properties for animation is risky and could impair performance. Let's go over how the browser calculates its styles. Here are the steps that the browser takes:

1. **Recalculate styles**: Calculate the styles that are being applied to each element.
2. **Layout**: Assign the width, height, and position of each element.
3. **Paint**: Paint all elements to layers.
4. **Composite layers**: Draw the layers to the screen.

The least-heavy step is composition. To achieve good performance, use only the `transform` and `opacity` properties. The figure below compares `left` and `transform: translateX` for animation.


Notice how busy the `left` timeline is. The browser keeps recalculating the styles while the animation is happening. Meanwhile, `translateX` is very different; the browser's work is light.

To check the performance of your web page, open up the DevTools, and select the "Performance" tab. From there, you can start profiling and doing a test. A profile is like a test that runs on the page for some time (usually seconds). When it's done, you can see a timeline with all of the details on how the browser calculated the styles.

Our concern with the CSS is the recalculating and compositing. Avoid using heavy CSS properties for animation.

## Multiple Browser Profiles

You likely use different browsers, each of which stores the history and private information of your browsing. Debugging and testing websites in a browser you use every day might not make sense. You'll need something fresh, without a history or cache, so that you can test without any unwanted issues, like CSS caching, and to avoid extensions that might cause bugs.

For this reason, a dedicated browser profile for testing is recommended. Here is how to create one in Chrome:

1. In the top-right corner, click on the user avatar.
2. Click "Add +."
3. Name the profile (for example, "Dev"), and click "Add."

In Firefox, it's a bit different:

1. Open `about:profiles` in the browser's URL field.
2. Click on "Create a new profile."
3. Choose a name and click "Done."
4. On the same page, scroll down to find the profile that you created, and click on "Launch profile in a new browser."

Done! You've created a profile especially for testing and debugging.

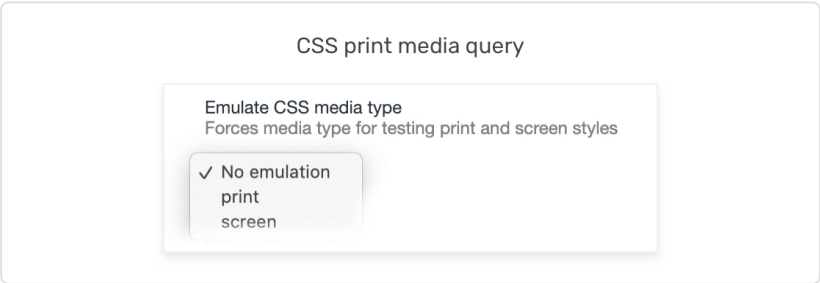
## Rendering and Emulation

In Chrome, we can emulate different rendering and emulation media queries, to help us debug for the CSS query `@media print`. We can also debug the light- and dark-mode versions of a website with the `prefer-color-scheme` media query.

To access the rendering and emulation settings, follow these steps:

1. Open the DevTools, and click the vertical dots menu.
2. Choose "More tools" and then "Rendering."
3. Scroll down and you will find the emulation options.

### CSS Print Styles

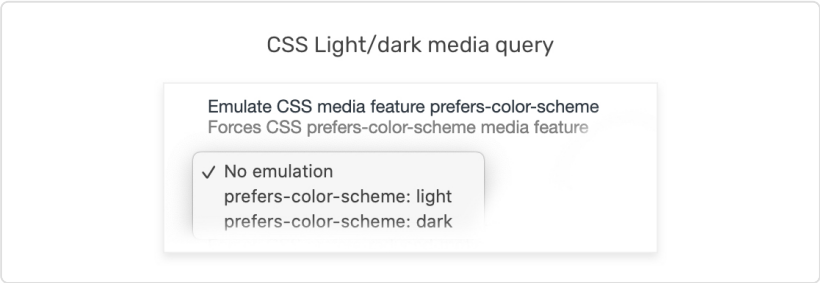



We can use a media query to edit the CSS styles and tailor the page to be printed. To debug and emulate how a web page will look when it's printed, we can either print the page and save it as a PDF file or use the emulation feature in Chrome.

```
@media print {
 /* All of your print styles go here. */

 .header, .footer {
     display: none;
 }
}
```

The header and footer of a website might not need to be printed, so they can be hidden with the `print` media query.

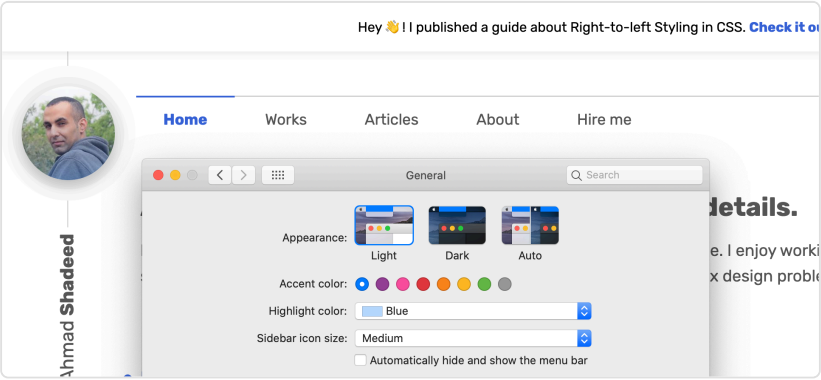
## CSS Media `prefer-color-scheme`




With iOS officially supporting them, dark-mode user interfaces are rising in popularity and becoming supported on the web as well. In CSS, we can use the following to detect whether the user prefers dark or light mode:

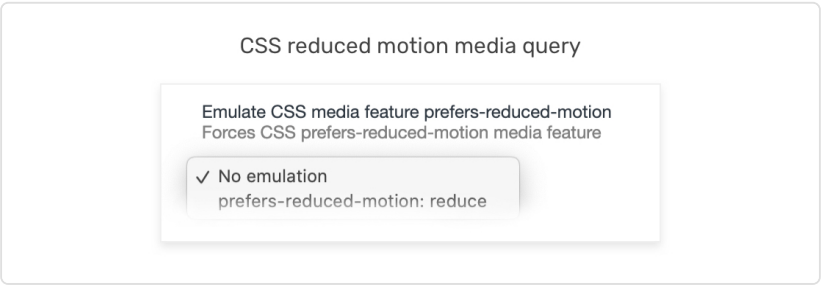
```
.element {
    /* Light-mode styles (the default) */
}

@media (prefer-color-scheme: dark) {
    /* Dark-mode styles */
}
```

On macOS, you can switch between the dark and light mode of a website by changing the system preferences.




While this works, it might not be practical when you're working on a lot of changes. Thankfully, it's possible to test that in the rendering settings. We can emulate the media query `prefer-color-scheme: dark` or `prefer-color-scheme: light`.

## CSS Media `prefers-reduced-motion`




You can't assume that all users will be fine with animation playing here and

there on your website. Some people prefer not to see animation on a page because it can affect accessibility. A media query checks whether the user has requested that the system minimize non-essential motion.

```
.element {
    animation: pulse 1s linear infinite both;
}

@media (prefers-reduced-motion) {
  .element {
    animation: none;
  }
}
```

Better yet, you can have simpler animation for users who prefer reduced motion.

```
@media (prefers-reduced-motion) {
  .element {
    animation-name: simple;
  }
}
```

With that, we're done going over the browser's DevTools. Let's go over the other methods we can use to test and debug CSS.

## Virtual Machines

In the course of your work as a web developer, you will need to test in browsers and on operating systems other than the ones you normally use. For instance, you might use macOS but want to test on Chrome for Windows. In this case, the cheapest solution is to use a virtual machine. I recommend

using VirtualBox because it's free, easy to use, and works on both macOS and Windows.

Also, Microsoft offers free copies of Windows to test in Edge and Internet Explorer 11 browsers.

## Online Services

Similar to a virtual machine, some online services enable you to test on hundreds of types of devices. However, they're not free, and they depend on a fast internet connection.

## Mobile Devices

Testing on real mobile devices can't be compared to testing with the browser's DevTools. The reason is that it's not possible to emulate a real device in the DevTools. The touch screen itself plays a huge role in testing a website.

I recommend, if feasible, buying two Android phones to keep as test devices. Don't invest more than $150 per device. Another solution would be to use your family's phones. I always borrow my mom's phone to double-check things on Android.

**Tip**: If your web project is run on `localhost`, you can open its link on any mobile devices that are connected to the same Wi-Fi network. For example, if the project is running on `localhost:3000`, here is how macOS users can get the full IP address:

1. Go to "System Preferences", then "Network."
2. In the "Connected" section, note the IP address (mine is `192.168.1.252`).

3. On your mobile device, type the address with the port number (mine would be `192.168.1.252:3000`).

Then, you can access the project on your mobile device. From there, you can update and edit the CSS to test it.

## Mobile Browsers

The browsers on mobile devices are different from the ones we use on the desktop. They are simpler and more lightweight. On iOS, the default browser is Safari. On Android, it depends on the phone's manufacturer. For example, Samsung phones have a preinstalled browser named Samsung Internet.

## Inspecting Your Mobile Browser

By connecting your phone to your computer via USB or USB-C, you can inspect the code. For iOS, we can connect an iPhone and then inspect it with Safari on the computer. This makes checking and testing faster. For Android devices, the process is more complex. Follow along with this great resource from Chrome DevTools blog.

## Mobile Simulators

macOS developers can access the iOS simulator, where they can test on multiple iOS device sizes (iPhone, iPad). Also, it's possible to open the DevTools for each device you test.

## Browser Support

When starting a new front-end project, decide on the browsers you want to support. For example, will you support Internet Explorer 11? Or an old version of Firefox? Answer these questions ahead of time in order to prepare for what is coming.

During the development process, you might accidentally use a CSS feature that is not supported in the browsers you want to support. Because you are designing according to progressive enhancement, you'll need to check whether the CSS feature is supported, and, if so, then you would apply the feature as an enhancement.

Tools such as doiuse can be installed in your project via npm. Tell it the minimum browsers to support. The sample command below would run in the command line:

```
doiuse --browsers "ie >= 9, > 1%, last 2 versions" main.css
```

The output would list the CSS features, along with warnings — for example, that property X is supported only in Internet Explorer 11 and above.

## Can I Use

Can I Use is a tool that is very useful for searching for specific CSS features. It will tell you the history of the feature's support. Sometimes, the support table for a property will save you hours of trial and error when fixing an issue.

## Vendor Prefixes

Browser vendors add prefixes for experimental web features that are still not finalized. In theory, developers shouldn't use those properties on production websites until they are 100% supported; then, they can use the unprefixed versions. However, many developers are not patient enough to wait years for a property to be fully supported.

According to MDN:

> Browser vendors are working to stop using vendor prefixes for experimental features. Web developers have been using them on production Web sites, despite their experimental nature. This has made it more difficult for browser vendors to ensure compatibility and to work on new features; it's also been harmful to smaller browsers who wind up forced to add other browsers' prefixes in order to load popular web sites.

That means you won't see any vendor prefixes for future CSS features. That is great; it will make new features much faster to ship.

MDN adds:

> Lately, the trend is to add experimental features behind user-controlled flags or preferences, and to create smaller specifications which can reach a stable state much more quickly.

MDN lists the prefixes for all major browsers:

- `-webkit-`: Chrome; Safari; recent versions of Opera; almost all iOS browsers, including Firefox for iOS; basically, any WebKit-based browser
- `-moz-`: Firefox

- `-o-` : old pre-WebKit versions of Opera
- `-ms-` : Internet Explorer and Microsoft Edge

Here is sample usage of vendor prefixes:

```
-webkit-transition: all 4s ease;
-moz-transition: all 4s ease;
-ms-transition: all 4s ease;
-o-transition: all 4s ease;
transition: all 4s ease;
```

Adding those prefixes manually while developing a website is not practical. A tool named Autoprefixer will do it automatically for you. Specify the browsers to support, and it does the rest.

## Wrapping Up

In this chapter, we've covered debugging environments (such as the DevTools), virtual machines, mobile devices, and online services. Mastering every detail of the DevTools will substantially reduce the time you spend on solving and debugging CSS.

Next, we'll explore some common CSS issues and learn how to solve them. Ready?

# Chapter 4

## CSS Properties That Commonly Lead to Bugs

Web browsers have evolved a lot in the last few years, which has resulted in more consistent websites. Nevertheless, they're still not perfect, and some issues can confuse developers. Compounding the challenge are different screen sizes, multilingual websites, and human error.

As you implement a design, you might encounter CSS bugs for various reasons, both visual and non-visual, which I covered in Chapter 2. Some of them are hard to track.

In this chapter, we'll dive deep into CSS properties and their issues. Our goal is to get a detailed understanding of common bugs with certain properties, and to learn how to solve them properly.

## Box Sizing

The `box-sizing` property controls how the total width and height of an element are calculated. By default, the width and height of an element are assigned only to the content box, which means that `border` and `padding` are added to that width.

If you're not using a CSS reset file, you might forget to reset the `box-sizing` property, which could cause one of the following:

- **Horizontal scrolling**
  A block element takes up the full width of its parent. If it has a border or padding, this will add to its width, which will result in horizontal scrolling.
- **Oversized elements**
  An element's size can be bigger than you want it to be.

To avoid this, make sure that the property is reset properly:

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

*, *:before, *:after {
  box-sizing: inherit;
}
```

With that, we can be sure that the most important CSS property works as expected. It's worth mentioning that making `box-sizing` to be inherited is better because it will enable all elements to inherit this `box-sizing` property from the `html` element by default.

## Display Type

The `display` CSS property controls whether an element is a block or inline element. It also determines the layout type applied to its child items, such as grid or flex.

When used incorrectly, the `display` type can cause confusion for developers. In this section, we'll go through some ways in which the `display` property can go wrong.

### Inline Elements

Elements such as `span` and `a` are inline by default. Suppose we want to add vertical padding to a `span`:

```
span {
    padding-top: 1rem;
    padding-bottom: 1rem;
}
```

This won't work. Vertical padding doesn't work for inline elements. You would have to change the element's `display` property to `inline-block` or `block`.

The same goes for `margin`:

```
span {
    margin-top: 1rem;
    margin-bottom: 1rem;
}
```

This margin won't have an effect. You would have to change the `display` type to `inline-block` or `block`.

If you are wondering how to cook with passion,
don't worry, we will come into this later with details.

### Spacing and Inline Elements

Each inline element is treated as a word. Take the following:

```
<span>Hello</span>
<span>World</span>
```

This will render `Hello World`. Notice the spacing between the two words. Where did this come from? Well, because an inline element is treated as a word, the browser automatically adds a space between words — just like there is a space between each word when you type a sentence.

This gets more interesting when we have a group of links:


The links are next to each other, with a space between them. Those spaces might cause confusion when you're dealing with `inline` or `inline-block` elements because they are not from the CSS — the spaces appear because the links are inline elements.

Suppose we have an inline list of category tags, and we want a space of 8 pixels between them.

```
<ul>
    <li class="tag"><a href="#">Food</a></li>
    <li class="tag"><a href="#">Technology</a></li>
    <li class="tag"><a href="#">Design</a></li>
</ul>
```

In the CSS, we would add the spacing like this:

```
.tag {
    display: inline-block;
    margin-right: 8px;
}
```

You would expect that the space between them would equal 8 pixels, right? This is not the case. The spacing would be 8 pixels **plus an additional 1 pixel from the character spacing** mentioned previously. Here is how to solve this issue:

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

By adding `display: flex` the the parent, the additional spacing will be gone.

## Block Elements

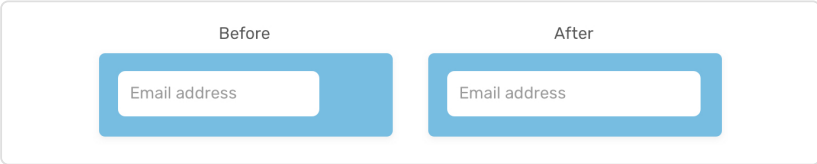
The `block` display type is the default for certain HTML elements, such as `div`, `p`, `section`, and `article`. In some cases, we might need to apply the `block` display type because an element is inline, such as:

- form labels and inputs,
- `span` and `a` elements.

When `display: block` is applied to `span` or `a`, it will work fine. However, when it's applied to an input, it won't affect the element as expected.

```
input[type="email"] {
    display: block; /* The element does not take up the full width. */
}
```

The reason is that form elements are **replaced elements**. What is a replaced element? It's an HTML element whose width and height are predefined, without CSS.




To override that behavior, we need to force a full width on the form element.

```
input[type="email"] {
    display: block;
    width: 100%;
}
```

There are replaced elements other than form inputs, including `video`, `img`, `iframe`, `br`, and `hr`. Here are some interesting facts about replaced elements:

- It's not possible to use pseudo-elements with replaced elements. For example, adding an `:after` pseudo-element to an `input` is not possible.
- The default size of a replaced element is 300 by 150 pixels. If your page has an `img` or an `iframe` and it doesn't load for some reason, the browser will give it this default size.

Consider the following example:

```
img { display: block }
```

We have an image with `display: block`. Do you expect that it will take up the full width of its container? It won't. You need to force that by adding the following:

```
img {
  display: block;
  width: auto;
  max-width: 100%;
}
```

It's worth mentioning that when an image fails to load, it's not considered a replaced element. You can actually add `::before` and `::after` pseudo-elements to it:

```
img::after {
    content: "The image didn't load";
}
```

### Spacing Below an Image

Have you ever noticed a little bit of space below an `img` that you've added? You didn't add a margin or anything. The space is there because the `img` is treated as an inline element, which is similar to having a character with some space below it.

To fix this, add `display: block` to the image. The spacing will be removed.

### The `legend` Element

If you are using `fieldset` to group form inputs, add a `legend` element. By default, it won't take up the full width of its parent unless you force it.

```
<fieldset>
  <legend>What's your favorite meal?</legend>

  <input type="radio" id="chicken" name="meal">
  <label for="chicken">Chicken</label>

  <input type="radio" id="meat" name="meal">
  <label for="meat">Meat</label>
</fieldset>
```

The `legend` element is block-level, but its width will stay the same because it has `min-width: max-content` by default, which means it has the width of its text content. To make it full width, do this:

```
legend {
    width: 100%;
}
```

Notice that the `legend` element doesn't accept `inline` or `inline-block`. If you use them as the display type, the computed display will be `block`.

## Using `display` With Positioned Elements

When an element has a `position` value of `absolute`, it becomes a block-level element by default. This means that adding `inline-block` or `block` as the display type won't affect it at all.

```
.element {
    position: absolute;
    left: 0;
    top: 0;
    width: 100px;
    display: block; /* Not necessary */
}
```

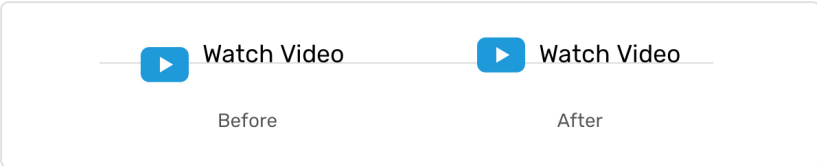
The only exception to using the display property with an absolutely positioned element is in the following example:

```
.element {
    position: absolute;
    left: 0;
    top: 0;
    display: none;
}

@media (min-width: 800px) {
    .element {
        display: block;
    }
}
```

In this case, the element is hidden for small views and shown for large ones. Using `display: block` in this way is OK.

## Alignment of Inline Elements

A common issue in CSS comes up when you try to align an icon and text. Right away, you notice that they are not aligned vertically. Either the icon or the text is off by a few pixels.




Thankfully, we can use the `vertical-align` property to fix that alignment issue. This works with the `inline`, `inline-block`, `inline-flex`, `inline-grid`, `inline-table` and `table-cell` display type.

```
.icon {
    vertical-align: middle;
}
```

## An Inline Display Overriding One in a CSS File

Suppose you're working on a layout, and you add an inline CSS style to hide an element. Later on, you forget about it, and you head over to the CSS file to try to make the element visible, and you add `display: block`. This won't work, because inline styles have a higher specificity than rules in a CSS file.

```
<p style="display: none;">Debugging CSS</p>
```

```
p {
    display: block;
}
```

## Float and Block Display

If an element has `float` applied to it, then the browser will display it as a block-level element, regardless of its type.

```
<img src="food.jpg alt="" />
```

```
img { float: left; }
```

An image is not a block-level element. In the third chapter of this book, I explained about greyed-out properties in the "Computed" panel. If you inspect an element that has `float: left` applied to it, you will notice that the

`display: block` property is greyed out. This means the display type is being defined by the browser, not by us.

## Float and Flex Display

It's worth mentioning that when you apply a `float` to an element with a display type of `flex` or `inline-flex`, it won't affect the element at all.

```
.element {
    display: flex;
    float: left; /* Has no effect! */
}
```

## Showing and Hiding the `br` Element

The `br` element produces a line break in the text, which is equivalent to hitting the `Enter` key while typing on a keyboard. It's useful to use in a paragraph in which you want two lines.

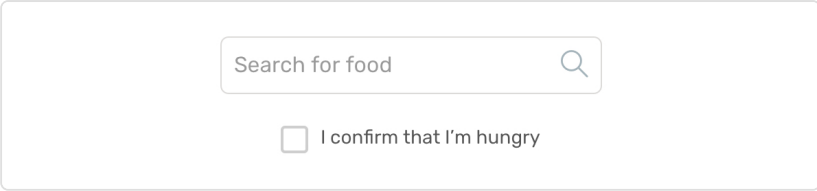
A less commonly known tip is that you can hide the `br` element with CSS. Say you want two lines on mobile, and three on desktop. This can be easily done!

```
@media (min-width: 800px) {
    br {
        display: none;
    }
}
```

## Situations to Avoid the Display Type

If you want to hide an element on the page, you might be tempted to use

`display: none` because it's easy and quick. But in doing so, the element will be completely hidden from screen readers. Let's look at a couple of cases in which you shouldn't use the `display` property.




### To Hide a Form's Input Label

Using only an `input`'s placeholder as the label is a common UI pattern. It's tempting to just hide the `label` with CSS. This is bad for accessibility. Hide the label only visually, using the method discussed in the third chapter.

### To Style a Checkbox

Styling a checkbox is possible in CSS. But you might run into an issue if you use `display` incorrectly. Hiding the input completely from the page will impair accessibility. Hide it visually only.

```
<input type="checkbox" id="food" class="visually-hidden"/>
<label for="food">Are you hungry?</label>
```

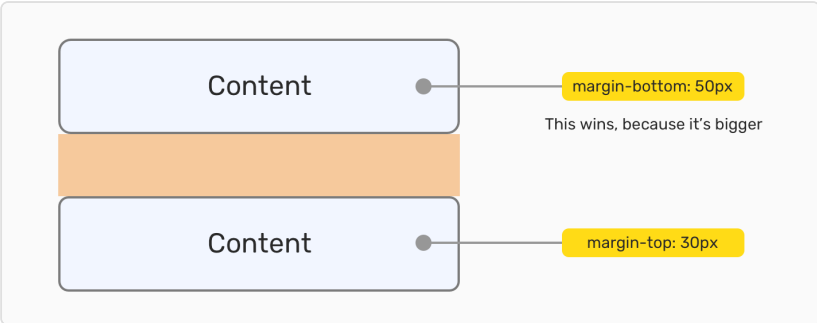
Here is a great article about styling a custom checkbox by Geoffrey Crofte.

## Margin

If two or more elements are close to each other on the page, the user will assume that they are related to each other. The `margin` CSS property is important in helping us make a design look more ordered and consistent.

### Margin Collapse

This is one of the most common issues with margins. Say you have two elements, the one above with `margin-bottom`, and the one below with `margin-top`. The greater of the two values will be used as the margin between the elements, and the other will be ignored by the browser.




```
<div class="item-1"></div>
<div class="item-2"></div>
```

```
.item-1 { margin-bottom: 50px; }
.item-2 { margin-top: 30px; }
```

Mixing top and bottom margins leads to problems. To avoid this, use one-

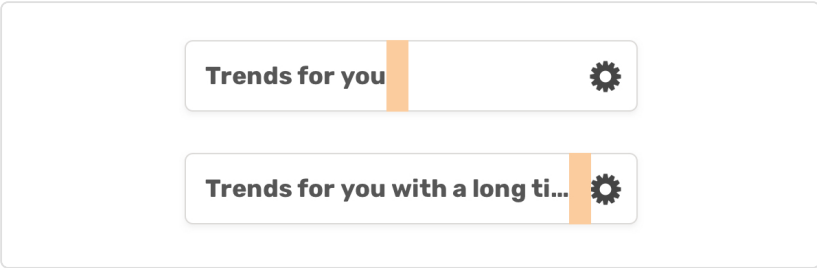
directional margins — for example, adding `margin-bottom` to all elements. Note that if an element is a part of a flexbox or grid container, then the margins won't collapse.

## Margin and Inline Elements

As mentioned in the `display` section, inline elements such as `span` don't accept vertical margins until their display type is changed. Make sure vertical margins are added to the correct type of element.

## Just-in-Case Margin

I call this a "just-in-case" margin because that's what it is. Suppose we have two elements:




We add a margin to the right or left side of one of the elements (in this case, to the right of the title), just in case its content becomes too long and brings the element too close to the adjacent one. If the title runs too long, the margin prevents it from sticking to the icon.

## Centering an Element

Because `margin: auto` is a popular way to center an element, it's important to mention that it only works with block-level elements.

```
span {
    width: 500px;
    margin: 0 auto;
}
```

This won't work unless the `span` is changed to a block-level element.

## Auto Margin and Positioning

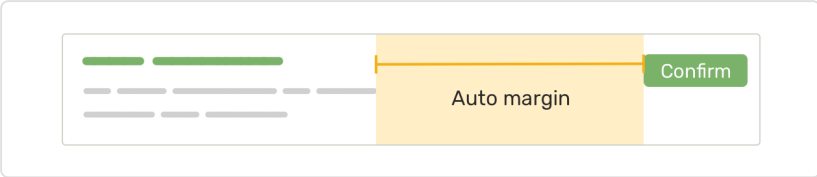
When an element is positioned absolutely, it's possible to use `margin: auto` to center it horizontally and vertically, without using transforms or other CSS techniques such as flexbox.


```
.element {
    position: absolute;
  left: 0; top: 0; bottom: 0; right: 0;
  width: 120px;
  height: 120px;
  margin: auto;
}
```

Setting a `width` and `height` on the element makes it possible to center the item with `margin: auto` only.

### Auto Margin and Flexbox

With flexbox, we can use an `auto` margin to push an element all the way over in one direction.



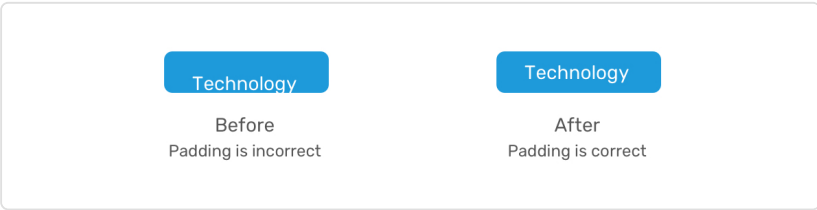

The button has the rule `margin-left: auto`, which pushes it to the far right. Flexbox and auto margins work great together for such purposes. We can use this technique to align an element without additional markup.

## Padding

The `padding` property adds space **inside** an element. That's what differentiates it from `margin`. There are some misconceptions about padding. Let's explore them.

### Using Padding With Height

Suppose you have an element with a fixed height, such as a button. Controlling the vertical spacing within the element can be confusing because large padding values might push the text downwards and break the button.

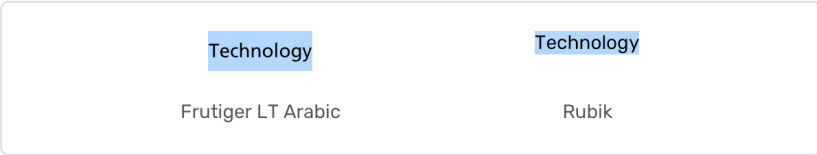



In the button on the left, notice how the text is pushed too far down. The reason is because of this in the CSS:

```
.button {
    height: 40px;
    padding-bottom: 10px;
}
```

A `button` element should never be given a fixed height. It will make things complex and controlling the button will be harder. Instead, you should use vertical `padding`.

You might encounter a case when dealing with a web font that has additional spacing in its characters. In this case, you might need to tweak the top or bottom padding in order to center the button's text vertically.




```
.button {
    padding: 3px 16px 8px 16px;
}
```

Here, we've tweaked the padding so that the text can be centered vertically in the button.

## Padding and Inline Elements

As mentioned in the `display` section, vertical padding won't work unless an element's display type is something other than `inline`.

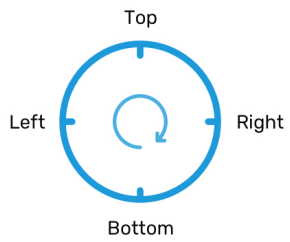
## The Padding Shorthand

The shorthand for padding is ordered as top, right, bottom, left. It's sometimes confusing to use, and the same applies for `margin` as well. You could end up doing the following:

```
.element {
    padding-top: 20px 20px 0 20px;
    /* … instead of: */
    padding: 20px 20px 0 20px;
}
```

This would be an error. The `padding-top` property takes only one value, so writing four values would make the rule invalid. It could be a reason why the padding is not working as you intended. Make sure to type the correct property name.

Remembering the correct order of the `padding` and `margin` shorthand is confusing, even for an experienced front-end developer. A clock is a simple way to remember it:




Remember to start from the `top`, and the rest will follow.

## Percentage-Based Padding

Using a percentage for padding is OK, but to make it work as you expect, remember that it works based on the element's width.

```
.element {
    width: 200px;
    padding-top: 10%;
}
```

The computed value of this element's `padding-top` is `20px`. Remember how it's calculated when you're debugging an element with percentage-based padding.

It's worth mentioning that percentage-based padding for `top` and `bottom` was treated differently for flex items in old versions of Firefox. Firefox used the height, rather than the width, of an element to determine the padding's value. This issue got fixed in Firefox 61.

## Width Property

Setting width is one of the most important things in web design. We can set a width explicitly or implicitly. In this section, we'll go over cases in which `width` might be confusing.

### Inline Elements Don't Accept a Width or Height

An inline element such as `span` won't accept the `width` or `height` property. This can be confusing. An element accepts `width` and `height` only if its display is set to something other than `inline` (such as `inline-block` or `block`).

### Fixed Width Is Not Recommended

When a fixed width is used on an element, there is a high probability that it will cause horizontal scrolling on mobile. Using `max-width` is better because it will prevent the element from being wider than the viewport.


Here we have a list of headings, along with a description. The description

needs to have a maximum width to keep the number of characters per line easy to read. If you use a fixed width for the text, you'll notice horizontal scrolling on mobile. I spent five minutes wondering about the reason for the issue, before identifying a fixed width as the culprit.

## Full Width for Image

By default, an HTML `img` will be sized according to its content. To prevent an image from being larger than the viewport, we can set the `width` property.

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

With `width: 100%`, an `img`'s width will be equal to its parent's width. However, sometimes we don't want that behavior. There is a better alternative, which is to set `max-width`.

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

The method above ensures the following:

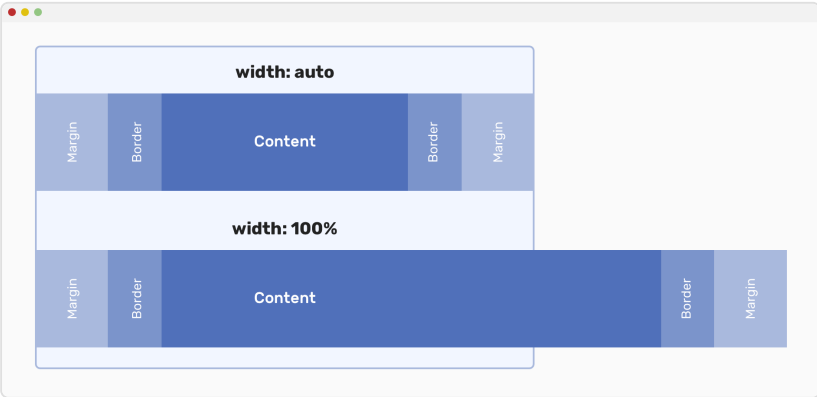
- A small image (say, 650 by 250 pixels) won't take up the full width of a wide parent (say, 1500 pixels). Imagine such an image taking up that container! It would look pixelated.
- On the other hand, if an image is wider than the viewport, then its width would be equal to 100% of its parent.

## Using `100%` vs. `auto` for Width

The initial width of block-level elements such as `div` and `p` is `auto`, which lets the elements take up the full width of their parent. In some cases, you might need a `div` not to take up the full width.

```
div {
    width: 50%;
    margin: 20px;
}

@media (min-width: 800px) {
    div {
        width: 100%;
    }
}
```

This element's width is `50%` of its parent. And when the viewport is big enough, we want it to take up the full width. Setting the width to `100%` would cause its contents to take up the full width of its parent without the `margin` being calculated.




This is a problem. To solve it, we should use `auto` instead of `100%`. According to the CSS specification:

> 'margin-left' + 'border-left-width' + 'padding-left' + 'width' + 'padding-right' + 'border-right-width' + 'margin-right' = width of containing block

Notice that when `box-sizing: border-box` is used, `padding-left` and `padding-right` are **not** included in the calculation.

Setting the width to `auto` would result in the width of the content box being the content itself minus the margin, padding, and border.

```
@media (min-width: 800px) {
    div {
        width: auto;
    }
}
```

I've written a detailed article about `auto` in CSS that is worth checking out if you want to dig more into the topic.

## An Image With `position: absolute` Doesn't Need Width or Height

You might not think about this, but it's interesting to know. Consider the following:

```
<div class="media">
    <img src="cool.jpg" alt=""/>
</div>
```

```
.media {
    position: relative;
    width: 300px;
    height: 200px;
}

.media img {
    position: absolute;
    left: 0; top: 0; right: 0; bottom: 0;
}
```

You might expect that the image will take up the full width of its parent because it is absolutely positioned against the four sides. Well, that would be wrong. If the image is large enough, it will break out of its parent.

To prevent this from happening, set a width and height for the image.

```
.media img {
    position: absolute;
    left: 0; top: 0; right: 0; bottom: 0;
    width: 100%;
    height: 100%;
}
```

# Height Property

## Full Percentage-Based Height

Setting a percentage-based height in CSS might seem intuitive at first, but it's not. You can't set a percentage-based height for an element unless the height of its parent is **explicitly** defined.

```
.parent {
    padding: 2rem;
}

.child {
    height: 100%;
}
```

The child won't take up the `100%` of its parent. Here is how to make it take up the full height:

```
.parent {
    height: 200px;
    padding: 2rem;
}
```

This way, the percentage-based height value of the child will be based on something, and it will work as expected, even if using an absolute `height` value is not recommended.

## Filling the Height of the Remaining Space Available

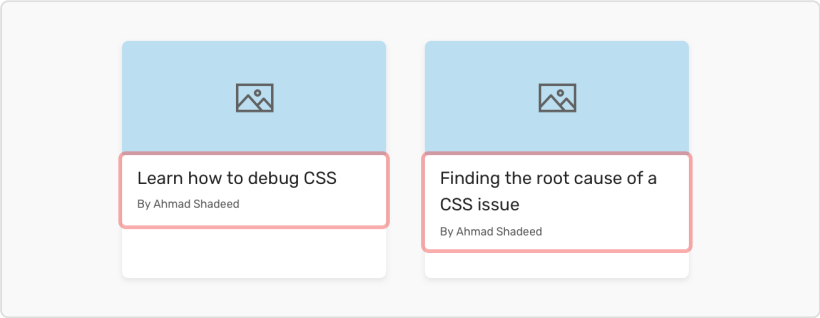
Let's suppose that we have a grid of cards, and we're using CSS grid to lay them

out.

```
<div class="media-list">
    <div class="card">
        <img class="card__thumb" src="thumb.jpg" alt="">
        <div class="card__content">
            <h2><!-- Title --></h2>
            <p class="card__author"><!-- Author --><p>
        </div>
    </div>
    <div class="card"></div>
</div>
```

```
.media-list {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(265px, 1fr));
    grid-gap: 1rem;
}
```

By default, CSS grid will make the height of the cards equal, and that's useful. But there is a problem, when one card has a longer title than the other, the `.card__content` element height will be different.

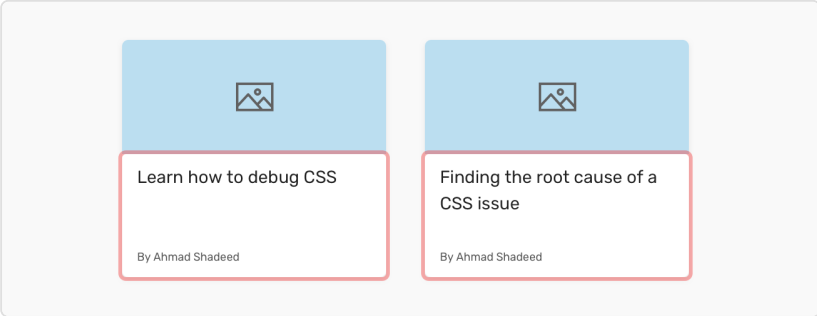


To solve this, we need to make the card as a flex container, and then force the `.card__content` to fill the available space.

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

.card__content {
    flex-grow: 1;
}
```

Now, we want to make the `.card__content` element as a flex container. Finally, the `.card__author` element will be given `margin-top: auto` so it can always be at the baseline of the card.

```
.card__content {
    flex-grow: 1;
    display: flex;
    flex-direction: column;
}

.card__author {
    margin-top: auto;
}
```



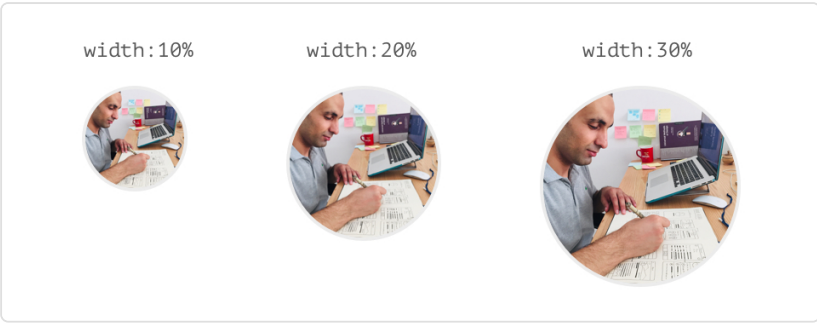

## Percentage-Based Width and No Height

Sometimes, you need a way to resize an element without having to change both the width and height. I like a pattern that I found on Twitter's website, which resizes the avatar by changing only the `width` property.

```
<a href="#" class="avatar">
    <div class="avatar-aspect-ratio"></div>
    <img alt="" src="me.jpg">
</a>
```

```
.avatar {
  position: relative;
  width: 25%;
  display: block;
}

.avatar-aspect-ratio {
  width: 100%;
  padding-bottom: 100%;
}

.avatar img {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  width: 100%;
  height: 100%;
}
```

By adding an element ( `.avatar-aspect-ratio` ) with a rule of `padding-bottom: 100%` , which ends up being equal to the `width` of the avatar, the result will be a square. The image itself is positioned absolutely.



Notice that only the `width` property is being resized; the height will follow. For more details about the technique, here is a great article on CSS Tricks.

## Height and Viewport Units

We can use a viewport unit with `width` or `height` to make an element take up the full width or height of the viewport. We'll deal here with the viewport-height unit.

```
body {
    height: 100vh;
}
```

This will make the `body` element take up the full height of the viewport. However, Safari on mobile has a problem because it doesn't include the address bar in its calculation, which results in a higher value for the `height`.

One solution is to get help from JavaScript by using the `innerHeight` method.

```
// Get the viewport height and multiply it by 1% to get the vh value.
let vh = window.innerHeight * 0.01;
// Set the vh value in the CSS property
document.documentElement.style.setProperty('--vh', `${vh}px`);
```

Then, we use that in CSS:

```
.my-element {
  height: 100vh; /* Fallback for browsers that do not support custom
properties */
  height: calc(var(--vh, 1vh) * 100);
}
```

By getting the `innerHeight` of the browser, we can use that value in the `height` property.

There is a solution that doesn't use JavaScript, which I learned from @AllThingsSmitty.

```
.my-element {
  height: 100vh;
  height: -webkit-fill-available;
}
```

By using an intrinsic value for the `height`, the browser will fill only the available vertical space. The downside is that this breaks in Chrome because that browser also understands `-webkit-fill-available` and won't ignore it. My advice is **not to use** this solution until its behaviour is consistent in browsers.

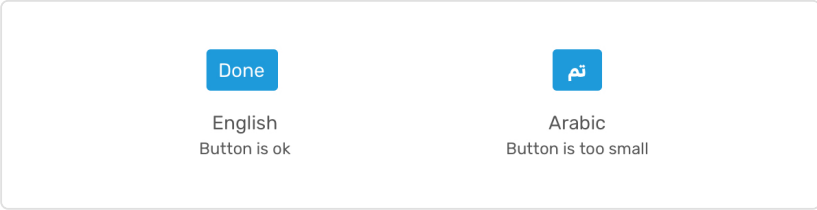
## Setting a Minimum or Maximum Width

In CSS, we have the `min-width` and `max-width` properties. Let's explore the common mistakes and points of confusion that happen with them.

### Minimum Width

#### Minimum Width for Buttons

When setting a minimum width for a button element, keep in mind that it should work across multilingual layouts.

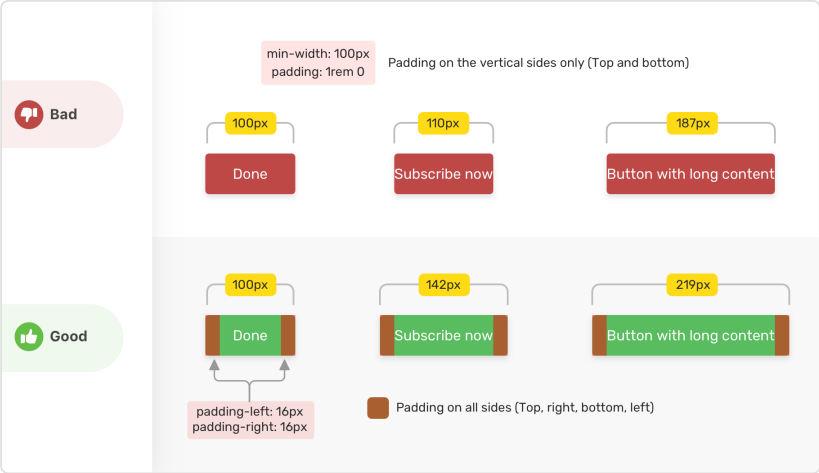



Here, we have a button with `min-width: 40px`. It works perfectly for English layouts. However, when translated to Arabic, it becomes very small because of its minimum width. This example is from Twitter's website. The issue here is having a very short width for a button, it will make it harder to the user to notice it.

To prevent such an issue, always test with multiple kinds of content. Even if the website is for one language only, testing with different content won't hurt.

### Minimum Width and Padding

Another point of confusion is when we depend only on `min-width`. For example, a button with `min-width` might look good because the content suits the size. However, when the button's text gets a bit long, the text will run close to the edges.

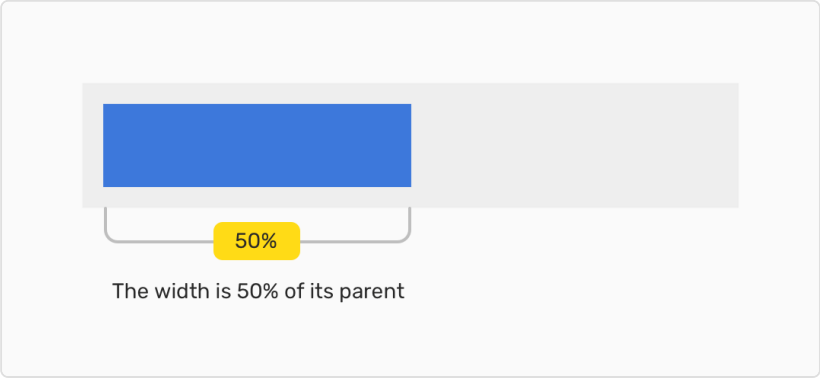



The reason for this is that the author is blindly depending on `min-width` and has not considered whether the content might be longer.

### Which Has Higher Priority: `min-width` or `max-width`?

When using both `min-width` and `max-width` for an element, it might be confusing to know which of them is active. Let's clarify this.

If the value of `min-width` is greater than that of `max-width`, then it will be taken as a width.




```
.element {
    width: 100px;
    min-width: 50%;
    max-width: 100%;
}
```

### Resetting `min-width`

Let's explore ways to reset the `min-width` property in CSS.

#### Setting to `0`

The default value of `min-width` is `0`, but this is different for flex child items. The `min-width` of flex child items is `auto`, as explained previously.

#### Setting to `initial`

The `initial` value will reset to the browser's initial value, which would be either `0` or `auto`, depending on whether the item is a flex child.

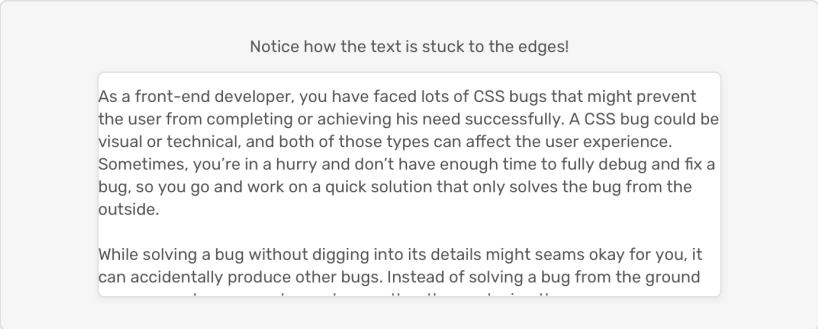
Generally, I recommend using `initial` to reset. However, depending on the use case, you might need to use `min-width: 0` for flex child items.

## Max Width

### Max Width for Page Wrappers

A common use case for the `max-width` property is to add it as a constraint on an element, such as a page wrapper or container.

```
.wrapper {
    max-width: 1200px;
    margin: 0 auto;
}
```




This might seem OK, until you resize the screen to be narrower than 1200 pixels. Then you'll notice that the child elements of `.wrapper` are stuck to the left and right edges, which is not want we want. Make sure to add padding to the page container so that it has a horizontal offset on mobile.

```
.wrapper {
    max-width: 1200px;
    margin: 0 auto;
    padding-left: 16px;
    padding-right: 16px;
}
```

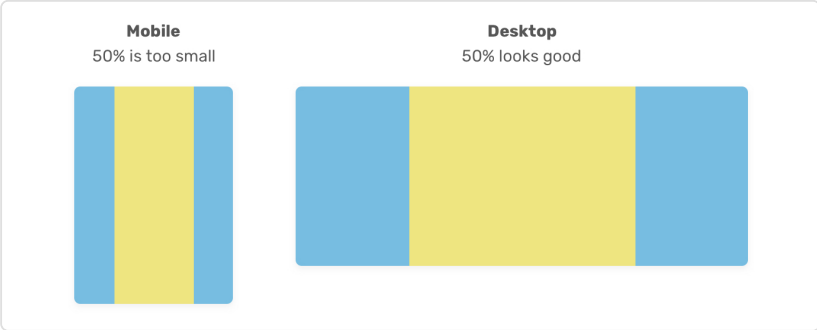
Notice how the text is stuck to the edges!

As a front-end developer, you have faced lots of CSS bugs that might prevent the user from completing or achieving his need successfully. A CSS bug could be visual or technical, and both of those types can affect the user experience. Sometimes, you're in a hurry and don't have enough time to fully debug and fix a bug, so you go and work on a quick solution that only solves the bug from the outside.

While solving a bug without digging into its details might seams okay for you, it can accidentally produce other bugs. Instead of solving a bug from

### Percentage for Maximum Width

When using a percentage value for the maximum width, it's common to forget about it on mobile.




```
.element {
    max-width: 50%;
}
```

This might work smoothly on laptops or desktops. However, on mobile, the `50%` could be 150 or 200 pixels, depending on the viewport's width. Whatever it is, the computed pixel value will be very small, so it's important to consider mobile sizes.

```
@media (min-width: 800px) {
    .element {
        max-width: 50%;
    }
}
```

Much better. The media query will activate the `50%` width once there is enough space.

### Setting a Maximum Width Based on the Content

This could be regarded as either a common mistake or a common need, so I

will try to address them as both. Sometimes, you need to set a maximum width based on the content you have. This can be tricky when the content varies. The mistake is setting the width based on the content.


We have a section with a heading and a description. We want the wrapper to be as wide as the content, so we'll try setting it in pixels.

```
.wrapper {
    max-width: 567px;
}
```

Using a hardcoded value like `567px` in CSS is not a good practice because this fails easily when the content changes. The solution is to use an intrinsic CSS value.

```
.wrapper {
    max-width: max-content;
}
```

This way, the width of the wrapper will adjust to the content, without our having to hardcode the value.

### Constraining an Image in a Wrapper

A common use case for `max-width` is to constrain an `img` not to be bigger than its container. Because the `img` element is a **replaced element**, its size is based on its content.

Sometimes, a large image could extend beyond its container. The solution is simply to use the previously mentioned technique:

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

### Resetting `max-width`

Suppose we need to reset a CSS property for a particular viewport size or condition. There are a couple of ways to reset `max-width` in CSS.

#### The `none` Keyword Value

The `none` value sets no limit on the size, which is exactly the goal of resetting the property.

#### The `initial` Keyword Value

This sets the property to its initial default value, which is `none`.

#### The `unset` Keyword Value

The `unset` keyword resets the value to the inherited value if the property

inherits from its parent. If not, the value will be `initial`.

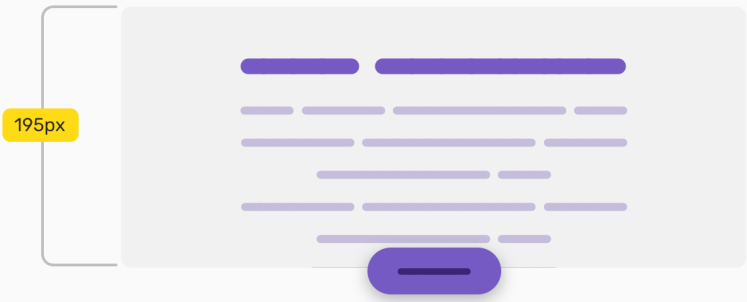
I recommend using the `none` keyword because it's the clearest, and you won't have to think through the consequences.

## Minimum Height

### Setting a Minimum Height for Variable Content

A common challenge in CSS is setting a fixed height for a section with content that will change or that is inputted by the user. Setting a fixed height might break the section if the content goes too long.

Using `min-height` fixes this. We set a value that would be the minimum height, and if the content grows longer, the height of the section will expand.




Fixed height hero, long content

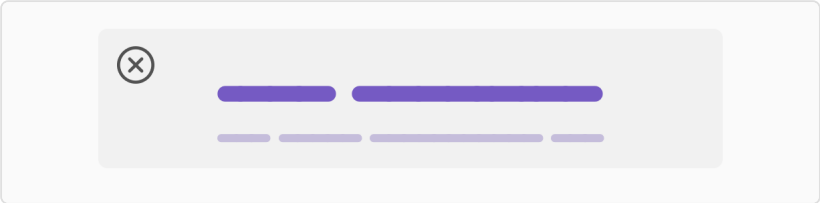
Notice how the content overflows the section vertically. This is because it has a fixed height.

```
section {
    min-height: 450px;
    /* … instead of… */
    /* height: 450px; */
}
```

This fixes the issue.

### Setting a Minimum Height for Positioned Elements

Usually, a modal component contains content such as form elements, text, an image, etc. In case the content is too short, the modal height will collapse and the layout will look bad.



Setting `min-height` is better so that the modal can't go below that value. Thus, we'll prevent any unwanted behavior.

```
.modal-body {
    min-height: 250px; /* 250px is just an example. Tweak according to
your project's needs. */
    padding: 16px;
}
```

## Maximum Height

### Setting a Maximum Height for Positioned Elements

Let's start with this issue because it's related to the previous one about modal content. What if the modal's content is too tall? The height of the modal will become equal to the viewport's height, which is not good.

So, we should use not only `min-height`, but also `max-height`, so that however tall the content is, it won't exceed the value we've set.

```
.modal-body {
    min-height: 250px;
    max-height: 600px;
    overflow-y: auto;
}
```

Don't forget to make the modal scrollable by adding `overflow-y: auto`. Without it, the content will exceed its parent.

### Setting a Percentage-Based Maximum Height

This one is also related. We set the maximum height in pixels, remember? This will work, but it has a pitfall. What if the viewport's height is too short and the value of `max-height` is greater than it?

A better solution would be to use a percentage for `max-height`. This way, whatever the length of the content, the height of the modal won't exceed that value.

```
.modal-body {
    min-height: 250px;
    max-height: 90%;
    overflow-y: auto;
}
```

### Transitioning an Element's Height

A common question I hear is how to transition the `height` property of an element. Unfortunately, the property is not animatable because often times, we want to animate the height from `0` to `auto`, and the value `auto` is not valid for animation. It's possible to use JavaScript, by adding `height` as an inline style and incrementing it.

There is a CSS solution that is kind of a hack, but it works. By using the `max-height`, we can set a maximum value, and it will transition.

```
.element {
    max-height: 0;
    overflow: hidden; /* This prevents child elements from appearing
while the element's height is 0. */
    transition: max-height 0.25s ease-out;
}

.element.is-active {
    max-height: 200px;
}
```

### Maximum Height Depending on the Element's Defined Height

If an element has `max-height: 90%`, then it needs one of the following to work:

- a parent or containing block with an explicitly defined `height`,
- an absolutely positioned element.

When you apply `max-height` with a percentage value, make sure one of the conditions above is met. Otherwise, the computed value will be `none`.

## Shorthand vs. Longhand Properties

As you can guess, a shorthand is the short version of a CSS property, and a longhand is the long one.

```
.element {
    padding: 10px;
}
```

This is a shorthand property. This `padding` has four values, all of them being

`10px`. We could write the four values like so:

```
.element {
    padding: 10px 10px 10px 10px;
}
```

But because they are all equal, there is no need to write them out. The longhand version looks like this:

```
.element {
    padding-top: 10px;
    padding-right: 10px;
    padding-bottom: 10px;
    padding-left: 10px;
}
```

When setting a background for an element, it will be either a solid color or an image. We have to be mindful of how we write it. Suppose we write this:

```
.element {
  background: green;
}
```

We'll get a green color, but actually we're doing this:

```
.btn {
  background-image: initial;
  background-position-x: initial;
  background-position-y: initial;
  background-size: initial;
  background-repeat-x: initial;
  background-repeat-y: initial;
  … and so on…
  background-color: green;
}
```

Because `background` is a shorthand property, it will **reset** all other background properties to their initial values when added. This will introduce some confusing bugs to your layout. Use the longhand properties in such situations.

A similar case happened to me multiple times using `margin`.

```
.wrapper {
    margin: 0 auto;
}
```

What if we defined `margin-top` and `margin-bottom` for the wrapper earlier? Our new CSS declaration will reset them to `0`. Believe me, when you've been working all day and make a mistake like this, you could spend an hour before realizing why this is happening. Use the shorthand properties only when needed, as CSS Wizardy advises. Here is a proper use:

```
.button {
    padding: 10px 12px 15px 10px;
}
```

We might set the `padding` for a button in this way because it has a weird font that is causing some alignment issues.

## Positioning

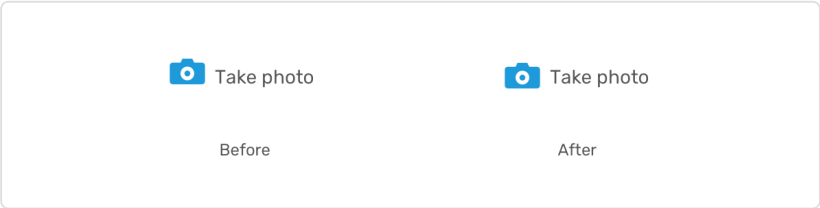
CSS positioning issues often happen because of incorrect use of the `position` property, whether because the author doesn't completely understand it or because of a plain old bug.

### Using the Positioning Offset Properties

When using one of the properties `top`, `right`, `bottom`, or `left`, make sure that the position is not `static`, the default value. If it is, then the offset properties won't have any effect on the element.

### Icon Alignment

Sometimes, aligning an icon to the text beside it is a challenge. Even with properties like `vertical-align` and flexbox, it's still not easy. The reasons for such issues vary, but the most annoying reason is having a font with space above and below its characters. In this case, using `position` is a good fix.




```
.icon {
    position: relative;
    top: 3px;
}
```

We push the icon 3 pixels towards the bottom. Granted, we are hardcoding the value here, but in this case it's a valid use. The downside is that when the font changes, the icon's alignment might break, so be aware of that.

## Using the `width` and `height` Properties

I have come to notice an unnecessary pattern of using the `width` or `height` properties for positioned elements.

```
.element {
    position: absolute;
    left: 0; top: 0; right: 0; bottom: 0;
    width: 100%;
    height: 100%;
}
```

This element is already positioned to the four sides. It's already taking up the full space, so setting the width and height is not needed.

**Reminder:** This doesn't apply to HTML replaced elements such as `img`. If the element above was a large image, then the width and height would be needed, otherwise you could expect horizontal scrolling.

## How Padding Works for Positioned Elements

A positioned element can have spacing that offsets it from the four sides of its parent element. If we want an element to have a 10-pixel offset, then each property of `top`, `right`, `bottom`, and `left` should have a value of `10px`.

There are some tricky situations involving the `padding` and offset properties.


Padding

Here we have a card with a footer that is offset from the left, right, and bottom sides by 12 pixels. How would we do that in CSS?

```
.card-footer {
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    padding: 0 12px 12px 12px;
}
```

We position the footer to the four sides, and we rely on `padding` instead of the offset properties for the 12 pixels. This way, it's easier to control when testing. You might need to tweak the `padding` value.

## Using `z-index`

The `z-index` property is responsible for setting the order of positioned elements and their descendants on the z-axis. It doesn't work unless `position` is set to something other than `static` or if the element has properties that trigger a new stacking context like: `transform`, `opacity` less

than one, and others. We will get into that later.

### Resetting the Position

Resetting the position of an element from `absolute` or `fixed` to another value can get confusing. For instance, if we have an element that should be `absolute` only on mobile, we could do the following:

```
.element {
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
}

@media (min-width: 700px) {
    position: static;
}
```

Setting the value to `static` will reset the value of `position`. When that is done, keeping the values of `top`, `right`, `bottom`, `left` is fine because they won't have any effect.

## The Z-Index Property

The `z-index` property enables us to control how HTML elements are positioned on top of each other, using numeric values. At first, it might seem that positioning an element on top of its siblings or parent is as simple as setting `z-index` to `999999`. That's not always the case — `z-index` has certain rules to be followed. Let's explore the most common issues with it.

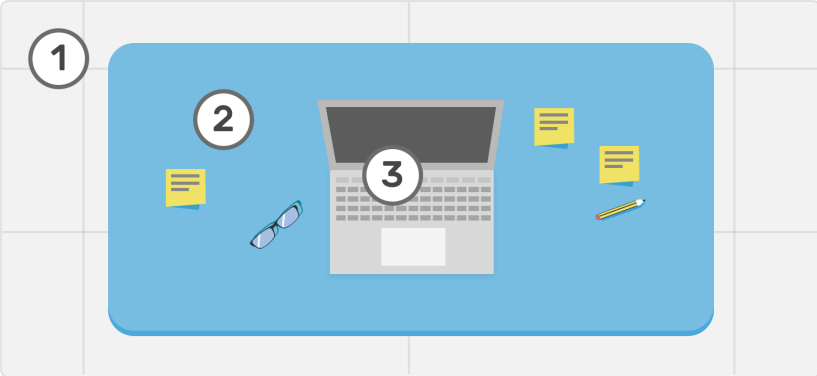
## Forgetting to Set the Position

The `z-index` property won't work with `position`'s default value of `static`. The value must be `relative`, `absolute`, `fixed`, or `sticky`. Make sure to set a position or to double check that stacking context exists.

## Default Stacking Order

In HTML, an element that sits at the bottom of a container will be positioned above the preceding elements.

```
<div class="home">
    <!-- ::before element -->
    <div class="floor"></div>
    <div class="desk"></div>
    <div class="laptop"></div>
    <!-- ::after element -->
</div>
```




Hopefully, this real-life example makes it clearer. The laptop is on the desk,

and the desk is on the floor. That is, the last child will be positioned on top of its siblings by default. Without understanding this, things might get confusing.

The same goes for pseudo-elements. In the HTML markup, notice how the `::before` and `::after` pseudo-elements are added to the `.home` element. The `::after` element will appear in the layout on top by default, and the `::before` element will appear under everything else in a normal stacking context.

## CSS Properties That Create a Stacking Context

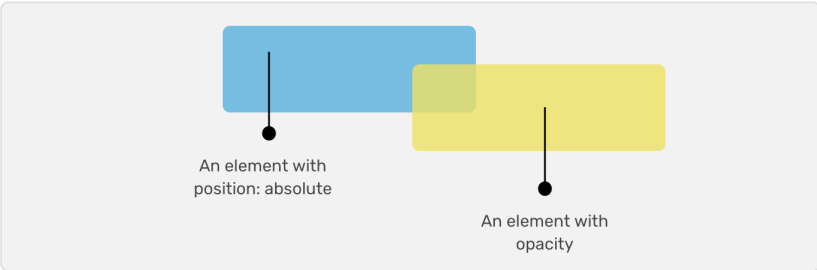
Some properties will trigger a new stacking context. The CSS specification lists the properties that trigger a stacking context. They include a `position` value other than `static`, `opacity`, `transform`, `filter`, `perspective`, `clip-path`, `mask`, and `isolation`.

```
<div class="elem-1"></div>
<div class="elem-2"></div>
```

```
.elem-1 {
    position: absolute;
  left: 10px;
  top: 20px;
}

.elem-2 {
    opacity: 0.99;
}
```

Which element will appear on top of the other? In this case, `elem-2` will be on top, because adding the `opacity` value will trigger a new stacking context,

thus putting `elem-2` on top of `elem-1`, even though `elem-1` is absolutely positioned.




When `z-index` is not behaving as expected, check whether any properties have triggered a new stacking context.

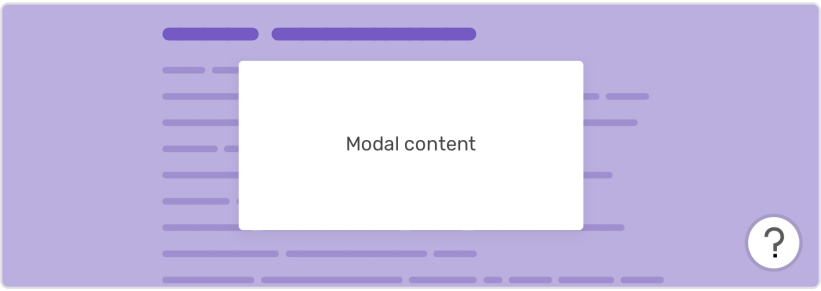
## An Element Can't Appear Above Its Parent's Siblings

For this issue, let's start with the HTML first, so that you can imagine it better.

```
<div class="wrapper">
    <!-- other content -->
    <div class="modal"></div>
</div>

<div class="help-widget"></div>
```

```
.wrapper {
    position: relative;
    z-index: 1;
}

.modal {
    position: fixed;
    z-index: 100;
}

.help-widget {
    position: fixed;
    z-index: 10;
}
```

From the markup, can you tell which element will be on top? It's tempting to think that the `.modal` element will be on top because it has a higher `z-index`, right?



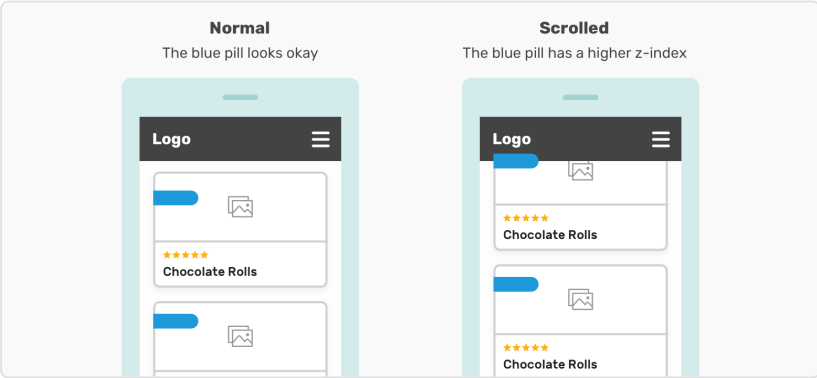

Wrong. The `.modal` element is a child of `.wrapper`, and the `.help-widget` element is a sibling of `.wrapper`. Positioning `.modal` above `.help-widget` is not possible unless we change the markup:

```
<div class="wrapper">
    <!-- other content -->
</div>

<div class="modal"></div>

<div class="help-widget"></div>
```

Thus, we can position `.modal`. As a rule of thumb, if you have an element such as a modal or popup, keep it outside of the page's main wrapper, to avoid such confusion.

## An Element Floating Above Its Siblings

One tricky case is when an element has a higher `z-index` than a fixed header. This issue can easily trip us up because it isn't easy to notice.




These cards have a blue element positioned absolutely in the top-left corner (they might indicate the category of the card). When the user scrolls down, the category will scroll above the fixed header. Fixing this is simple: You just

need to set an appropriate `z-index` value.

## The `calc()` Function

CSS' `calc()` function allows us to calculate the values of certain CSS properties. A common mistake with writing `calc()` is omitting spaces.

```
.element {
    width: calc(100%-30px); /* Invalid */
}
```

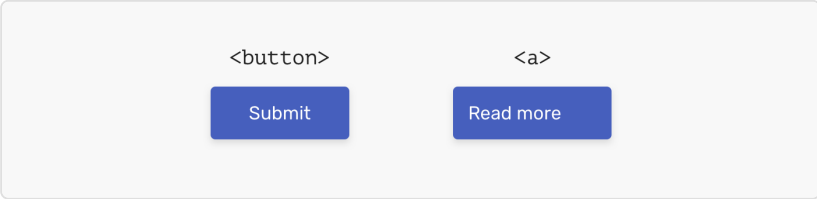
This value is invalid. You must add spaces around the subtraction symbol.

```
.element {
    width: calc(100% - 30px); /* Valid */
}
```

## Text Alignment

### Forgetting to Center a Button's Content

Suppose you would like to add some CSS classes to an HTML `button` or to an `a` link that is functioning like a button. A `button`'s content is centered by default, but an `a` element is not. So, you should center the latter manually.




```
<button class="button" type="submit">Submit</button>
<a class="button" href="/debugging-css">Read more</a>
```

```
.button {
    /* Other styles */
    text-align: center;
}
```

Without doing this, you might be surprised to find later that some buttons on your website are left-aligned!

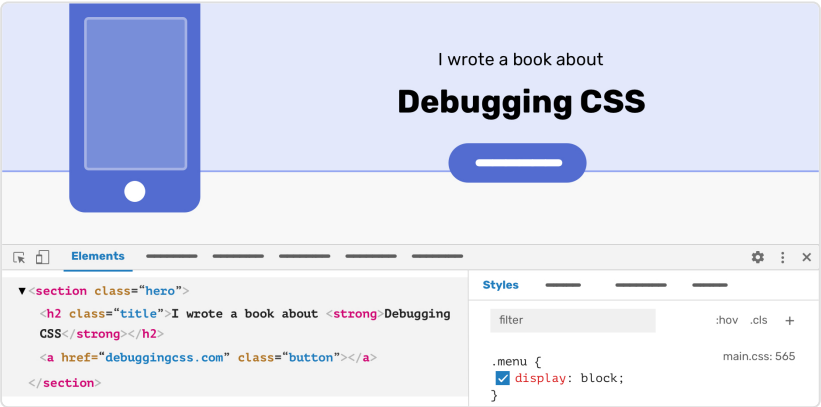
## Viewport Units

### Using `height: 100vh` Is Risky

When you add `height: 100vh` to, say, a hero element, the elements within it might look fine when the viewport is tall enough. I was once browsing someone's website on a 15-inch laptop. The hero section took up 100% of the viewport's height. It looked great!

I got curious and opened up the DevTools to see how it's built, and — boom! — the hero section broke. The elements within it overlapped the next section. The elements in the hero section didn't fit the available height once I opened

the DevTools. Why? It's because when `100vh` is used, opening the DevTools or shrinking the browser's height will reduce the height.




Speaking of which, the DevTools can be annoying when you're testing for the viewport's height. I usually unlock the DevTools to a separate window when debugging for the viewport's height.

## Pseudo-Elements

CSS pseudo-elements are one of the most useful additions to CSS. Misusing them can be confusing, so let's explore some common issues with them.

### Forgetting the `content` Property

The core of a pseudo-element is the `content` property. We often forget about it and set the following:

```
.element::before {
    display: block;
    background: #ccc;
}
```

Then, we wonder why the element does not appear. I've spent some time fixing such a bug. To avoid this, make sure that the `content` property is the first thing you add when creating a pseudo-element, before rushing on to other properties.

## Using Width or Height

The default `display` value of a pseudo-element is `inline`. So, when you add a width, height, vertical padding or vertical margin, it won't work unless the `display` type is changed to a value other than `inline`.

```
.element::before {
    content: "";
    background: #ccc;
    width: 100px;
    height: 100px; /* Neither the width nor height will work. */
}
```

## Using Pseudo-Elements With Grid or Flexbox

When you apply flexbox or grid to a container, any pseudo-element within it will be treated as a normal element. That might be confusing and could cause unexpected issues.

One common issue I remember is applying flexbox to the `.row` element in Bootstrap 3. Because the columns were built with the `float` property, the

`.row` element had `::before` and `::after` pseudo-elements:

```
.row::before,
.row::after {
    content: "";
    display: table;
}
```

This is the "clearfix" hack, which fixes the layout of floated elements without the addition of presentational markup.

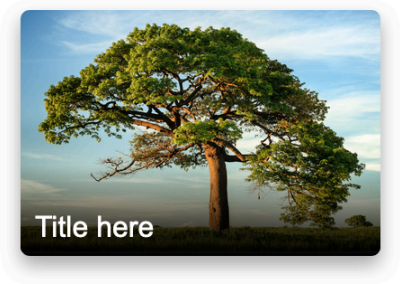
When flexbox is applied to the `.row` element, the two pseudo-elements will be treated as normal elements, and that can create some weird spaces in the layout. In such a case, the pseudo-elements wouldn't have any benefit, so they should be hidden.

```
.row::before,
.row::after {
    display: none;
}
```

### When to Use `::before` and When to Use `::after`

The `::before` pseudo-element becomes the first child of its parent, whereas `::after` is added as the last child. You might wonder whether this is useful?

There is a common use case for pseudo-elements, which is to absolutely position an overlay on top of a card component. In this case, it would matter whether you use `::before` or `::after`, because one of them will be easier to deal with. Can you guess which? Consider the following example:




```
<article class="card">
  <img src="article.jpg" alt="">
  <h2>Title here</h2>
</article>
```

We need to add a gradient overlay to make the text easy to read. The stacking order of absolutely positioned elements (The title and the `::after` element) starts from bottom to top. The element at the very bottom, the `h2`, will appear on top of the image. If we use `::after` for the gradient overlay, it will be the last element, which will put it on top of everything, so we would need to use `z-index: -1` to move it below the title.

However, if we use `::before`, then the gradient would appear below the title by default, without any adjustment to the `z-index`. Thus, we save additional work and avoid a bug.

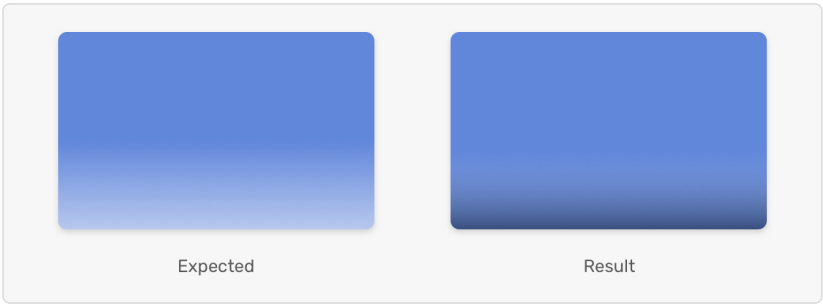
```
.card::before {
    content: "";
    /* The CSS for the gradient overlay */
}
```

## Color

The `color` property is an important one in CSS because it sets the color of text elements. It might sound simple, but it's not. Using it incorrectly can cause problems and additional work.

### The `transparent` Keyword

The `transparent` keyword is a shortcut for `rgba(0, 0, 0, 0)`. Some browsers compute it as black with an alpha value of `0`. This can make a transparent gradient look a bit black-ish.




This behavior was seen in old versions of browsers such as Chrome and Safari. To prevent this, avoid using the `transparent` keyword, especially with CSS gradients. To solve the issue, it's recommend to use the following:

```
.element {
    background: linear-gradient(to top, #fff, rgba(0, 0, 0, 0));
}
```

## Not Taking Advantage of the Cascade

By default, the `color` property is inherited by child elements such as `p` and `span`. Instead of setting the `color` property on each element, add it to the `body` element, and then all `p` and `span` elements will inherit that color, unless you override it.

```
body {
    color: #222;
    /* All elements will inherit this color. */
}
```

However, the `a` element doesn't inherit `color` by default. You can override its color or use the `inherit` keyword.

```
a {
    color: #222; /* … or… */
    color: inherit;
}
```

I consider a developer not taking advantage of the cascade to be a bug because it's so important. Why add more CSS than you need to?

## Forgetting the Hash Notation

The hash notation that comes before a color's hex value is important. I'll often copy and paste a color from a design app such as Adobe Experience Design (XD) or Sketch. When coping from Sketch, the color is copied like `275ED5`, whereas Adobe XD adds the hash: `#275ED5`. This difference can lead to unexpected results if you are not 100% focused.

```
a {
    color: 275ed5; /* Forgetting the hash */
    color: ##275ed5; /* Doubling the hash */
}
```

Notice the hash incorrectly doubled in the second rule. While working on a project, you might paste the color value with the hash, and then, after editing the color in another app, you might double-click on the value (including the hash) and blindly paste it back in the CSS, leading to a double hash.


Avoiding such an issue is possible with a style linter, of course. However, it's important to train ourselves to keep an eye when copying and pasting things into a code editor.

## CSS Backgrounds

Backgrounds in CSS are commonly used to add a background color, to add an image, or for decoration. Let's explore some issues with them.

### The Order of the Background's Size and Position

In the shorthand of the `background` property, writing out the background size and position can be confusing. They have a certain order, separated by a slash. If the order is missed, the `background`'s entire definition will become invalid.

According to Mozilla Developer Network (MDN):

> The `<bg-size>` value may only be included immediately after `<position>`, separated with the '/' character, like this: "center/80%".

```
background: url("image.png") center/50px 80px no-repeat;
```

Notice the `center/50px 80px`. The first one is for `background-size`, and the second is for `position`. The order can't be reversed. Spaces around the slash are fine.

## Don't Use the Shorthand to Set a Color Only

It might be tempting to use the shorthand of `background` to add a background color, but it's not advisable because this will reset all other background-related properties with it.

## Dynamic Background

If the background is being set with JavaScript, use the dedicated properties for `background-size`, `position`, and `repeat`. The `background-image` is the only property that needs to be set dynamically with JavaScript. If you set the whole `background` with JavaScript, that will be a lot of unnecessary work.

## Forgetting About `background-repeat`

When setting a background, we can easily forget about `background-repeat`. For instance, the background of a section might look good on a 15-inch laptop, but it might repeat on a 27-inch desktop. Remember to specify whether the background should repeat.

```
.element {
    background: url("image.png") cover/center no-repeat;
}
```

Generally speaking, I recommend to combine both the longhand and shorthand properties. See below:

```
.element {
    background: url("image.png") center no-repeat;
    background-size: cover;
}
```

### Printing CSS and Backgrounds

CSS backgrounds are not included in print by default. We can override that behaviour and force backgrounds to be included in print by using the following CSS property:

```
.element {
    background: url('cheesecake.png') center/cover no-repeat;
    -webkit-print-color-adjust: exact; /* Forces the browser to render
the background in print mode */
}
```

## CSS Selectors

Targeting and styling HTML elements is a core skill of web developers. If we don't learn how to properly use CSS selectors, we will run into bugs.

## Forgetting the Dot Notation for Classes

Selecting an element by class name won't work without the dot notation. This often happens when we are not focused.

```
button-primary {
    /* The styles won't work. */
}
```

## Grouping Selectors

Here is an interesting bug that you might not think about it. Grouping a valid and invalid selector together can lead to the whole declaration being ignored.

```
a, ..button-primary { }
```

According to the CSS specification:

> If just one of these selectors were invalid, the entire selector list would be invalid.

The `..button-primary` class has two dot notations, which makes it invalid. Grouping it with the `a` element would make the browser ignore the entire declaration.

This mistake is easily made when selecting the `::selection` pseudo-element (to target selected text) or the `::placeholder` pseudo-element (to target input placeholders). We also see it in vendor-prefixed selectors, used for cross-browser support; when one vendor-prefixed selector of a group is incorrect, the whole style declaration will be ignored.

## Calling a CSS Selector More Than Once

A common mistake with CSS specificity is calling a selector more than once.

```
.title { /* Some styles */ }

/* 300 lines and.. */

.title { /* Another style */ }
```

This alone is not a bug, but it easily opens the way for bugs. Avoid this pattern, and use a style linter that warns about such things.

## Customizing an Input's Placeholder

Firefox makes placeholder text of input elements semi-transparent. When setting a custom color for placeholder text, keep in mind that it will appear a bit dimmed. This is not good for accessibility. Make sure to fix that by resetting the semi-transparency:

```
::-webkit-input-placeholder { color: #222; opacity: 1 }
::-moz-placeholder { color: #222; opacity: 1 }
:-ms-input-placeholder { color: #222; opacity: 1 }
```

## The Order of User-Action Pseudo-Classes

The order of the `:visited`, `:focus`, `:hover`, and `:active` pseudo-classes is important. If they don't appear as follows, then they won't work as expected:

```
a:visited { color: pink; }
a:focus { outline: solid 1px dotted; }
a:hover { background: grey; }
a:active { background: darkgrey; }
```

## Targeting an Element With More Than One Class

A common mistake that I see among beginners is incorrectly targeting two classes at the same time in order to select an element. Consider the following:

```
<div class="alert success"></div>
<div class="alert danger"></div>
```

```
.alert.success { background-color: green; }
.alert.danger { background-color: red; }
```

The first style will work with elements that have both `.alert` and `.success` classes, while the second one will work only with elements that have both `.alert` and `.danger` classes. However, suppose we did the following:

```
.alert .success { background-color: green; }
```

The mistake here is **adding a space** between the two classes. This space changes the whole thing, and it won't work. We are basically selecting an element with a `.success` class inside an element with an `.alert` class. It assumes an HTML structure like the following:

```
<div class="alert">
    <div class="success"></div>
</div>
```

Use the correct selector, or else you could waste a lot of time wondering why it's not working.

## Targeting Classes on Particular Elements

Accessibility of a website to all users is a core principle of website design. Neglecting it can lead to bad results. A common problem we see is using a `div` element for a button and making it clickable only with JavaScript.

One way to prevent developers who you are working with from adding a class to any element and calling it a button is by targeting classes together with their elements.

```
button.btn { }
```

This way, the `.btn` class won't work on any element except the `button`. It's a good way to restrict usage of the class to actual `button` elements.

## An Alternative to `!important`

Sometimes, a style won't work because it's being overridden by another style in the CSS file. Using `!important` is not recommended. There is a better way, with CSS classes only.

```
.btn.btn { }
```

Calling a class twice increases the **specificity** of a selector, thus making the rule work without `!important`. Make sure there is no space between the classes. Note that you can call it three, four, or however many times you like.

# CSS Borders

## Border on Hover

A common mistake when showing a border on hover is to add the border *only* on hover. If the border is 1 pixel, then the element will jump by that much when the user hovers over it. To avoid the jump, add the border to the normal state with a transparent color.

```
.nav-item { border: 2px solid rgba(0, 0, 0, 0); }

.nav-item:hover { border-color: #222; }
```

This way, the border has already been added and is reserving space, and the appearance of the border on hover will be based on `border-color`.

We see this often with inline navigation menus, where items should have a border on hover. Notice in the figure how the elements are slightly pushed to the right once the navigation item is hovered over.

## Multiple Borders

When you add more than one CSS border to an element — for example, borders on the left and bottom edges — you might notice that the point where the two borders meet is kind of weird. The bottom end of the left border and the left end of the bottom border will look like cut-off triangles.

This is normal and expected. CSS borders work like that. If you want multiple borders, then you could combine a border with a shadow to fix the issue. The left border can be kept as it is, and the bottom one can be a shadow.

## Border and `currentColor` keyword

This is not necessarily a bug, but worth mentioning. The `currentColor` keyword is the default value of `border-color`.

```
.element {
    color: #222;
    border: 2px solid;
}
```

Notice that a color isn't declared in the `border` rule. The default value is `currentColor`, which inherits its value from the `color` property. Our example could be written out as follows:

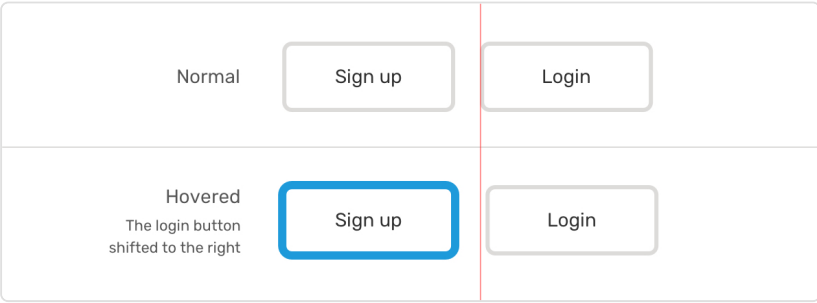
```
.element {
    color: #222;
    border: 2px solid currentColor;
}
```

The point is that adding a color to the `border` rule is **not necessary** when it's

the same value as `color`.

## Border Transition on Hover

There are many ways to transition a border with CSS. One common way is to modify `border-width`. Suppose we have two buttons:




We want to expand the border of the first one, so we use `border-width`. Hovering over the button will shift the position of the other button because of the expanded border width. There are two main problems with this approach:

- The transition is slow. That is, the browser will not smoothly animate the width. Instead of increasing it like `1, 1.1, 1.2 … 3`, it will increase like `1, 2, 3`. This is stepped animation.
- It's also bad for performance. A change to `border-width` will trigger a repaint of the layout in the browser. The sibling button will move around because of the new border width. With each frame of the animation, the browser will repaint their positions.

The preferable solution is to use `box-shadow`. A shadow is much easier to transition, and performance is good enough.

Suppose we want to animate the bottom width of an element's border from 3 to 6 pixels. To do that, we can use `box-shadow`, working with its `y` value as an alternative to `border-width`.

```
:root {
    --shadow-y: 3px;
}

.element {
    box-shadow: 0 var(--shadow-y) 0 0 #222;
    transition: box-shadow 0.3s ease-out;
}

.element:hover {
    --shadow-y: 6px;
}
```

Going further, I defined a CSS variable to hold the `y` value of the shadow, and I changed that on hover. Using a CSS variable, instead of copying the whole `box-shadow` rule again, reduces redundancy in the code.

## Changing a Border's Width Based on Screen Size

A `border-width` that works for laptop or desktop screens might be too big for mobile. Usually, we would use a media query to change `border-width` at a certain screen size. While that works, with the CSS tools available today, we have better alternatives.

The border width is growing based on the screen width

By using CSS' comparison function, we can create a shadow that respond to screen size without having to use a media query.

```
.element {
    border: min(1vw, 10px) solid #468eef;
}
```

Thus, `border-width`'s maximum value will be 10 pixels, and it will get smaller as the screen narrows.

## Adding a Border to Text Content

When I started learning CSS, I thought it was possible to add a border to text. It's not. This might trip up anyone who is new to CSS. However, it is doable with the `text-stroke` or `text-shadow` property. Let's explore both solutions.

CSS is awesome

**CSS is awesome**

The most common solution is to set `color` to `transparent` and then add the border.

```
.element {
    color: transparent;
  -webkit-text-stroke: 1px #000;
}
```

While this works, the text will be inaccessible in unsupported browsers, such as Internet Explorer and old versions of Chrome, Firefox, and Safari.

We can use CSS' `@supports` query to detect whether `-webkit-text-stroke` is supported and, if so, use it.

```
@supports (-webkit-text-stroke: 1px black) {
  .element {
      color: transparent;
    -webkit-text-stroke: 1px #222;
  }
}
```

Also, we can replace `color: transparent` with something else.

```
@supports (-webkit-text-stroke: 1px black) {
  .element {
      -webkit-text-fill-color: #fff;
    -webkit-text-stroke: 1px #222;
  }
}
```

### `border: none` vs. `border: 0`

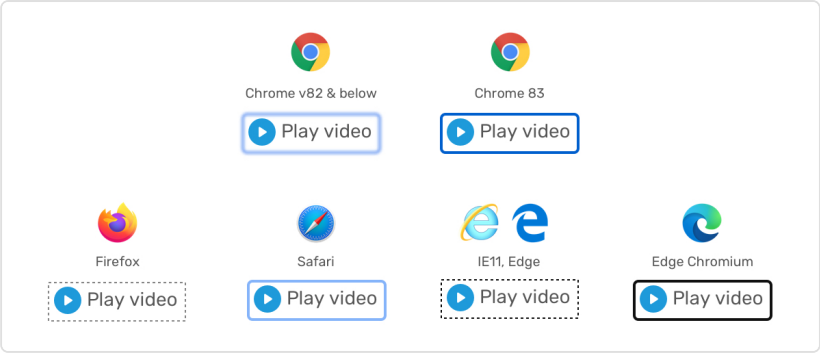
Both `border: none` and `border: 0` will reset the border to its initial state. It resets `border-width` to `0px` and `border-style` to `none`, and `border-color` will inherit its value from the `color` property.

I would prefer to use `border: 0`. However, if we look at another property such as `box-shadow`, resetting it to `box-shadow: 0` is invalid. This is confusing because you would expect that `0` would reset both of those properties. The `none` keyword works with both, though. I recommend using it instead of `0`.

## Focus Outline

This is not directly related to the `border` property, but it's easy to confuse `border` and `outline`. For example, a quick search on StackOverflow for "css border" returns a couple of questions whose titles contain "focus border, blue border". So, I decided to cover it here.

The blue border or outline that appears when an element is focused is not a bug, but rather a feature that helps keyboard users to know where they are, to take action, and more. Its implementation is not consistent across browsers.




Instead of removing that outline, we can override it with a custom one.

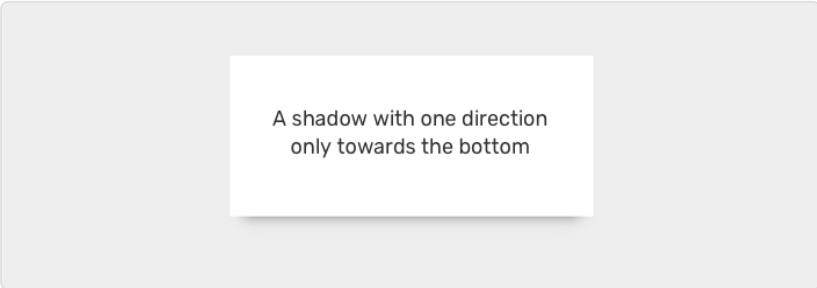
```
.nav-item a:focus {
    outline: dotted 2px blue;
}
```

The possibilities are endless. But please don't remove that outline under any circumstances, because it will affect the accessibility of the website.

## Box Shadow

### A Shadow on One Side of an Element

When you add a shadow in CSS, it will spread out from the four sides of the element by default. I've seen a common request in my research for how to add a shadow in one direction.

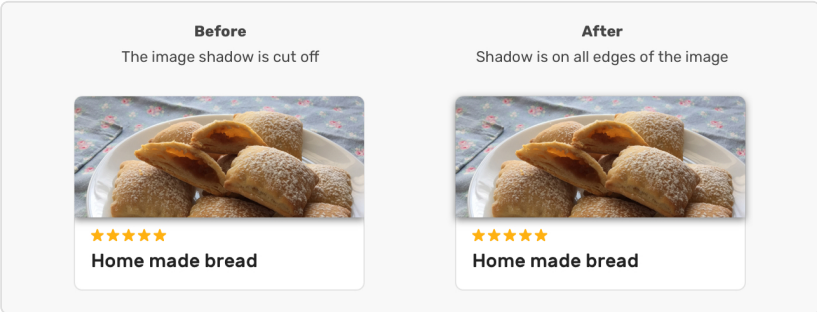



With `box-shadow`, the spread value controls the size of the shadow's area of coverage. By using a negative value, we can add a shadow to one direction of the element.

```
.element {
    /* The value of -5px is the spread of the shadow. */
    box-shadow: 0 7px 7px -5px #ccc;
}
```

## `box-shadow` and `overflow: hidden` Don't Mix Well

If you need to use `overflow` for an element, and some of its children have a `box-shadow` property, the shadow will be cut off on the left and right sides.




In this example, the thumbnail's shadow is cut off on the left and right sides. The reason is that `overflow: hidden` is being applied to the parent element (the card). Avoid using `overflow: hidden` when you want a shadow to be visible on child elements.

## Multiple Box Shadows

Sometimes we need to add multiple shadows to an element. This is supported and can be done without additional HTML elements or pseudo-elements. Each shadow would be separated by a comma.

```
.element {
    box-shadow: 0 5px 10px 0 #ccc, 0 10px 20px #222;
}
```

## White-Space Issue With Box Shadow and Inline Image

Do you remember when we talked about the `display` of an image and how an image has a little white space below it? The reason is that it's an inline element. That also happens when we use `box-shadow` with the parent of an inline image.

```
<div class="img-wrapper">
  <img src="ahmad.jpg" alt="">
</div>
```

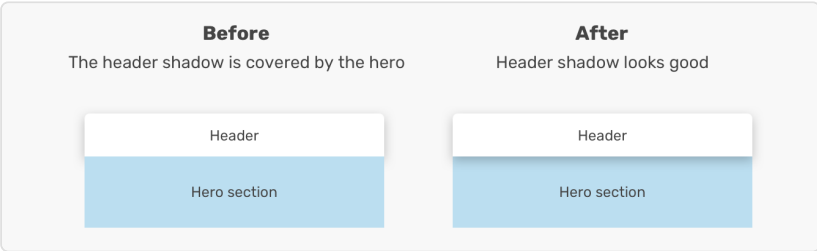
```
.img-wrapper {
  box-shadow: 0 5px 10px 0 #ccc;
}
```

In this example, there is white space below the image, which becomes visible only after the shadow is added. Make sure to reset the `display` value of the

image to avoid this issue.

## Box Shadow on Header Element

When the website's header is directly followed by, say, a hero image, then adding a shadow might be tricky. If you try to add a shadow to the header, it will be covered by the hero section.



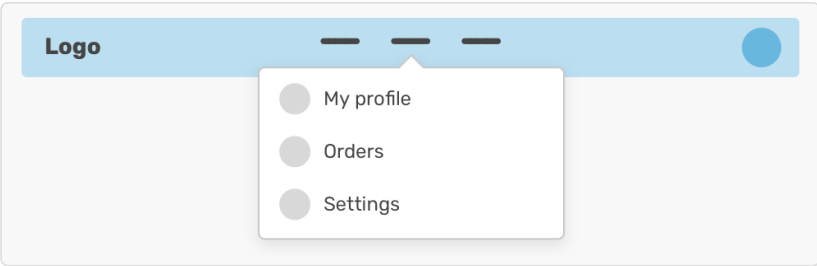

Solving this issue can be done by changing the stacking context of the header element. The easiest solution is to add the following:

```
.site-header {
    position: relative;
}
```

Make sure that this fix doesn't have unintended side effects.

## Shadow on Arrow of Speech Bubble

A common design pattern for tooltips and dropdown menus is to add an arrow that points to the parent element of the tooltip or dropdown menu. There are many ways to make an arrow in CSS, the most common being to create a pseudo-element with a border on one side.

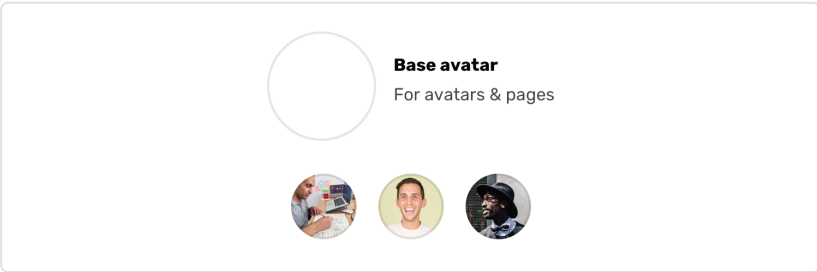



How do we add `box-shadow` to the arrow? We can simulate a shadow coming from one direction by using a negative value for the `x` and `y` values.

```
.element::before {
    content: "";
    width: 20px;
    height: 20px;
    background-color: #fff;
    position: absolute;
    left: 50%;
    top: -10px;
    transform: translateX(-50%) rotate(45deg);
    box-shadow: -1px -1px 1px 0 lightgray;
}
```

## An `inset` Shadow on Image Elements

Suppose we have an image, and we want to add a translucent inner border to it that acts as a fallback, in case the image fails to load.




The border would also be useful in preventing bright images from blending into light backgrounds. The first solution you might think of is to use an inset `box-shadow`:

```
img {
    box-shadow: inset 0 0 0 2px rgba(0, 0, 0, 0.2);
}
```

Unfortunately, an inset `box-shadow` doesn't work with images. We need a workaround. I learned a few solutions while analyzing facebook.com's new design, which we'll explore below.

### Using an Additional HTML Element for the Border

With an additional element, we would keep its background transparent and add a border only. The following shows the HTML's structure and the CSS:

```
<div class="avatar-wrapper">
    <img class="avatar" width="40" height="40" src="avatar.jpg"
width="40" alt="">
    <div class="avatar-outline"></div>
</div>
```

```
.avatar-wrapper {
    position: relative;
}

.avatar {
    display: block;
    border-radius: 50%;
}

.avatar-outline {
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    box-shadow: inset 0 0 0 1px rgba(0, 0, 0, 0.1);
    border-radius: 50%;
}
```

### Using an SVG `image`

Another interesting solution is to use an `svg` element, instead of an `img`. This solution is well supported in browsers and is much easier to control. Here is the HTML:

```
<svg role="none" style="height: 100px; width: 100px;">
  <mask id="circle">
    <circle cx="50" cy="50" fill="white" r="50"></circle>
  </mask>
  <g mask="url(#circle)">
    <image x="0" y="0" height="100%" preserveAspectRatio="xMidYMid
slice" width="100%" xlink:href="shadeed.jpg" style="height: 100px;
width: 100px;"></image>
    <circle class="border" cx="50" cy="50" r="50"></circle>
  </g>
</svg>
```

Let's go over the SVG code:

1. We have a `mask` element as a circle.
2. A group follows, containing the image itself and a `circle` element. The `circle` element acts as a border, and it will be above the `image`.

```
.border {
  stroke-width: 2;
  stroke: rgba(0, 0, 0, 0.1);
  fill: none;
}
```

Facebook uses both solutions in its 2020 redesign. The SVG one is used rarely for things like profile pictures and user avatars in the sidebar. The solution with the additional HTML element is used a lot in the social feed, in comments, and more.

## CSS Transforms

### Applying Multiple Transforms

In CSS transforms, we can use one or more transforms on an element. For instance, it's possible to move an element 10 pixels to the right and then rotate it.

```
.element {
    transform: translateX(10px) rotate(20deg);
}
```

Occasionally, you might need to use multiple transforms on an element.

However, sometimes you'll need one transform on mobile and two on desktop. Here, we run into a common issue.

Let's say we have a modal that should be centered horizontally on mobile. In larger viewports, it should be centered both horizontally and vertically. A common mistake is to accidentally reset the transform.

```
.modal-body {
    transform: translateX(-50%);
}

@media (min-width: 800px) {
    .modal-body {
            transform: translate(0, -50%);
    }
}
```

The transform gets an unintended reset with `translate(0, -50%)`. This can easily cause confusion. The solution is straightforward: We need to keep the `translateX(-50%)`.

```
@media (min-width: 800px) {
    .modal-body {
            transform: translate(-50%, -50%);
    }
}
```

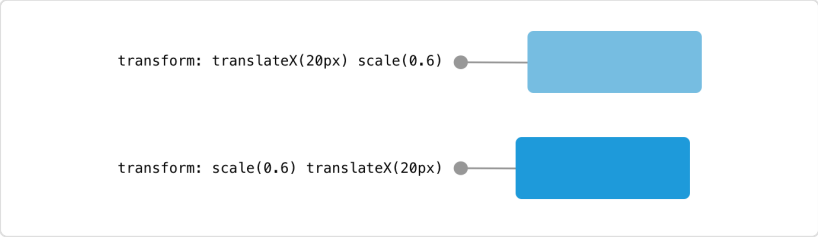
## The Order of CSS Transforms Matters

According to MDN:

> The transform functions are multiplied in order from left to right, meaning that composite transforms are effectively applied in order from

> right to left.

The order of transform functions is important. Keep an eye on them to avoid issues.




Notice how the order of the transform functions affects the visual position of each rectangle. In the first one, the element has been scaled first, then transformed 20 pixels to the right. The opposite happens with the second rectangle. When debugging CSS transforms, make sure the order meets your needs. Don't add transform functions randomly.

## Overriding a Transform by Mistake

When I started to learn CSS, I wasn't totally aware that the `transform` CSS property can include multiple transforms, and that we need to specify all of the transforms we want every time we declare the property. The following is a bug:

```
.element {
    transform: translateY(100px);
}

.element:hover {
    transform: rotate(45deg);
}
```

You might expect that both the `translate` and `rotate` functions will work, but that's not so. The second transform will override the first one; thus, we'll lose `translateY`. Instead, we would combine them:

```
.element:hover {
    transform: translateY(100px) rotate(45deg);
}
```

And keep in mind the importance of order, as explained earlier.

## Individual Transform Properties

At the time of writing, only Firefox 72 supports individually declared transforms. That would solve the issue mentioned just above really well, because we wouldn't have to combine all transforms together. Here is the reworked version of the previous example:

```
.element {
    translate: 0 100px;
}

.element:hover {
    rotate: 45deg;
}
```

Isn't that simpler? You can detect whether this is supported with the `@supports` media query.

```
@supports (translate: 10px 10px) {
    /* Add the individual transform properties */
}
```

## Transforming SVG Elements

The coordinate system for HTML elements starts at `50% 50%`. In SVG, it's completely different; it starts at `0 0`. Because of this difference, using CSS transforms with SVG elements can be confusing.

To transform an SVG element as expected, use pixel values and avoid percentages. And keep in mind that CSS transforms aren't supported in Internet Explorer but are supported as of Microsoft Edge 17.

Taken from Ana Tudor on CSS-Tricks, the following example illustrates the issue:

```
rect {
  /* This doesn't work in Internet Explorer and old versions of Edge.
*/
  transform: translate(295px, 115px);
}
```

```
<!-- This works everywhere. -->
<rect width='150' height='80' transform='translate(295 115)' />
```

We can use the inline `transform` attribute for an SVG child element. It's a bit different with the CSS transform, with no comma between the values.

## Using Transforms to Rotate Text by 90 Degrees

I wouldn't consider this a bug, but rather a question of finding a better way to solve this need. Say we want to rotate a text element.


The first approach you might consider is positioning the text and rotating it. While this would work, there is a better solution. By using CSS' `writing-mode`, we can easily change the writing direction from left-to-right to top-to-bottom. The property sets the direction (horizontal or vertical) of a text element. It was intended for languages such as Japanese and Chinese.

```
/* Without writing-mode */
.title {
    position: absolute;
    left: 40px;
    transform-origin: left top;
    transform: rotate(90deg);
}

/* With writing-mode */
.title { writing-mode: vertical-lr; }
```

With `writing-mode`, we can rotate the title with one line of CSS. Browser support is great, too.

## CSS Custom Properties (Variables)

### Scoped vs. Global Variables

A scoped variable is one that can only be used inside an element, whereas a global one, as evident by its name, can be used globally.

```
<div class="header">
  <div class="item"></div>
</div>
```

```
.header {
    --brand-color: #222;
}
```

We define a variable, `--brand-color`, which will only work with the `.header` element and its child items. An element with a class of `.item` can see the variable.

While researching this topic, I noticed a question that is marked as correct on StackOverflow, but is not actually correct. The answer claims that the following should work:

```
.header {
    --brand-color: #222;
}

body {
    background-color: var(--brand-color);
}
```

This will never work. The `body` element can't see the CSS variable because its scoped to the `.header` element. For this to work, the CSS variable must be defined **globally**:

```
:root {
    --brand-color: #222;
}

body {
    background-color: var(--brand-color);
}
```

This works perfectly.

## Setting a Fallback for a Variable

Sometimes when talking about fallbacks for variables, it might be confusing whether we're referring to a fallback for an old browser that doesn't support CSS variables or a fallback for the CSS variable itself.

```
.title {
    color: #222;
    color: var(--brand-color);
}
```

The first rule above is a fallback for old browsers, which can be automated using a tool such as PostCSS.

However, our focus here is on a fallback for the CSS variable itself:

```
.title {
    color: var(--brand-color, #222);
}
```

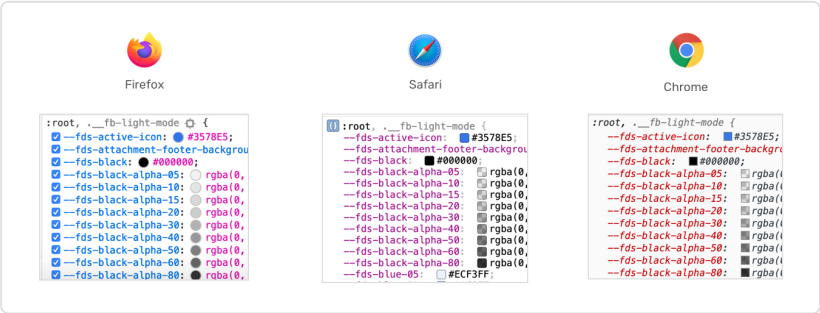
If, for some reason, the variable `--brand-color` is not available, then the value after the comma will be used instead. Note that you can use more than one fallback value. See below:

```
.title {
    color: var(--brand-color, var(--secondary-color, #222));
}
```

## Retrieving All CSS Variables Defined in a Document

Sometimes, you might want to see all of the global CSS variables in your application or website. Thankfully, we can get them from the browser's DevTools.

Select the `html` element, and on the right side, you should see all CSS variables defined within it.



The figure highlights how CSS variables look when we inspect the root element of the page (the `html` element). What I like about Firefox is that you can toggle a variable! This can be very useful for debugging or testing the fallback value of a CSS variable.

## Invalidation at Computed-Value Time

A declaration will be invalid at the computed-value time if it uses a valid custom property but if the property value, after substituting the `var()` functions, is invalid. When this happens, the property computes to its `initial` value. Consider the following example, taken from Lea Verou's blog:

```
#toc {
    position: fixed;
    top: 11em;
    top: max(0em, 11rem - var(--scrolltop) * 1px);
}
```

If the browser doesn't support the `max()` comparison function, it will make the property invalid at the computed-value time, which will compute to `initial`; and for the `top` property, the initial value will be `0`. This will break the design. The fix for this is to use the `@supports` function to detect support for the `max()` function. If it's supported, then the declaration will be used.

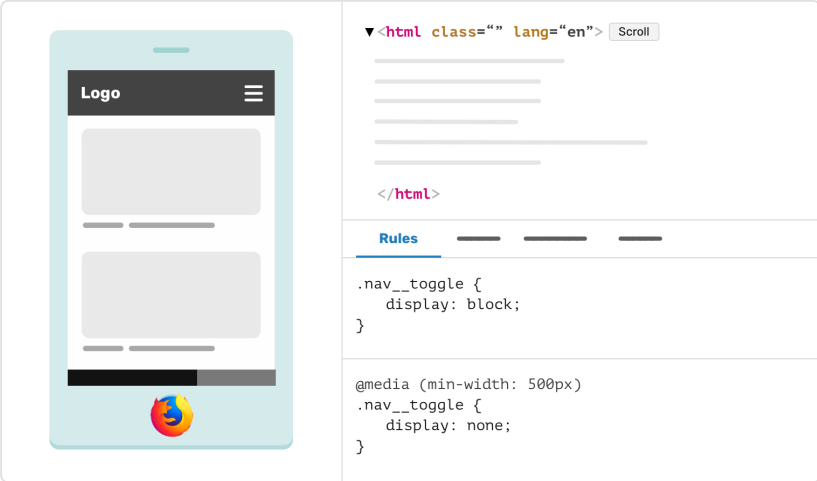
```
#toc {
    position: fixed;
    top: 11em;
}

@supports (top: max(1em, 1px)) {
    #toc {
        top: max(0em, 11rem - var(--scrolltop) * 1px);
    }
}
```

## Horizontal Scrolling

This is one of the most common issues in front-end development. Horizontal scrolling is an indication that an element is positioned outside the viewport's boundaries or that an element is wider than the viewport. The reasons vary. I will try to summarize them here, along with strategies you can use to find and fix the problem.

### Firefox Shows a `scroll` Label

A little assistance worth highlighting is that Firefox shows a "scroll" label for elements that are wider than the viewport. The label will guide you to debug the element that is causing the horizontal scrolling.




When you click on the `scroll` label, Firefox will highlight the element that is causing the horizontal scrolling.

```
▼<html class="" lang="en">
  <body class="" lang="en">
    ▼<section class="section s1">  Overflow
      <h2 class="title"></h2>
      <p class="desc"></p>
    </section>
  </body>
</html>
```

The `h2` and `p` elements are causing the horizontal scrolling because they are wider than the viewport. As a result, Firefox highlights them when the "scroll"

label is clicked.

## Finding Horizontal Scrolling Bugs

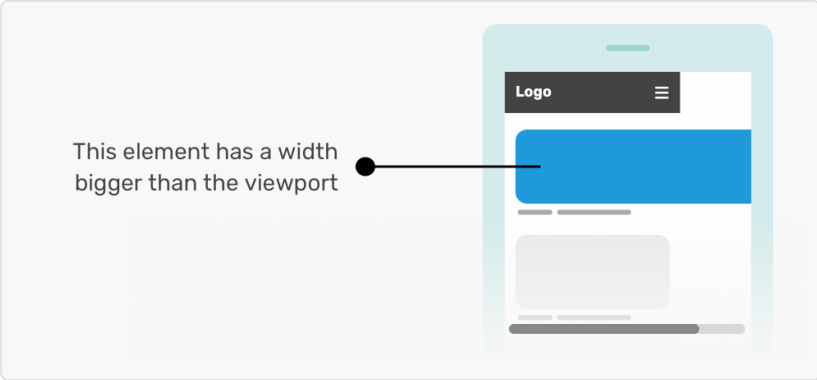
Let's focus first on how to find horizontal scrolling problems. The first thing to do is to make sure that the scrollbars are shown by default. macOS, for example, doesn't show the scrollbars until you start scrolling (either vertically or horizontally). Making the scrollbars visible can help us to spot scrolling issues much more quickly.

Go to "System Preferences" > "General" > "Show scroll bars" > "Always."

Windows shows the scrollbars by default, so there is no need to do anything there.

### Scrolling to the Left or Right

On the page you want to test, try scrolling to the left or right with your mouse or trackpad. Keep narrowing the screen, and repeat the process. If there is no scrolling, then activate mobile mode in the DevTools. A horizontal scrolling issue in mobile mode might look like the following:




This means there is an element wider than the `body` or `html` element.

### Using JavaScript to Get Elements Wider Than the Body

We can take it further and use a script to detect whether an element is wider than the `body` or `html` element. This can be useful in a large project or one you're new to.

```
[].forEach.call(document.querySelectorAll("body *"), function (el) {
  if (el.offsetWidth > document.body.offsetWidth) {
    console.log(el.className);
  }
});
```

Here, we've selected all elements inside the `body`, checked whether an element is wider than it, and printed it out.

### Using `outline`

By using CSS' `outline` property, we can add an outline around every element

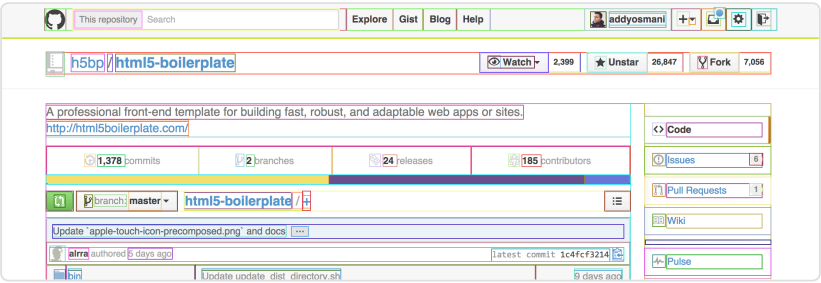
in the layout. That can be a great help in revealing any issues. For example, it can reveal whether any elements are taking up more space than allowed for them.

```
*,*:before, *:after {
    outline: solid 1px;
}
```

This works perfectly, but we can take it to the next level with a script created by Addy Osmani:

```
[].forEach.call($$("*"),function(a){a.style.outline="1px solid
#"+(~~(Math.random()*(1<<24))).toString(16)})
```

This script will add an outline to every element on the page, with a different color for each one. (Having all outlines in the same color would get a bit confusing in a complex layout.)




Note that using `outline` is much better than `border`, for a few reasons:

- An `outline` will be added after a border, in case an element has one. In other words, the `outline` won't take up space because it is drawn outside

of the element's content.

- Using a `border` might break some design components, in case `box-sizing` is not set to `border-box` or if an element already has a border. It would get confusing.
- An `outline` won't be affected by an element's `border-radius`. All outlines added to the page will be rectangular.

## Fixing Horizontal Scrolling

Now that we've identified horizontal scrolling issues, it's time to learn how to debug them. When you find horizontal scrolling, you might not see the cause of the issue at first glance, so you need to experiment.

Open up the browser's DevTools and start deleting the main HTML elements one by one, to see whether the scrolling disappears (Hint: you can use `CMD + z` for macOS or `CTRL + z` or Windows to cancel the deletion of an element). Once you see that the scrolling is gone, note the element that you just deleted. Refresh the page, and dig into that element to see what's there. There could be a few reasons for the horizontal scrolling. Let's explore them.

### A Fixed Width

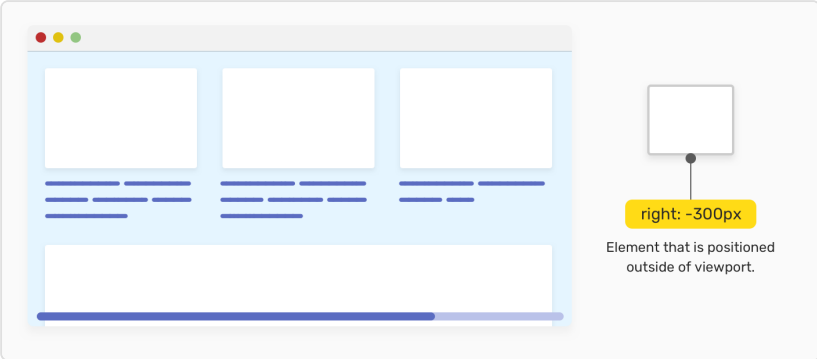
A fixed width will definitely cause horizontal scrolling. For example, the following will cause a bug when the viewport is narrower than 1000 pixels.

```
.section {
    width: 1000px;
}
```

To fix this, we need to set a maximum width for the element using `max-width`:

```
.section {
    width: 1000px;
    max-width: 100%; /* Prevent the element from getting wider than
1000 pixels when the viewport is small. */
}
```

### A Positioned Element With a Negative Value




An element for which one of the `position` properties (`top`, `right`, `bottom`, `left`) is set to a negative value will cause a horizontal scrolling.

```
.element {
    position: absolute;
    right: -100px;
}
```

The same thing can happen when you use a CSS transform to move an element out of the viewport.

```
.element {
    position: absolute;
    right: 0;
    transform: translateX(1500px);
}
```

If it's necessary to place an element outside its parent, then it's better to use the following:

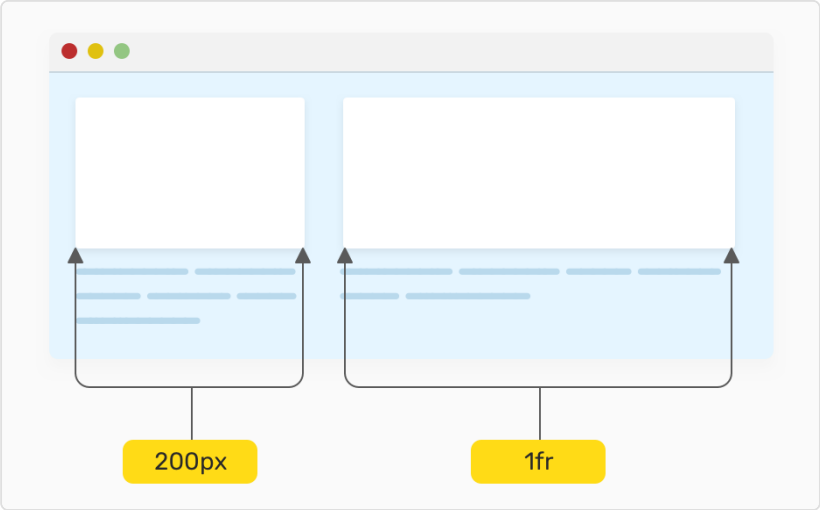
- apply a CSS `transform`
- use CSS `overflow: hidden` on the parent, in case you don't have another choice

### A Flexbox Wrapper Without Wrapping

When using flexbox, the row won't wrap by default. When the viewport gets small, horizontal scrolling will happen because not enough space is available to show all of the elements on one line. This is a common issue with flexbox. To solve it, you need to force wrapping on certain screen sizes.

```
.section {
    display: flex;
    flex-wrap: wrap; /* Force flex items onto a new line in case not
enough space is available. */
}
```

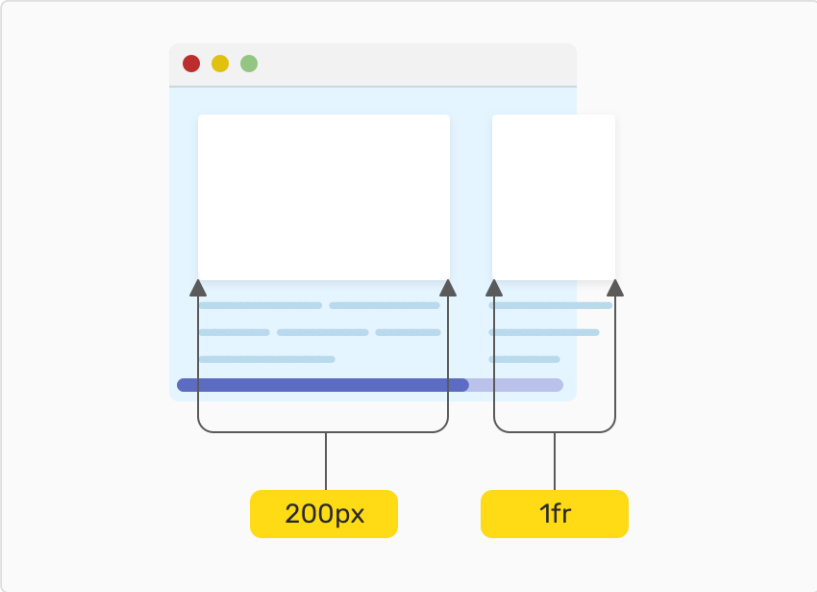
### A Grid Wrapper With minmax()




When using CSS grid, there is a possibility of horizontal scrolling. Say we have a grid with columns that are dynamic and that have a minimum width of 200 pixels.

```
.wrapper {
    display: grid;
    grid-template-columns: 200px 1fr;
    grid-gap: 16px;
}
```

Everything looks good until the viewport gets narrower. The space isn't enough, and as a result, horizontal scrolling occurs.




To fix this, we can apply the grid only when space is enough, using a media query.

```
@media (min-width: 400px) {
.wrapper { /* The grid goes here. */ }
}
```

### A Long Word or Inline Link

If an article has a very long word or link, it can easily cause horizontal overflow if it's not handled properly.


As you see, the long word causes horizontal scrolling. The solution is to use the `overflow-wrap` CSS property. It prevents a long word from overflowing its line box.

```
.content p {
    overflow-wrap: break-word;
}
```

It's worth mentioning that the property has been renamed from `word-wrap` to `overflow-wrap`.

### An Image Without `max-width: 100%`

If for any reason you're not using a CSS reset file, then you need to make sure that any image on the website doesn't exceed its parent's width. To do this, all you need is the following:

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

You guessed it — forgetting to include that line will cause horizontal scrolling.

## Transition

CSS transitions enable us to animate an element smoothly from one state to another. Let's explore some common issues with them.

### Transition on Resize

An annoying problem with transitions is seeing elements move and animate when the browser window is resized. This is because you've applied the transition to all properties, which breaks the behavior and might even cause performance issues.

```
.element {
    transition: all 0.2s ease-out;
}
```

The `all` keyword tells us that the transition will be applied to all properties of the element. This might be OK for one element, but using such a pattern at scale is not recommended. When I started learning about CSS, I got used to making the following mistake:

```
* {
    transition: all 0.2s ease-out
}
```

This bit of CSS adds a transition to every element on the page. Don't do this, please! It's not a good idea.

## Transitioning Height

A common need is to transition an element's height — for example, from `0` to `auto`. The `auto` value would make the height of the element equal the content within.

```
.element {
    height: 0;
    transition: height 0.2s ease-out;
}

.element:hover {
    height: auto;
}
```

Unfortunately, this is not possible in CSS. You can't transition to `auto`. However, there is a workaround. Instead of using `height`, we can use a `max-height` value that is great than the content's height. For example, if we have a mobile menu with a height of 200 pixels, then the value of `max-height` should be at least 300 pixels. The reason for the greater value is to make sure that the height of the element never gets to that point.

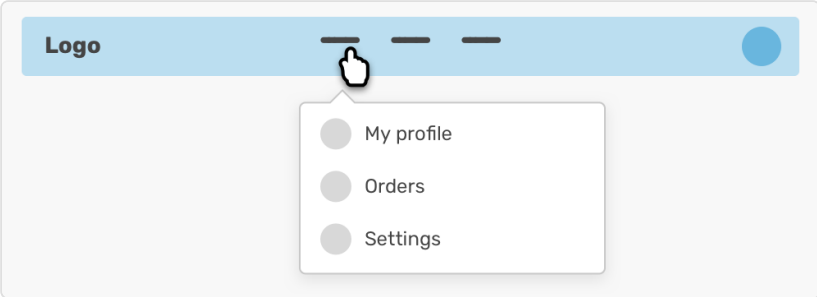
```
.element {
    max-height: 0;
    overflow: hidden;
    transition: max-height 0.2s ease-out;
}

.element:hover {
    max-height: 300px;
}
```

I added `overflow: hidden` to clip any content that might be visible when `max-`

`height: 0` is set on the element.

## Transitioning Visibility and Display

Transitioning the `display` property of an element is not possible. However, we can combine the `visibility` and `opacity` properties to mimic hiding an element in an accessible way.




Here we have a menu that should be shown on mouse hover and keyboard focus. If we used only `opacity` to hide it, then the menu would still be there and its links clickable (though invisible). This behavior will inevitably lead to confusion. A better solution would be to use something like the following:

```
.menu {
    opacity: 0;
    visibility: hidden;
    transition: opacity 0.3s ease-out, visibility 0.3s ease-out;
}

.menu-wrapper:hover .menu {
    opacity: 1;
    visibility: visible;
}
```

CSS' `visibility` property is animatable. When added in the `transition` group, it will be animated, and the menu will fade in and out nicely, without suddenly disappearing.

## Overflow

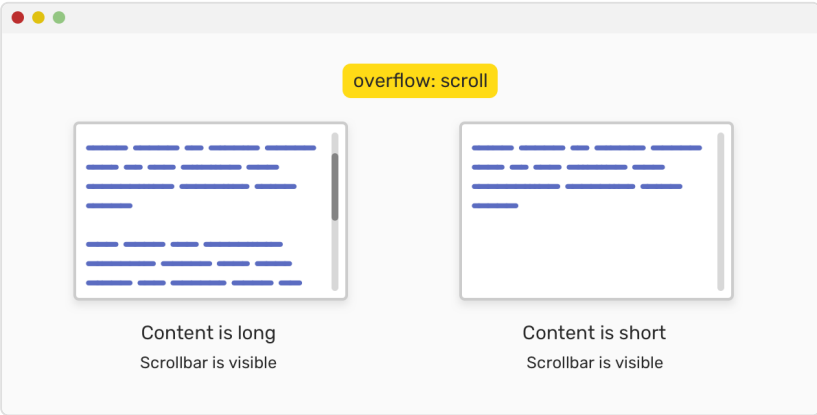
The value of the `overflow` property is `visible` by default. Other values are `hidden`, `scroll`, and `auto`.

### `overflow-y: auto` vs. `overflow-y: scroll`

When we have a component with a fixed height and scrollable content, using `overflow-y: scroll` is tempting. The downside is that when the content is too short, a scrollbar will be visible on the Windows operating system. For macOS, the scrollbar is hidden by default.

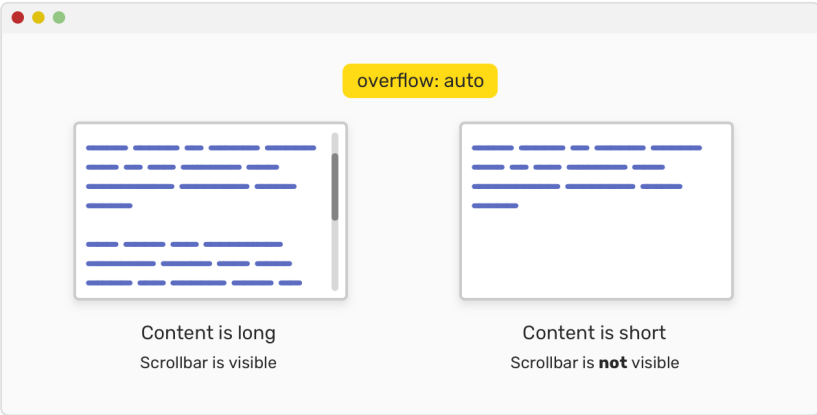
```
.section {
    overflow-y: scroll;
}
```

## 4. CSS Properties That Commonly Lead to Bugs




To fix this and show the scrollbar only when the content goes long, use `auto` instead.

```
.section {
    overflow-y: auto;
}
```




## Scrolling on Mobile

When we have, say, a slider, it's not enough to add `overflow-x` and call it a day. In Chrome on iOS, we need to keep scrolling and moving the content manually. Luckily, there is a property that enhances the scrolling experience.

```
.wrapper {
    overflow-x: auto;
    -webkit-overflow-scrolling: touch;
}
```

This is called **momentum-based scrolling**. MDN describes it thus:

> where the content continues to scroll for a while after finishing the scroll gesture and removing your finger from the touchscreen.

As an advice, make sure to avoid using `-webkit-overflow-scrolling: touch` for a big scrolling context (e.g. a full page layout) as this might cause some

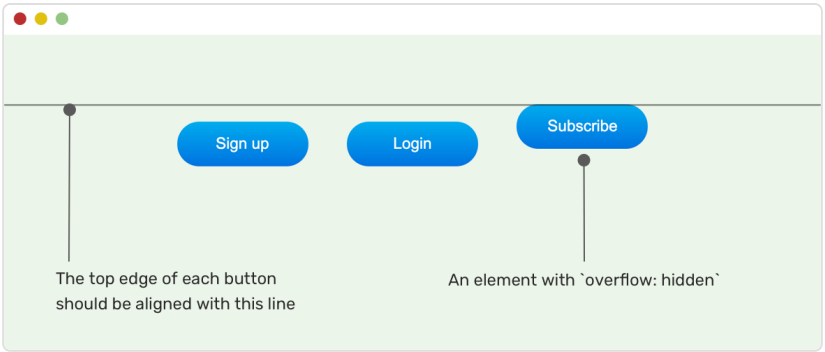
random bugs on Safari iOS.

## Inline-Block Elements With `overflow: hidden`

According to the CSS specification:

> The baseline of an "inline-block" is the baseline of its last line box in the normal flow unless it has either no in-flow line boxes or if its "overflow" property has a computed value other than "visible", in which case the baseline is the bottom margin edge.

When an `inline-block` element has an `overflow` value other than `visible`, this will cause the bottom edge of the element to be aligned according to the text baseline of its siblings.

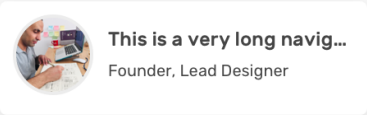



To solve this, change the alignment of the button that has `overflow: hidden`.

```
.button {
    vertical-align: top;
}
```

## Text Overflow

The `text-overflow` property sets how text with an overflow is shown. The most common value is `ellipsis`: The text will get clipped, and at the end of it will be three dots, `like this…`.




The property is sometimes confusing to use. A common hurdle is that a declaration of `text-overflow: ellipsis` does not work as you would expect.

```
span {
  text-overflow: ellipsis;
}
```

For `text-overflow` to work, the following is required:

- the element's `display` type should be set to `block`,
- the element must have the `overflow` and `white-space` properties set.

```
span {
  display: block;
  text-overflow: ellipsis;
  overflow: hidden;
  white-space: nowrap;
}
```

With these set, it will work as expected. Out of curiosity, I tested other display types, including `inline-block` and `flex`, and none of them work as expected.

## The `!important` Rule

Using the `!important` rule without good reason can cause bugs and waste your time. Why is that? Because it breaks the natural cascade of CSS. You might try to style an element and find that the style is not working. The reason could be that another element is overriding that style.

```
.element { color: #222 !important; }
.element { color: #ccc; }
```

The element's color is `#222`, even though a different color is declared a second time. In a large project, using `!important` randomly can cause a lot of confusion.

Avoid `!important` in general. Here are some things to consider before using it:

- Try to identify the source of the specificity issue with the DevTools.
- It's sometimes warranted when you're working with a third-party CSS file. You might not have any option but to override the external style.

Utility CSS classes have become more popular recently. I would consider these a good justification for `!important`.

```
<div class="d-block"></div>
```

```
.d-block { display: block !important; }
```

The `d-block` class sets the element to display as a `block` type. Adding `!important` ensures it will be applied as expected.

## Flexbox

The flexbox layout module provides us with a way to lay out a group of items either horizontally or vertically. There are many common issues with flexbox: Some are done mistakenly by the developer, and others are bugs in a browser's implementation.

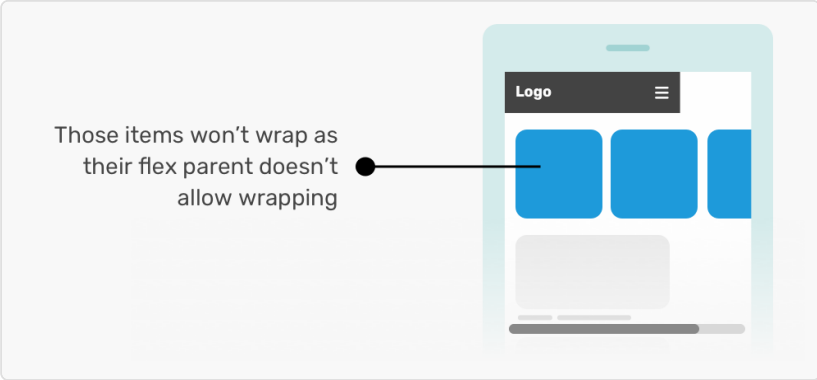
### User-Made Bugs

#### Forgetting `flex-wrap`

When setting an element as a wrapper for flexbox items, it's easy to forget about how the items should wrap. Once you shrink the viewport, you notice horizontal scrolling. The reason is that flexbox doesn't wrap by default.

```
<div class="section">
    <div class="item"></div>
    <div class="item"></div>
    <div class="item"></div>
</div>
```

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

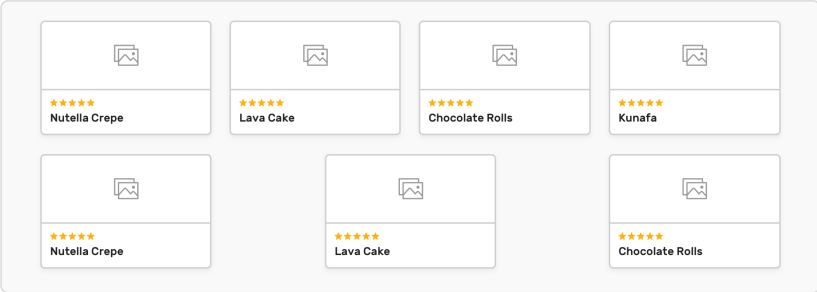



Notice how the items aren't wrapping onto a new line, thus causing horizontal scrolling. That is not good. Always make sure to add `flex-wrap: wrap`.

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

### Using `justify-content: space-between` for Spacing

When we use flexbox to make, say, a grid of cards, using `justify-content: space-between` can be tricky.




The grid of cards above is given `space-between`, but notice how the last row looks weird? Well, the designer assumed that the number of cards would always be a multiple of four (4, 8, 12, etc.).

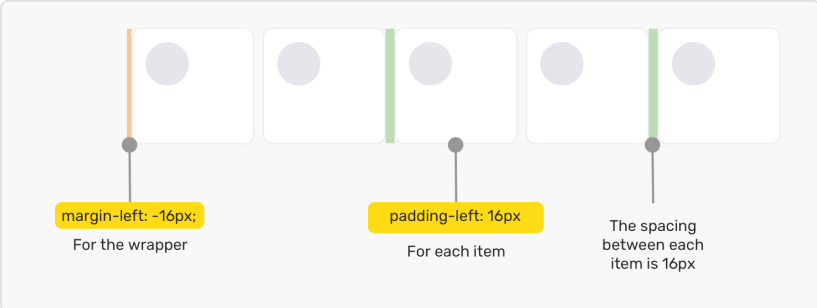
CSS grid is recommended for such a purpose. However, If you don't have any option but to use flexbox to create a grid, here are some solutions you can use.

### Using Padding and Negative Margin

```
<div class="grid">
    <div class="grid-item">
        <div class="card"></div>
    </div>
    <!-- + 7 more cards -->
</div>
```

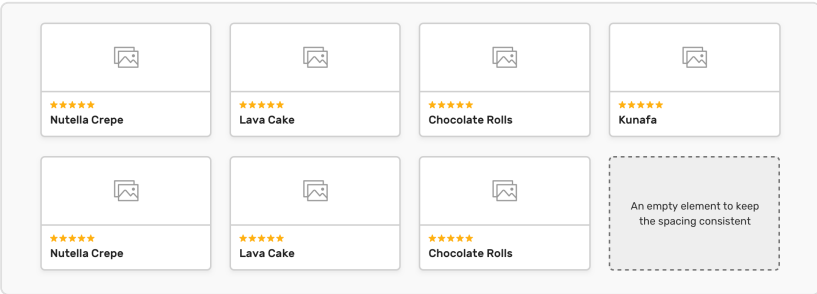
```
.grid {
    display: flex;
    flex-wrap: wrap;
    margin-left: -1rem;
}

.grid-item {
    padding: 1rem 0 0 1rem;
    flex: 0 0 25%;
    margin-bottom: 1rem;
}
```

Each `grid-item` has padding on the left side, but it's not needed for the first grid item of each row. To avoid having to use complex CSS selectors, we can just push the wrapper to the left by using a negative margin on the left side.




### Adding Empty Spacer Elements

While studying Facebook's new CSS, I noticed an interesting use case of a spacer element for the problem we are solving now.




If we have a grid of six items, the last two will be added as empty spacer elements. This ensures that `space-between` works as expected.

```
<!-- before -->
<div class="grid">
    <div class="grid-item">…</div>
    <div class="grid-item">…</div>
    <div class="grid-item">…</div>
</div>

<!-- after -->
<div class="grid">
    <div class="grid-item">…</div>
    <div class="grid-item">…</div>
    <div class="grid-item">…</div>
    <div class="empty-element">…</div>
</div>
```

Again, the empty element's purpose is to keep the spacing working as expected. Of course, this should be done dynamically.

### Hiding a Flexbox Element in Certain Viewports

Hiding a flexbox element on mobile and showing it on desktop can be tricky.

```
.element { display: none; }

@media (min-width: 768px) {
    .element {
        display: block;
    }
}
```

You might thoughtlessly type `display: block` because that is the common way to show a hidden element. However, because the element is a flex wrapper, a display value of `block` could break the layout. This mistake could lead to some debugging time.

```
@media (min-width: 768px) {
    .element {
        display: flex;
    }
}
```

### Stretched Images

By default, flexbox does stretch its child items to make them equal in height if the direction is set to `row`, and it makes them equal in width if the direction is set to `column`. This can make an image look stretched.

```
<article class="recipe">
    <img src="recipe.png" alt="" />
    <h2>Recipe title</h2>
</article>
```

```
.recipe { display: flex; }

img { width: 50%; }
```

A simple online search reveals that this issue is common, and it has inconsistent browser behavior. The only browser that still stretches an image by default is Safari version 13. To fix it, we need to reset the alignment of the image itself.

```
.recipe img { align-self: flex-start; }
```

**Recipe title**

Some description

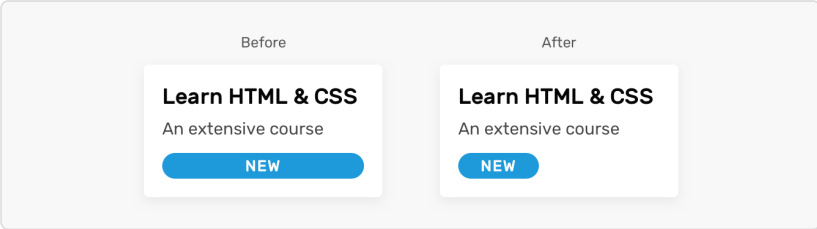
While Safari version 13 is the only one that has the inconsistent behavior of stretching the image, the `button` element is stretched in all browsers. The fix is the same (`align-self: flex-start`), but small details like this make you think about the weirdness of browsers.

We see a related problem when a flex wrapper has its direction set to `column`.

```
<div class="card">
    <h2 class="card__title"></h2>
    <p class="card__desc"></p>
    <span class="card__category"></span>
</div>
```

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

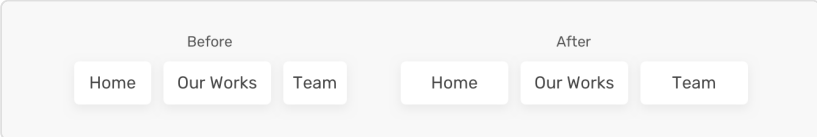
The `.card__category` element will stretch to take up the full width of its parent. If this behavior is not intended, then you'll need to use `align-self` to force the `span` element to be as wide as its content.




```
.card__category {
    align-self: flex-start;
}
```

### Flexbox Child Items Are Not Equal in Width

A common struggle is getting flexbox child items to be equal in width.




According to the specification:

> If the specified flex-basis is auto, the used flex-basis is the value of the flex item's main size property. (This can itself be the keyword auto, which sizes the flex item based on its contents.)

Each flex item has a `flex-basis` property, which acts as the sizing property for that item. When the value is `flex-basis: auto`, the basis is the content's size. So, the child item with more text will — you guessed it — be bigger. This can be solved by doing the following:

```
.item {
    flex-grow: 1;
    flex-basis: 0%;
}
```

With that, each child item will take up the same space as its siblings.

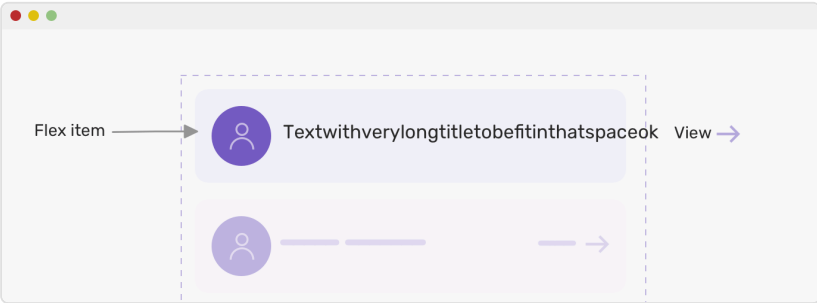
### Setting the Minimum Width to Zero With Flexbox

The default value of `min-width` is `auto`, which is computed to `0`. The `min-width` of a flex item is equal to the size of its contents.

According to the CSS specification:

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

Consider the following example:



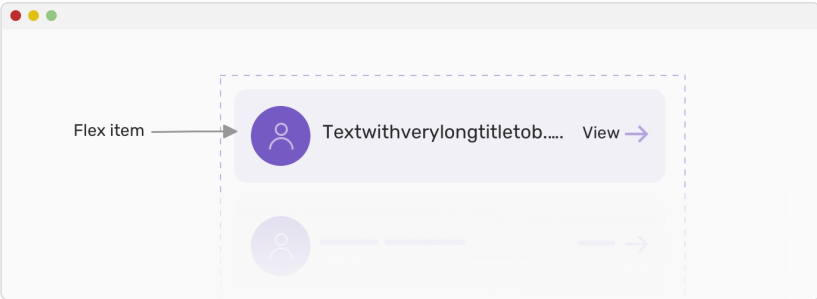

The person's name is long, which causes horizontal scrolling. So, we add the following to truncate it:

```
.c-person__name {
    white-space: nowrap;
    overflow: hidden;
    text-overflow: ellipsis;
}
```

The trick is to add `min-width: 0` to the element.

```
.c-person__name {
    /* Other styles */
    min-width: 0;
}
```

Here is how it should look when fixed:




### Flex Formatting Context

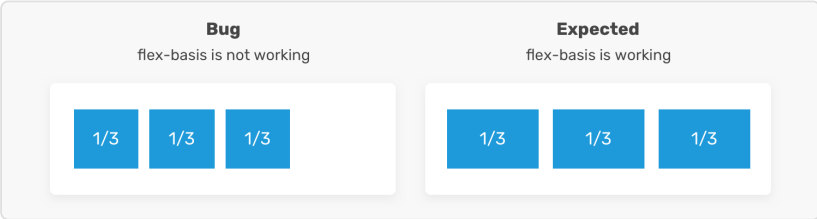
It's worth mentioning that when we assign `display: flex` to an element, the flex container establishes a new flex formatting context. Adding `float` won't work. Moreover, there is no margin collapse for the child items of a flex container.

## Browser Implementation Bugs

Let's walk through some of the most common issues with flexbox related to incorrect or inconsistent browser implementation.

In this section, I will rely heavily on Flexbugs, a great resource by Philip Walton for all browser bugs related to flexbox.

### `flex-basis` Doesn't Support `calc()`




When using the shorthand version of the `flex` property, Internet Explorer versions 10 to 11 ignore any `calc()` functions.

```
.element {
    flex: 0 0 calc(100% / 3);
}
```

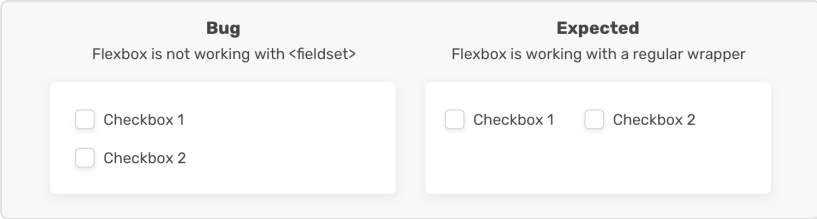
The solution is to write out the longhand version.

```
.element {
    flex-grow: 0;
    flex-shrink: 0;
    flex-basis: calc(100% / 3);
}
```

In Internet Explorer 10, the `calc()` function doesn't work in the longhand `flex-basis` declaration. To work around this, we do the following:

```
.element {
    flex: 0 0 auto;
    width: calc(100% / 3);
}
```

### Some HTML Elements Can't Be Flex Containers




Elements such as `button`, `fieldset`, and `summary` don't work as flex containers. The flexbox repository gives the following reason:

> The browser's default rendering of those element's UI conflicts with the `display: flex` declaration.

Consider the following example:

```
<fieldset>
  <legend>Enter your information</legend>
    <p>
        <label for="name">Your name</label>
        <input type="text" id="name">
    </p>
    <p>
        <label for="email">Email address</label>
        <input type="email" id="email">
    </p>
</fieldset>
```

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

You would assume that the inputs will be displayed next to each other, right? That's not the case with this bug. It won't work. A workaround is to wrap the inputs in another element that can act as a flex container.

```
<fieldset>
    <div class="inputs-group">
        <!-- inputs -->
    </div>
</fieldset>
```

```
.inputs-group {
    display: flex;
    flex-wrap: wrap;
}
```

That fixes the issue.

The `button` element bug is fixed in Chrome, Firefox, and Safari.

### Inline Elements Not Treated as Flex Items

All inline elements, including `::before` and `::after` pseudo-elements, don't work as flex items in Internet Explorer 10. In version 11, this bug was fixed for regular inline elements, but it still affected the `::before` and `::after` pseudo-elements.

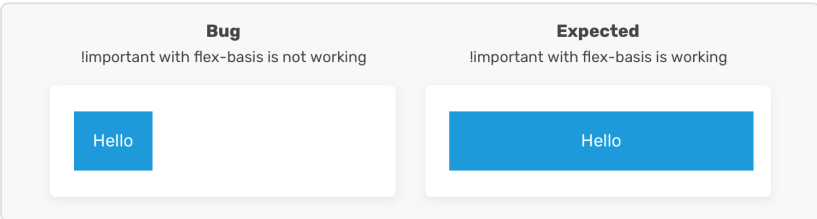
```
<div class="element"></div>
```

```
.element {
    display: flex;
}

.element::before {
    content: "Hello";
    flex-grow: 1;
}
```

The `::before` pseudo-element won't work as a flex item. The workaround is to add a `display` value other than `inline` to the item (for example, `inline-block`, `block`, or `flex`).

```
.element::before {
    content: "Hello";
    flex-grow: 1;
    display: block;
}
```

### Importance Is Ignored in `flex-basis` When `flex` Shorthand Is Used




In Internet Explorer 10, the `!important` rule doesn't work with `flex-basis` in the shorthand version.

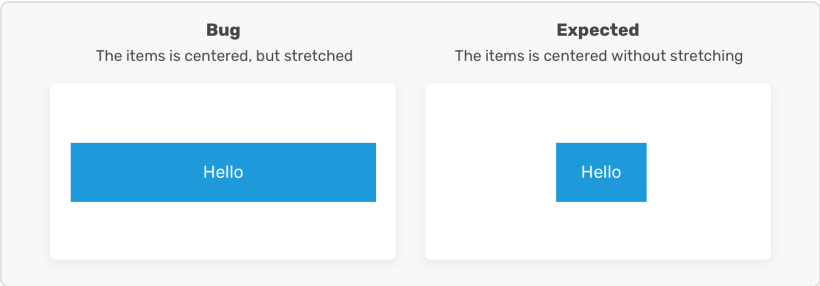
```
.element {
    flex: 0 0 100% !important;
}
```

This won't work. The `flex-basis` setting of `100%` will be ignored. We need to write out the longhand version.

```
.element {
    flex: 0 0 100% !important;
    flex-basis: 100% !important;
}
```

Note that this bug was fixed in Internet Explorer 11.

### Centering a Flex Item With `margin: auto` Doesn't Work With Flexbox Wrapper Set to Column




You can use the `margin: auto` to center a flex item in its container. In Internet Explorer versions 10 to 11, this feature doesn't work when the direction of the flexbox wrapper is a column.

```
<div class="wrapper">
    <div class="item"></div>
</div>
```

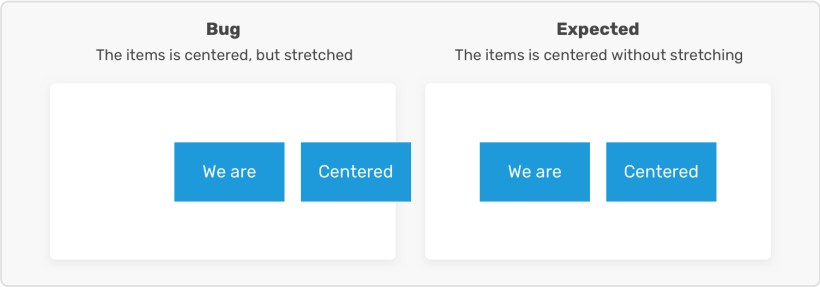
```
.wrapper {
    display: flex;
    flex-direction: column;
}

.item { margin: auto; }
```

Instead of the `.item` being centered, it is rendered according to `align-self: stretch` (the default value). The solution is either to:

- use `align-self: center` on the item itself,
- use `align-items: center` on the wrapper.

This issue has been fixed in Microsoft Edge.

### Flex Items Don't Justify Correctly With `max-width`




When `max-width` is used for a flex item, in conjunction with `justify-content` on the flex wrapper, the spacing is not calculated correctly.

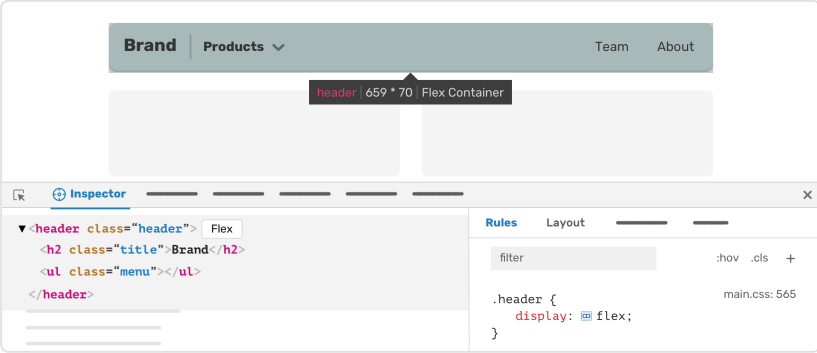
```
.item {
    flex: 1 0 0%;
    max-width: 25%;
}
```

The expected result here is that the size of the elements would start from `0%` ( `flex-basis` ) and won't be more than `25%` ( `max-width` ). We can achieve the same effect by setting a value for `max-width` instead of `flex-basis`, and we can let it shrink by setting a minimum size ( `min-width` for a row direction, and `min-height` for a column direction).

```
.item {
    flex: 0 1 25%;
    min-width: 0%;
}
```

## Firefox's Flexbox Inspector

Firefox has some great resources for debugging flexbox components in its DevTools. It shows a label of "flex" next to each element that is a flex container. When an element is hovered over in the "Inspector" panel, the information bar (the dark-grey one) shows the type of the flex element.



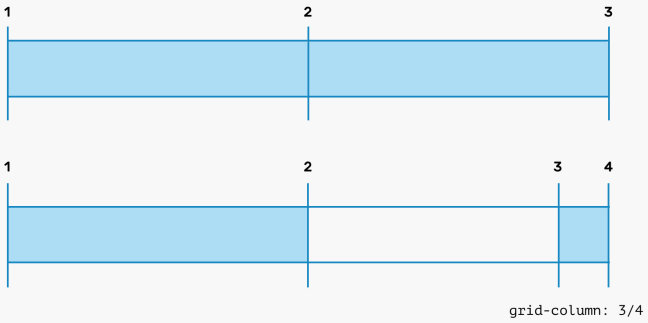

The great thing is that the "flex" label is clickable. When it's clicked, Firefox will highlight the flex layout items. It can also be accessed from the little flexbox icon beside the CSS declaration in the "Rules" panel.


The highlight is useful when you're in doubt of how a flexbox layout works. Take advantage of these tools — they enable you to make sure that nothing weird is happening and clear up any confusion about a flexbox container.

# CSS Grid

## Unintentional Implicit Tracks

A common misstep with CSS grid is to create an additional grid track by placing an item outside of the grid's explicit boundaries. First, what's the difference between an **implicit** and **explicit** grid?




```
.wrapper {
    display: grid;
    grid-template-columns: 1fr 1fr;
}

.item-1 {
    grid-column: 1 / 2;
}

.item-2 {
    grid-column: 3 / 4;
}
```

The `.item-1` element has an **implicit** grid track, and it's placed within the grid's boundaries. The `.item-2` element has an **explicit** grid track, which places the element outside of the defined grid.

CSS grid allows this. The problem is when a developer is not aware that an implicit grid track has been created. Make sure to use the correct values for `grid-column` or `grid-row` when working with CSS grid.

## A Column With `1fr` Computes to Zero

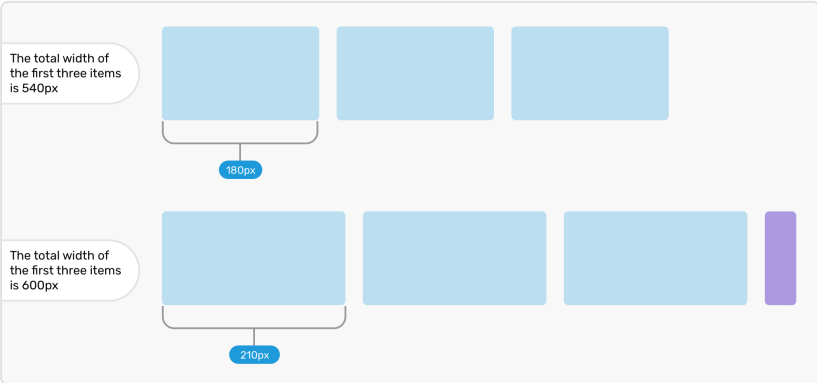
There is a case in which a column with `1fr` will compute to a width of `0`, which means it's invisible.

```
<div class="wrapper">
    <div class="item"></div>
    <div class="item"></div>
    <div class="item"></div>
    <div class="item"></div>
</div>
```

```
.wrapper {
    display: grid;
    grid-template-columns: repeat(3, minmax(50px, 200px)) 1fr;
    grid-template-rows: 200px;
    grid-gap: 20px;
}
```

We have three items with a minimum of 50 pixels and a maximum of 200 pixels. The last item should take the remaining space, `1fr`. If the sum of the widths of the first three items is less than 600 pixels, then the last column will be **invisible** if:

- it has no content at all,
- it has no border or padding.



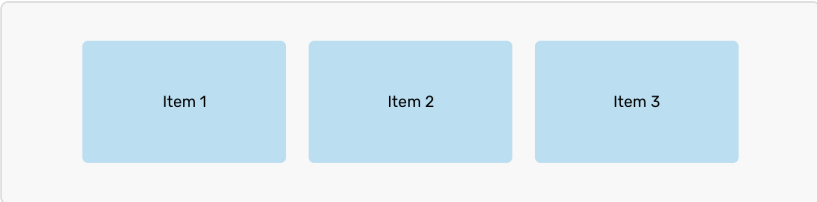

Keep that in mind when working with CSS grid. This issue might be confusing at first, but when you understand how it works, you'll be fine.

## Equal `1fr` Columns

You might think that the CSS grid fraction unit, `fr`, works as a percentage. It doesn't.

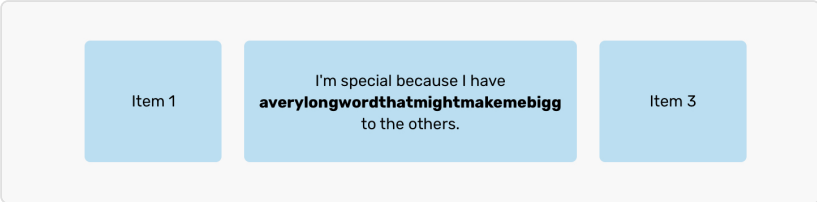
```
<div class="wrapper">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
</div>
```

```
.wrapper {
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
    grid-template-rows: 200px;
    grid-gap: 20px;
}
```




The items look equal. However, when one of them has a very long word, its width will expand.

```
<div class="wrapper">
    <div class="item">Item 1</div>
    <div class="item">I'm special because I have
averylongwordthatmightmakemebiggerthanmysiblings.</div>
    <div class="item">Item 3</div>
</div>
```




Why does this happen? By default, CSS grid behaves in a way that gives the `1fr` unit a minimum size of `auto` (`minmax(auto, 1fr)`. We can override this

and force all items to have equal width. The default behavior might be good for some cases, but it's not always what we want.

```
.wrapper {
    /* other styles */
    grid-template-columns: repeat(3, minmax(0, 1fr));
}
```

Beware that the above will cause horizontal scrolling. See the section on horizontal scrolling for ways to solve it.

## Setting Percentage Values

The unique thing about CSS grid that it has a **fraction** unit, which can be used to divide columns and rows. Using percentages goes against how CSS grid works.

```
.wrapper {
  display: grid;
  grid-template-columns: 33% 33% 33%;
  grid-gap: 2%;
}
```

Using percentage values for `grid-template-columns` and `grid-gap` would cause horizontal scrolling. Instead, use the `fr` unit.

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-gap: 1rem;
}
```

## Misusing `auto-fit` and `auto-fill`

I wouldn't consider this a bug, but misusing `auto-fit` and `auto-fill` can lead to an unexpected result. Let's differentiate them first. Take the following grid:

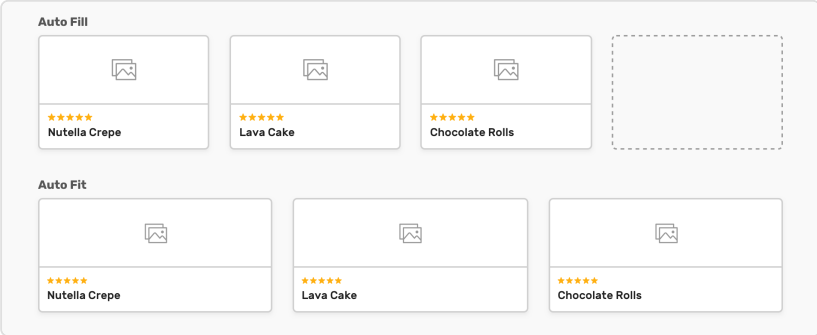
```
.wrapper {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
    grid-gap: 1rem;
}
```

Out goal is to have a minimum 200-pixel width for the grid item.

In `auto-fill`, the empty tracks won't collapse to `0`, thus keeping the space as it is.

In `auto-fit`, the browser will keep a minimum size of 200 pixels, and if space is available, the empty tracks will collapse to `0`. Thus, the grid items will take up the remaining space.

I was working on coding a layout of a new section for a client, and while testing, I found a bug telling me that there was an **empty** space on the right side. I opened up the DevTools and realized that I was using `auto-fill` for the grid.




As you can see, this is not exactly a bug, but it affected the result and confused me.

## Horizontal Scrolling and `minmax`

As I mentioned in the section on horizontal scrolling, using `minmax()` without proper testing can cause grid items to be wider than the viewport, which will result in horizontal scrolling.

```
.wrapper {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(350px, 1fr));
    grid-gap: 16px;
}
```

If the viewport is narrower than 350 pixels, then horizontal scrolling will occur. We can avoid that by setting up a media query.

```
.wrapper {
    display: grid;
    grid-template-columns: 1fr;
    grid-gap: 16px;
}

@media (min-width: 400px) {
    .wrapper {
        grid-template-columns: repeat(auto-fit, minmax(350px, 1fr));
    }
}
```

This way, the `minmax()` function will be applied only when there is enough space.

### Browser Implementation Issues

Even though CSS grid is relatively new (being released in March 2017), it can still get challenging, especially when supporting the old version of it released in Internet Explorer 11. To avoid any issues, I recommend using the `@supports` query to detect whether the browser supports the new grid specification.

```
@supports (grid-area: auto) {
    /* CSS grid code goes here */
}
```

I used `grid-area` because it's a part of the new grid specification. With this, Internet Explorer 11 won't apply CSS grid. Supporting grid in Internet Explorer is not impossible, but you need to stick to its old implementation. Rachel Andrew has written about the topic in detail.

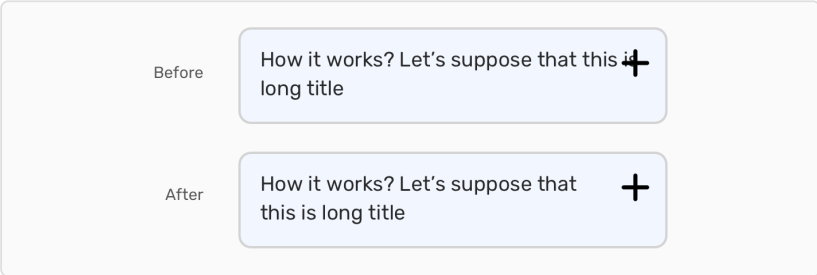
## Handling Long and Unexpected Content

The heading of this section echoes an article I wrote for CSS-Tricks. I will revisit that article and list the issues that come up in our daily work. We sometimes put effort into building components without considering how long the content might run. Think about such questions, and decide what to do in those cases.

> When you code CSS, you're writing abstract rules to take *unknown* content and organize it in an *unknown* medium. - Keith J Grant

### Forgetting to Set Padding Between Text Label and Icon

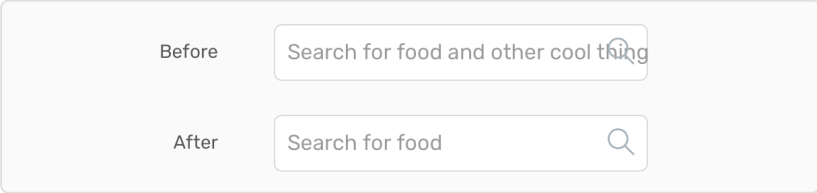
In some layouts, we need to add an icon as a CSS background for an accordion element or an input field.




Notice how the text overlaps the icon. That's because there is no padding on the right. Fixing this is simple, but finding the bug before a user does is hard. I will explain some techniques to prevent bugs from happening in the next chapter.

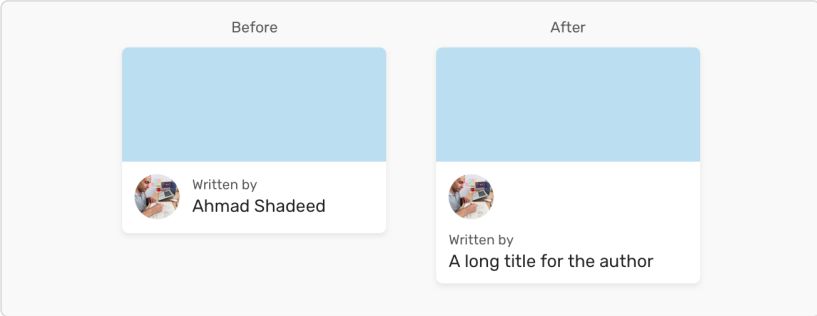
```
.accordion { padding-right: 50px; }
```

The same thing can happen with an input that has an icon.




## Long Name in Media Object

"Media object" is a term coined by Nicole Sullivan. It consists of an image on the left and descriptive text on the right. Without much effort on our part, the text will break onto a new line in case it's long and there is not enough space for it to fit beside the image. However, if it goes on a new line, it could break the design or look weird.




There is more than one solution to this problem. The most common are:

## 4. CSS Properties That Commonly Lead to Bugs

- the good ol' float,
- flexbox.

Suppose our markup looks like this:

```
<div class="card-meta">
    <img src="author.jpg" alt="">
    <div class="author">
    <span>Written by</span>
    <h3>Ahmad Shadeed</h3>
  </div>
</div>
```

### Solution 1: Float

To do this, we would need to float the image to the left and then add a clearfix to account for the issue caused by floats.

```
.card-meta img { float: left; }

.card-meta::after {
    content: "";
    display: table;
}
```

### Solution 2: Flexbox

Flexbox is better, because we only need to apply it to the parent element.

```
.card-meta { display: flex; }
```

This will keep the image and text on the same line. However, we should

account for another scenario, which is if we don't want the person's name to wrap onto a new line? In this case, `text-overflow` to the rescue.

```
.card-meta h3 {
    white-space: nowrap;
    text-overflow: ellipsis;
    overflow: hidden;
}
```

## Wrapping Up

Now that we've reached the end of this chapter, I hope you're more comfortable with the most common CSS properties and their issues. Of course, I haven't mentioned every single property, but I've tried to include the things that you will be addressing in your daily work.

If you've gone through the first four chapters carefully, then you will be able to tackle any CSS issue from the start to finish using the techniques you've learned.

# Chapter 5

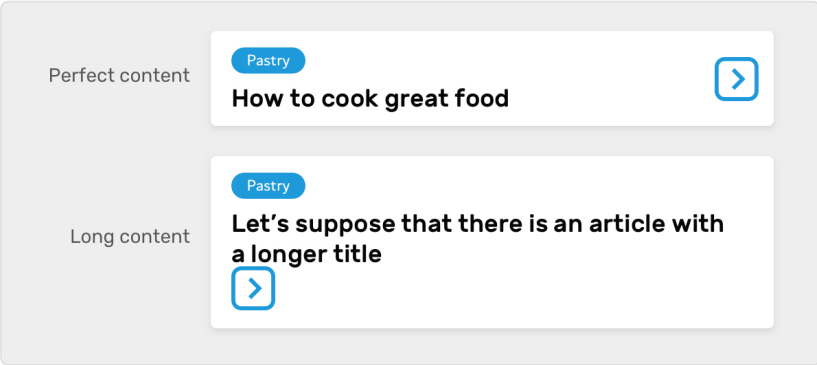
## Different Ways To Break a Layout

5

This whole book is about finding CSS issues and solving them. Can we do the opposite? In this section, we will explore different ways to break a CSS layout and make it fail. Yes, you read that correctly.

## Add Long Text Content

As we've seen, a common reason for a layout bug is text being longer than expected. Adding long text randomly can reveal CSS issues that you haven't thought about.




The article title in the first box is short; the designer wrote it to make it fit. The developer copied the text and implemented the component based on that. When I tried to add long text, an interesting thing happened! The link icon was pushed to a new line.

We can't assume this is an error, but we can ask ourselves a couple of questions:

- Is this behavior intentional? That is, when the text gets long, should it push other items to a new line?

- Or is this unexpected, and should it not happen at all?

Some CSS issues happen due to a misunderstanding between the designer and developer. The designer hasn't worked out all possible scenarios, and the developer hasn't thought about asking questions about the component. The fault is on both sides.

## forceFeed.js

Thankfully, tools exist to help us add random content and test for issues. forceFeed.js is one of them. Let's go through how it works.

### Install

First, install it via `npm` or `bower`:

```
npm install forcefeed
/* or */
bower install forcefeed
```

Or simply download the JavaScript file from the GitHub repository.

### Include the Script

Include the script after the page's content and before the end of the `body` element.

```
<script src="path/to/forceFeed.js"></script>
</body>
```

### Add Attribute to Elements

Add the attribute `data-forcefeed` to the elements you want to set random content on.

```
<div class="person">
  <h3 class="name" data-forcefeed="words|2"></h3>
  <p class="description" data-forcefeed="sentences|3|6">This will be
overridden</p>
</div>
```

The first one, `data-forcefeed="words|2"`, will generate two random words, according to the defined array, and `data-forcefeed="sentences|3|6"` will generate a random number of sentences ranging between three and six.

### Add the Arrays

```
window.words = ['Design', 'Work', 'Awesome', 'Cool'];

window.sentences = ['Can you break me?', 'I love food and baking',
'How are you today?', 'When was the last time you saw mom?'];
```

### Execute the Script

Finally, we need to make the script work on the page.

```
forceFeed({words: window.words, sentences: window.sentences});
```

With that, we can now refresh the page ( `Command` or `Control` + `R` ) to see the content change. Perhaps you will notice a broken element.

## Try Content in Different Languages

If you're working on a multilingual website, then the chances are high that some design components will break when they have different content.

Right to Left (RTL), Without letter-spacing

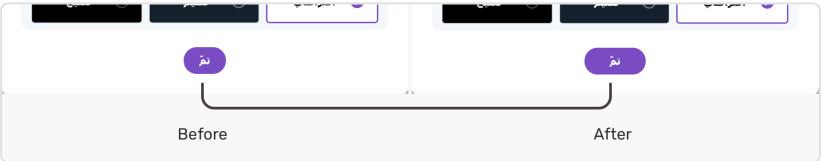
فطيرة الجبن بالتوت البري

Right to Left (RTL), With letter-spacing

فطيرة الجبن بالتوت البري

We might have an English title that has modified kerning (the spacing between characters). It might work great, but when the translated page has Arabic content in right-to-left (RTL) mode, the text breaks. Arabic has no such thing as kerning.

Another bug can occur when we assume a specific minimum size for a button component. When the content is translated into another language, it might look different.




On Twitter, the "Done" button looks good in English. In Arabic, the button looks too small and is not easily noticeable. The reason is that the button has a

rule of `min-width: 40px`. This can be fixed by increasing the minimum width of the button.

These bugs related to language are important and shouldn't be ignored. If you are interested in learning more about the issue, I've written a complete guide about it, RTL Styling 101.

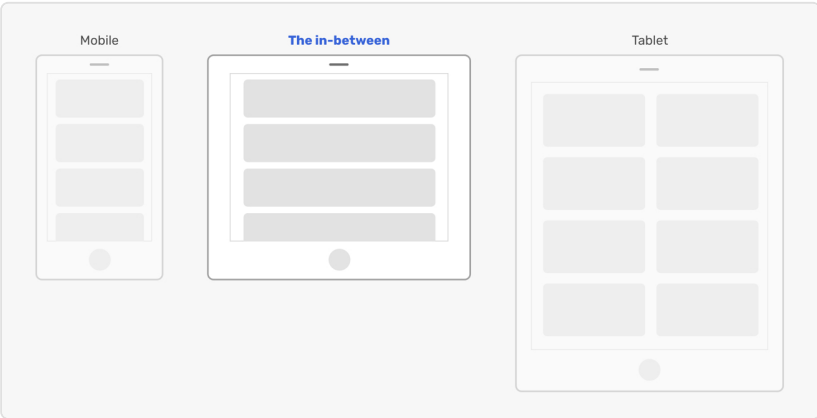
## Resize the Browser's Window

This is one of the easiest ways to break a layout and uncover its weaknesses. When you resize the browser window, you'll see some issues that you wouldn't normally notice. One interesting area to focus on is what I call "in-between" design cases. I've written a detailed article about it on my blog, and I'd like to go over the concepts again here.

> In responsive web design, it's common to work on different variations of a page. A typical web page should have two variations at least, one for small screen sizes (e.g. mobile) and the other for large screens (e.g. desktop). Often times, we forget about the **in-between** design variation, and we end up with a component or section being too wide or narrow.

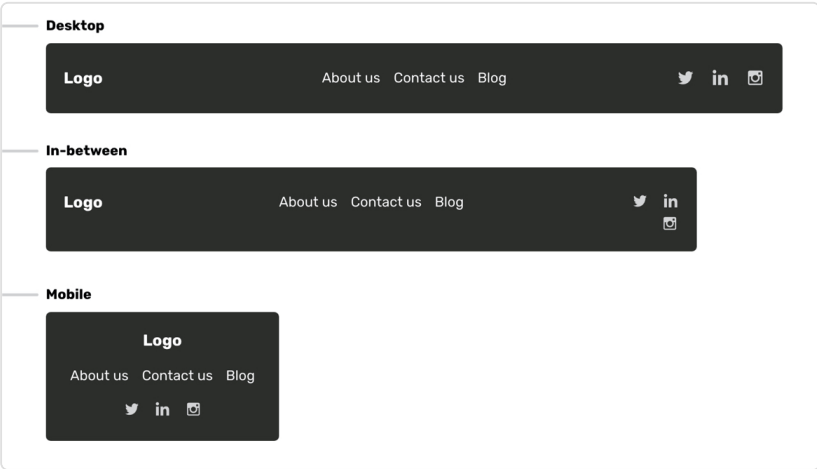
In other words, you will uncover more issues when you test the in-between design states. Believe me, you will find some interesting issues that you or the team haven't considered.

## 5. Breaking a Layout Intentionally




Here we have some cards that need to be one column on mobile and two columns on tablets. The in-between state makes the cards look **too wide**, which can affect readability. While this might not seem like a bug, it is.

Another clear example of the importance of testing the in-between state is the following footer design, taken from a real project.



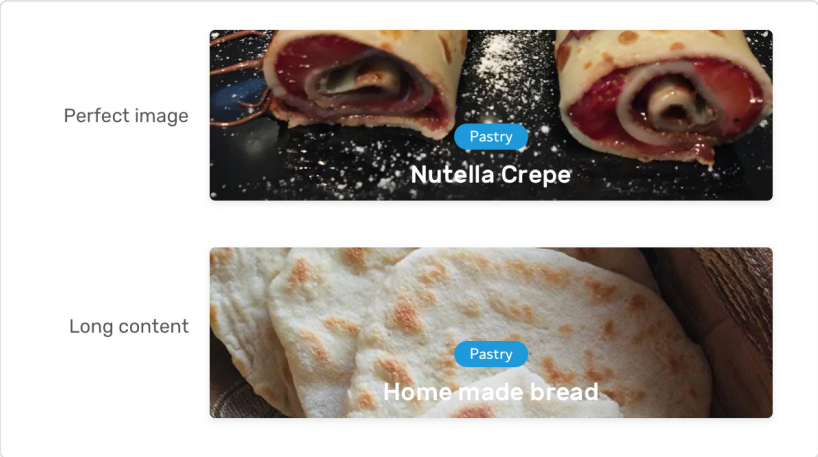

In the middle view, the social media icon for Instagram breaks onto a new line. Such behavior is not expected and shouldn't happen at all.

This should suffice to illustrate that such issues can't be ignored. Thanks to the simple trick of browser resizing, we can discover them fairly easily.

## Avoid Placeholder Images

Images play an important role in making web pages accessible and easy to read. Your job as a front-end developer is to provide a solid structure for a component that can handle any image used. For example, you might be working on a hero section with a perfect cover image and accompanying text.

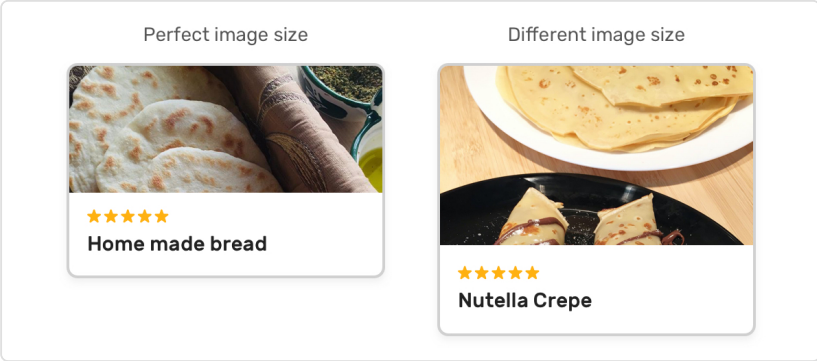
## 5. Breaking a Layout Intentionally




Breaking such a component is easy — just change the image. Suddenly, you'll see that the text is hard to read. We've forgotten to place a semi-transparent black overlay that would make the text easy to read. Unfortunately, some designers assume that the implementation of their design will exactly match their mockup. That isn't the case. A lot of changes will happen in the development process.

By changing images and trying different styles on them, we can uncover hidden issues.

Image sizes and dimensions are another area that deserves attention. As the front-end developer, you might prepare a media component that can be used for images in things like recipes and articles. One question to pose is: What image dimensions are expected or recommended? Should the content manager be restricted to uploading images with predefined sizes to the content management system (CMS), or should they be free to upload whatever sizes they like?




We have a media component containing an image here. The first one is the default image provided by the designer, and the second one is added by an author to the CMS. That inconsistency is not good. The designer and developer should agree on image sizes, and then teach the authors to follow that standard.

If you don't have much control over the image size, then I recommend using CSS `object-fit` with a fixed height for the image. That can keep all images within the same height without breaking them.

```
.media__thumb {
    height: 220px;
    object-fit: cover;
}
```

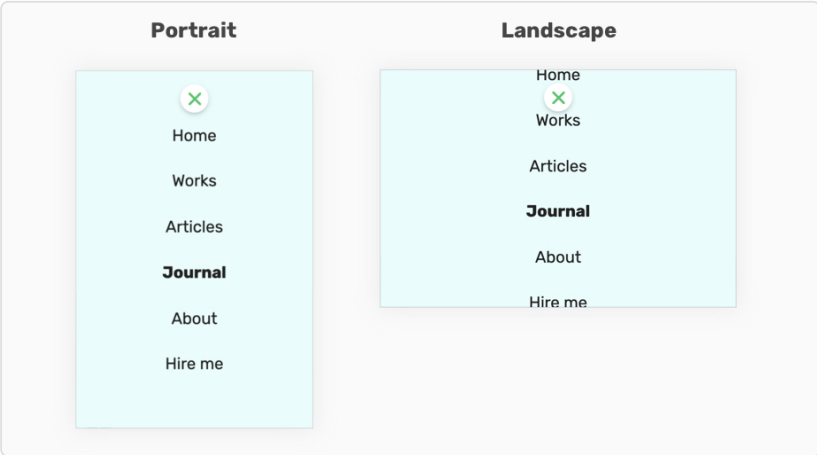
## Open in Internet Explorer

Nope, this is not a joke. Internet Explorer is well known for breaking websites. If your website is required to work in Internet Explorer, then you will need to think about every CSS decision you've made. For example, if you've used CSS

grid for the layout, then it's recommended to add a fallback using flexbox or an older layout method (floats, inline-block, etc.).

## Rotate Between Portrait and Landscape Orientation

While working on an update to the mobile menu of my personal website, I found an interesting design issue. Something as simple as rotating the device from portrait to landscape orientation can reveal unexpected issues, especially if some elements are fixed or absolutely positioned.




The "close" button is absolutely positioned and horizontally centered. In landscape orientation, the button overlaps with the navigation, which is clearly not intended. Testing in both orientations is important.

## Wrapping Up

In this chapter, we learned about how to intentionally break a layout. Next, we will explore browser inconsistencies and implementation bugs.

# Chapter 6

## Browsers Inconsistency and Implementation Bugs

We all know that web browsers have inconsistencies, and that's fine. As web developers, we have to fix those on day one of a project, so that we can start with a clean and solid code base. In this chapter, I will go over the most common CSS resets, along with how to make reduced test cases and do regression testing.

## Using a CSS Reset File

CSS reset files are an important part of web development. The two most common are Reset CSS by Eric Meyer and Normalize.css by Nicolas Gallagher.

By using a CSS reset file, you will save yourself a lot of time fixing and debugging issues that have already been fixed. Take the following example from Normalize.css:

```
/**
 * 1. Correct the line height in all browsers.
 */

html {
  line-height: 1.55; /* 1 */
}
```

Having a consistent `line-height` across all browsers is important. It will save you time from figuring out why the `line-height` does not work consistently across browsers.

Another example is making elements such as `b` and `strong` have a bold font weight. This doesn't work consistently in all browsers, so adding it will prevent unexpected behavior.

```
/**
 * Add the correct font-weight in Chrome, Edge, and Safari.
 */

b,
strong {
  font-weight: bold;
}
```

These are only a couple of examples. Remember to include a CSS reset file. If you don't want to, then at least work on your own reset file. Some will argue that a reset file is not always needed because it will increase the total size of your CSS files. I agree. But you can easily create your own small file and be done with it.

## Using Normalize.css

Compared to some other resets, Normalize.css only fixes issues with browser consistency, without resetting everything. It will keep common CSS styles among web browsers. For example, the margins of heading elements such as `h1` and `h2` will be preserved.

Here is a snippet from Normalize.css that resets the `body` element's margin:

```
body {
  margin: 0;
}
```

Depending on the nature of the project you're working on, you can decide what's best to use.

## Browser Implementation Bugs

Web browsers are made by humans, and humans make errors. It's totally normal to find that a browser does not support a particular feature as expected or implements a feature differently than other browsers. In this section, we'll walk through the steps of finding a bug in browser implementation.

First, what is a browser implementation bug? It's a bug caused by the browser itself. The bug might be in one or multiple browsers and is caused by improper implementation of the CSS specification.

### Verify the Bug

Is there really a bug, or are you mistaken? It's a waste of time to start debugging something, only to realize later that it's not a bug, but rather a deliberate feature.

Some bugs are ones that appear in all browsers on all devices — those are easy to find. Other bugs appear in specific browsers or devices, like the Nexus 5 Android phone — finding those are harder because you would need an actual Nexus 5 device or an online emulator, which is usually not free.

Verifying that a bug is indeed a bug will vary according to its complexity. Once you are sure that it's a bug, then you'll move on to the next step.

### Decide on the Correct Behavior

Once you've verified that it's a bug, you'll need to decide how the feature ought to look and behave. For example, you could decide that a particular element's height should be between 150 and 450 pixels, and if it surpasses that, then it

will be considered a bug.

### Isolate the Bug

Once you've verified the bug and decided on the correct behavior, try to reproduce it. Let's learn how to reproduce a bug through test-case reduction.

## Test-Case Reduction

One of the most underrated skills among web developers is creating a reduced test case. When we encounter a problem with a web page we're building, we need to identify the cause. The problem might be across browsers or only on mobile browsers. Debugging an issue by working with a page's entire HTML and CSS is not ideal. We need to isolate the problem in a test case, so that our time is spent more on fixing the problem than on identifying it.

Let's learn how to make a reduced test case.

1. **Disable JavaScript**
   If you disable JavaScript and the problem is still there, then you'll know the problem has nothing to do with JavaScript. This is helpful for quickly ruling out JavaScript-related issues. Adding on that, you can open the project in a private mode tab, or deactivate all the browser extensions. If the issue disappears, then it might be because a browser extension.

2. **Identify the problem**
   Is the problem related to alignment? Or horizontal flow? Whatever it is, we need to identify it. It helps to articulate the problem very precisely: "I'm going to debug the horizontal scrolling in the hero section."

3. **Isolate the HTML**
   Open up the browser's DevTools and comb through the `head` element of the page, eliminating any unneeded style and script files. If something looks unrelated, remove it and move on. Once the `head` is cleaned up, move on to the `body` element and remove all HTML that is not related to the problem.

4. **Isolate the CSS**
   Now that we're sure the HTML contains only the bit of code we want to debug, let's isolate the CSS needed for that HTML. Remove any decorative styles, like `color` and `background-color`. Keep the CSS as minimal as possible, leaving only the CSS that is causing the problem.

5. **Comment the code**
   If you doubt that a particular HTML element or CSS rule is causing the issue, add a comment explaining as much, so that you don't forget while debugging. Comments are extremely useful when you need to get help from others or as a reminder if you come back to the issue after some time.

6. **Use the reduced files**
   Once we're done isolating the problem, let's copy the remaining HTML and CSS to a new local file, and then we'll be ready. We now have a reduced test case of the problem. Better yet, we can add the HTML and CSS to CodePen, which will make it easy to test it ourselves or to get help from others.

## Example of Reduced Test Case

To make things clearer, let's go through a real example of a CSS bug and see how to convert it into a reduced test case.

## 6. Browser Inconsistencies and Implementation Bugs

In the figure below, we have a web page with a horizontal scrolling issue. We've tried hard and can't figure out how to solve the problem. So, let's isolate the problem as much as we can.

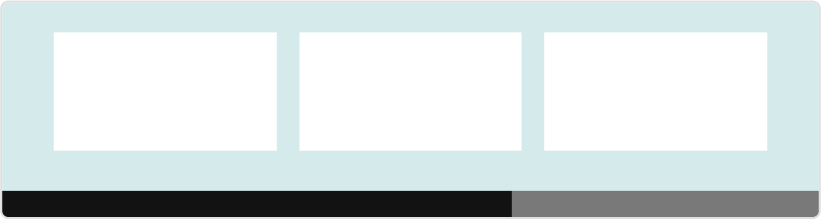

At first, we'll try the following:

1. Remove any additional styles and scripts from the page's `head` element that are not important to the demo.
2. Inspect the `body`, and delete the sections one by one. If deleting a section causes the issue to disappear, then keep it.
3. Once we've decided on the HTML that should be kept, the next step is the CSS. We need only the CSS required to make the demo work, such as `width`, `height`, and `display`. Decorative properties such as `background`, `border`, `color`, and `box-shadow` can be removed if they don't affect the issue.

Following the steps above, we should have the **least** amount of HTML and CSS to make the test case work. Repeat the steps above, and make sure that the code is clean. Here is how our reduced test case looks after we've isolated it:

It looks like the issue is in the grid section. Now, the question is, is it possible to reduce this even more? Here is what we can do:

- Remove the second row in the grid.
- Clean up the rounded corners and the shadow of the cards.
- Remove the title from each card.



Once it's cleaned, we can add some comments to aid with testing.

```
/* Not sure if 100vw is causing the horizontal scrolling. */
.section {
        width: 100vw;
        padding: 1rem;
}
```

The comment makes it helpful for us or anyone else who tries to fix the bug. This is the most we can do to reduce the test case. The next step is to extract the HTML and CSS, and upload them to our website or wherever we want.

## Make It Fail

In his book on debugging, Dave Agans identifies "make it fail" as one of the primary steps in debugging any coding problem. The extent to which we can follow this advice with CSS will depend on the type of bug we're dealing with. Some bugs are clear — for example, ones that appear in all viewport sizes. Other bug types are more complex, in which case making them fail is harder.

As Agans mentions in his book, the reasons for making it fail are so that:

- we can look at the bug,
- we can focus on the cause,
- we can tell if we've fixed it.

A line of his illustrates a misconception we often run into:

> The toast burns only if you put bread in the toaster; therefore the problem is with the bread.

Sometimes, we misunderstand the cause of a CSS issue. The toast only burns if we put it in the toaster for a long time. If it burns, then it's our fault, not the

toaster's. This same goes for CSS development. If we use a layout module to do something it wasn't designed for, then the fault is ours.

You might wonder, "What if I've tried everything I know of and still can't reproduce the issue?" Well, remind yourself that every failure has a cause. There is no secret recipe for finding it; it's hidden somewhere in the randomness.

## Back Up Your Work

Before testing, save your work in a new Git branch. If you don't use version control, copy your work in a backup, and start testing from there. By doing one of these, iterating and changing things will be much safer, and the chances of losing work will be very low.

## Document Everything

For a complex CSS bug, I like to write down the following:

1. what I did,
2. the order of steps I took,
3. what happened as a result of the steps taken.

Documenting these steps can be helpful for you and your team.

The following is an example of documented steps to reproducing a CSS bug.

1. Open the website in Safari on iOS 12.
2. Click on the mobile menu toggle.

3. Click on the close button. It doesn't work.
4. Click on the close button again. It works.

When doing the steps above, the page should be blank, except for the header, where the problem is.

## Test and Iterate

Once we have a reduced test case, we can start testing the bug in the browser or device in question. We would keep iterating and editing until we notice a difference.

When iterating, it's very important to **change one thing at a time**. Don't change the CSS randomly and hope that it will work. If it works, then you won't know how you did it, and the guesswork will start. Change one thing, test, and repeat.

The usefulness of this approach is that, when the bug is fixed, we can **compare** the changes we made to make it work. This wouldn't happen if we changed a million things at once to get it working.

## Research the Issue

If you've tried hard to fix the issue and can't do it, then the internet is your friend. Search online for the issue or pattern, and see whether others have faced the same issue. Chances are high that you are not the first.

## Report to Browser Vendors

If you believe that the bug you've found is unique and no one has ever encountered it, then it's time to report it to browser vendors. Every browser vendor has a public forum with all of the bugs submitted by users. If you've documented the steps taken to reproduce the issue, as recommended earlier, then all you need is to post the steps with your reduced test case files. Also, submit a screenshot or video if that would help.

Here is where to submit bugs in browser implementation:

- Firefox: bugzilla.mozilla.org
- Safari: bugs.webkit.org
- Chrome: bugs.chromium.org

## Never Throw Away a Debugging Demo

When working on a reduced test case, you might create multiple copies of it, each with a slight change. Don't delete them. Archive them, because they might be helpful in the future. You could do a few things with them:

- Write a blog post about the bug and your test cases, explaining how you fixed it.
- Keep them as a log for yourself, for when you face a similar bug.
- Share them with a colleague or team members who want to learn how you fixed it.

## Regression Testing

As explained on Wikipedia:

> Regression testing is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change. If not, that would be called a regression.

When you fix a bug, you might accidentally break another thing without knowing it. That is called a regression. Testing for regressions can be time-consuming, because a bug might occur in a particular environment, viewport size, or scroll position. Using a tool, we can do regression testing by defining our design components.

In this section, we will learn how to use BackstopJS to do regression testing. According to the official GitHub repository:

> BackstopJS automates visual regression testing of your responsive web UI by comparing DOM screenshots over time.

BackstopJS uses a headless Chromium browser, the same one used for Google's Chrome. Here is how it works:

1. Assign a URL path for the page we want to test.
2. Add the selectors we want to watch.
3. Generate reference screenshots for them.
4. Run `backstop test` to test our changes against the reference screenshots.

Let's learn how to use BackstopJS. First, install it globally:

```
$ npm install -g backstopjs
```

Inside our project's directory, we need to initialize a BackstopJS project.

```
$ backstop init
```

## BackstopJS Configuration File

When we initialize the project, a `backstop.json` file will be created in the root directory of the project. You will find everything that can be configured in there. For our simple lesson, the following are required:

- `id`
- `viewport`
- `scenarios`

In the `scenarios` array, there is a `selectors` array, where we can add all of the CSS selectors to watch. I've added the following:

```
"viewports": [
    {
      "label": "phone",
      "width": 320,
      "height": 480
    },
    {
      "label": "tablet",
      "width": 1024,
      "height": 768
    }
],
"scenarios": [
    {
        "url": "http://localhost:8080/"
        "selectors": [
        ".c-header--full"
      ]
    }
]
```

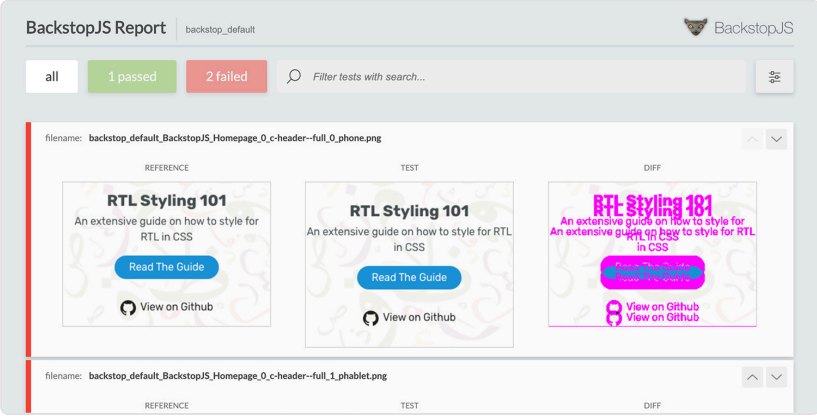
In this configuration, we've added two elements to watch, and the URL of the page is `index.html`. The next step is to generate the **reference** screenshots.

```
$ backstop reference
```

By generating a reference for the elements we've assigned, we can make changes in the CSS. If something is different from the reference, BackstopJS will throw an error, with a detailed UI.

To use the tool, let's increase the vertical padding in the header, and then rerun the test.

```
$ backstop test
```




The test failed because the reference screenshot is different from the tested one. Notice how the third screenshot has a pink color — this is a highlight of the difference.

In a real project, we might work on a fix for an issue and, in the process, inadvertently create a regression. We can't manually test everything in a large project, so having such tools makes our life easier, and makes us much more productive.

## Wrapping Up

In this chapter, we've learned about CSS resets, reduced test cases, and regression testing. We're almost done! In the next chapter, we will explore some general tips and tricks for debugging CSS.

# Chapter 7

## General Tips and Tricks

7

## Debugging Multilingual Websites

When it comes to debugging a multilingual website, we need to be aware of how to test for it and how things work. In talking about multilingual websites, we'll focus on left-to-right (LTR) and right-to-left (RTL) layouts, using the examples of English and Arabic, respectively.

### Common Bugs With LTR and RTL

#### Spacing Issues

When debugging for LTR and RTL, most of the issues will be related to spacing. The horizontal direction will be flipped for each language, and the spacing issues will usually come down to either `padding` or `margin`. Say we have the following:

```
.element {
    margin-left: 10px;
}
```

For RTL, it would be like this:

```
.element {
    margin-right: 10px;
}
```

We would do the equivalent for padding and the positioning properties (`top`, `right`, `bottom`, `left`).

Adding on that, we can use CSS logical properties to avoid writing more CSS for RTL. Here is how the above example will look:

```
.element {
    margin-inline-start: 10px;
}
```

The property `margin-inline-start` is logical. That means, it will be `margin-left` for LTR and `margin-right` for RTL. If you're interested to learn more about RTL styling, I recommend you reading this guide by yours truly.

## Alignment Issues

When text is aligned to the right in LTR, it should be flipped in RTL.

```
.element {
    text-align: right;
}
```

For RTL, it would be like this:

```
.element {
    text-align: left;
}
```

## Debugging RTL

Depending on how the website you're building works, switching the CSS from LTR to RTL for a given page might be easy. If the CSS is combined into one file, switching will be as easy as setting the `dir` attribute on the `html` element.

```
<html dir="rtl"></html>
```

We can first set the attribute in the DevTools, and then inspect the issues we want to fix.

If the CSS for the LTR and RTL isn't in one file, then it is most probably in two files, such as `main-ltr.css` and `main-rtl.css`. Switching the `dir` attribute won't be enough then; we would also need to edit the `src` of the style sheet in the `head` element.

### A Quick Way to Add RTL Content

Let's say we've built the CSS for the LTR and RTL layouts, and the only thing missing is to test the typography of the RTL content. When viewing the design in RTL mode with the LTR content, you can use Google's in-page translation to quickly translate all of the content. This will help you to create an RTL design with the content and make it suitable for the text direction.

I've written an extensive guide about this, in case you're interested, titled RTL Styling 101.

## Using `@supports`

In case you don't know about it, `@supports` is used to detect whether a given CSS feature is supported by the user's browser.

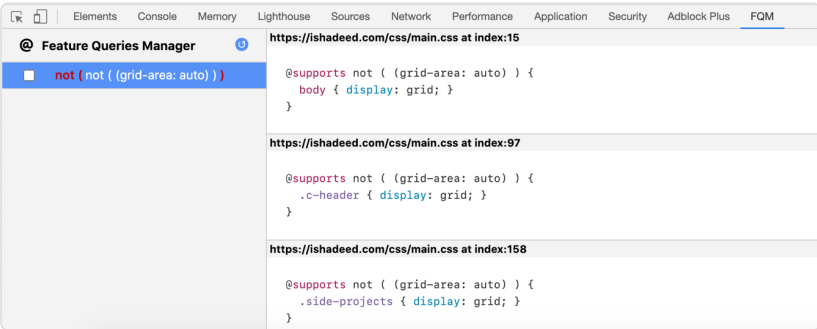
```
@supports (display: flex) {
    /* If flexbox is supported, apply this. */
    .element {
        display: flex;
    }
}
```

An interesting way to test it is to toggle its functionality. There are browser extensions for this, but we can do it manually by adding a random letter. When the random letter is added, it will break the rule; thus, the CSS won't work.

```
@supports (display: flexB) {
    ..
}
```

I added the letter "B" after `display: flex`. The browser won't recognize that, and you will get the default behavior, as if `@supports` is disabled. Cool, right?

However, in a large project with a lot of `@supports` rules, doing it manually is not practical. Thankfully, Ire Aderinokun has created a browser extension for this purpose, and it's available for both Chrome and Firefox.



The extension will add a new tab in your browser's DevTools. On the left, you'll see a toggleable list of the CSS features nested in `@supports` queries, and on the right will be a list of every `@supports` query that uses a particular feature. The CSS shown above is for grid-related stuff. Toggling the checkbox on the left will disable and enable CSS grid. This is a great way to test and break a layout. Let's get more into the ways to break a layout.

## Browser Extensions

### Grid Ruler

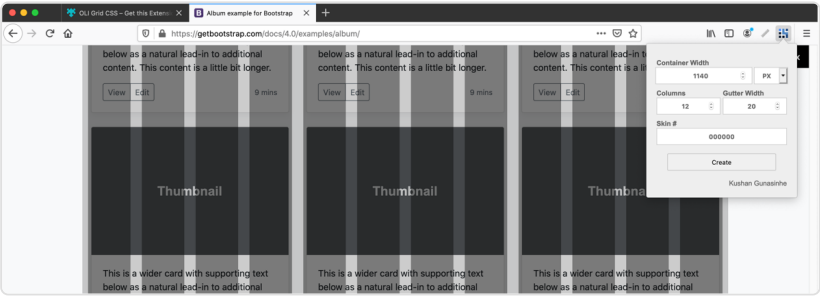
A great way to test whether two UI elements are aligned correctly is to use a ruler and guides. This can be easily done in design apps such as Sketch, Adobe XD, Photoshop, and Illustrator. In a browser, it's not possible without an extension.

One great extension, Grid Ruler, is available only in Google Chrome. It enables you to drag and place guides either horizontally or vertically. This is extremely useful for verifying that two elements are aligned correctly.


In this mockup, the grid line tells us that the user avatar and the button are aligned.

## OLI Grid CSS

The OLI Grid CSS plugin is available for Firefox and Chrome. What's nice about it is that it draw in the page columns, just like in Sketch and Adobe XD. This is helpful for seeing whether the layout you're working on aligns to the columns.




I tried testing the plugin with a Bootstrap-built page, and it works as expected. Note that you need to figure out the width of the `.container` element of the page first.

## Web Developer Extension

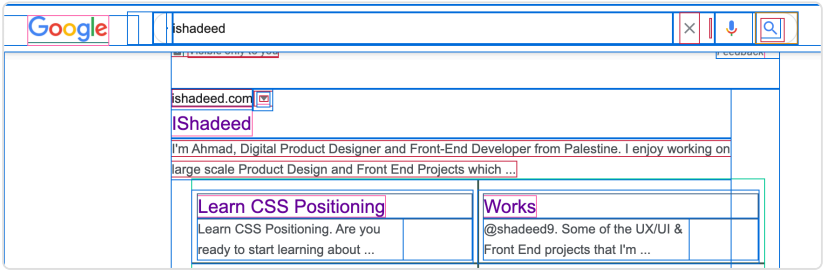

A very useful extension that provides a lot of functionalities to do. Here are some key things:

- Disable all styles
- Disable browser default styles

- Disable inline styles
- Disable print styles

And that's only a few from the CSS tab!

### Pesticide Extension




I explained previously about using the `outline` CSS property as a way to debug design issues. This blog does the same but in one click only. It adds random colored outlines to every single element on the page, with the ability to highlight a specific element.

## Mocking Up in the Browser

There are times when you want to quickly mock up a design idea in the browser by moving a few elements here and there. This is useful for showing a design concept to a developer, client, or designer. Being able to make such edits quickly is important to productivity.

Taking advantage of the browser's built-in tools, we can do that. In this section, we'll focus on concepts and examples for mocking up designs quickly in the browser.

## Good Ol' CSS Positioning

With CSS positioning, we can edit some elements in the DevTools by adding `position` to them and placing them where we want. This is a quick way to mock up a design idea while testing for bugs.

Here we have a card with a category. After some thinking, the designer tells you, the developer, that the team has decided it wants a different position for the category. You suggest that the category could be moved to the upper-left corner. This can be done while you both are on a video call. It's as simple as adding the following:

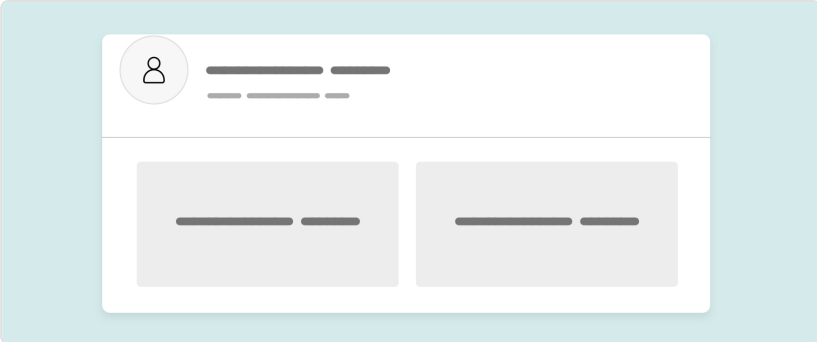
```
.card {
    position: relative;
}

.category {
    position: absolute;
    left: 0;
    top: 16px;
}
```

This kind of edit, which didn't take a minute, can allow decisions to happen

more quickly.

## Hiding Design Elements

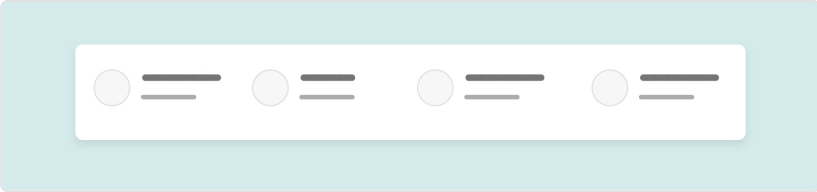
As I explained previously, being able to hide design elements quickly, such as with the `H` key in Chrome, is a useful trick. Doing this, we can hide some design elements and replace them with others, if we want to, for example, take a screenshot of a design concept.



Here we have a section header, which contains a bug that prevents the author's avatar and name from aligning. The design team has requested that it be removed temporarily. You can quickly delete it from the HTML, hide it with `display: none`, or use the `H` key in Chrome.

## CSS Flexbox

Using flexbox, we can quickly make a layout horizontal or vertical. Flexbox properties such as `align-items` and `justify-content` are powerful and can accomplish most any design idea you want to show.



This section header has a row of items. The problem is that the spacing between items is inconsistent. What can we do? The fastest solution is to add `display: flex` and `justify-content: space-between`. The design is changed instantly, and all of it happens in the DevTools! You can now proceed to screenshot this change and discuss it with your colleagues.

## CSS Grid Layout

This is the most powerful layout module in CSS. Suppose we have a featured news section, and the designer wants to lay out the items in a presentable way — say, as equal-height columns.

We simply use CSS grid to set the columns, and then we confirm with the designer that this is what they want.

```
.wrapper {
    display: grid;
    grid-template-columns: 2fr 1fr 1fr;
}
```

Is that still not enough for the designer? You can continue editing and showing them your changes. Moreover, you can try different layout concepts and tie each one to a CSS class, toggling each class in the ".cls" panel.

## CSS Viewport Units

We can use viewport units to make a section take up the full horizontal or vertical space of the viewport. We can also use them to size fonts. All of these use cases give us flexibility and make our designs more dynamic.

Suppose we have a hero section that is required to occupy 90% of the screen's full height. We want to verify the requirement with the designer, so we mock it up very quickly:

```
.hero {
    height: 90vh;
}
```

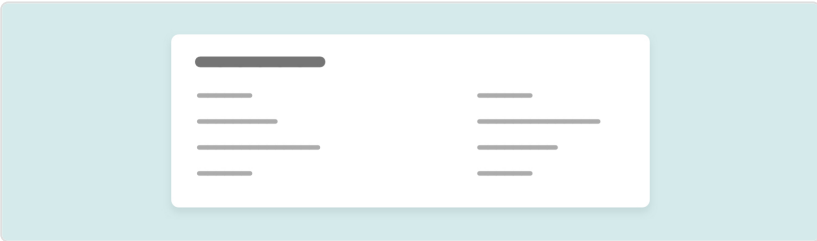
We give the hero section a height of `90vh`, which will make it occupy 90% of the screen's vertical space. We made this edit in less than a minute!

## CSS Columns

If we want a quicker method than CSS grid, we can use columns. For example, we could divide links in the footer into two equal columns. We can make this edit with one line of code, and get back to the designer right away.

```
.footer-section {
    columns: 2;
}
```

The other good thing about CSS columns is that we can change the number of columns with the keyboard's up and down arrows.



## CSS Filters

Let's say the designer wants to experiment with a dark mode for the website, but they haven't done any mockups for it. Using CSS filters, we can quickly whip up a dark mode.

```
html {
  filter: invert(90%) hue-rotate(25deg);
}
```

And to polish it, we can revert the elements that shouldn't have been inverted (such as images and videos):

```
html {
  img, video, iframe {
    filter: invert(100%) hue-rotate(-25deg);
  }
}
```

With that done, we can take a full-page screenshot and show it to the team — all in less than two minutes! Isn't that cool? With the mockup sent, the team can start thinking and deciding on it. Also, you've saved the designer's time! Making a dark mode for a small web page would take them at least 10 minutes.

## Desaturating the Design

Desaturating a page (i.e. converting it into black and white) using CSS filters is a useful trick, for a few reasons:

- If the website you are testing is heavy with colors, your eyes might get tired. Desaturating the page will help you focus on fixing the bug at hand.
- It's useful for testing and exploration. When the page is saturated, you can easily spot any colors that are not suitable for the design.
- Testing for accessibility becomes easier. Making the page grayscale will let you know which colors are easy to read and which are not.

To desaturate a web page with CSS, open up the browser's DevTools, select the `html` or `body` element, and add the following:

```
html {
    filter: grayscale(1);
}
```

That's all. You now have a black-and-white website!

### Wireframe Styling

When mocking up a design, we don't always have time to choose good colors and fonts. In this case, we can convert the whole web page into a wireframe style using a bit of CSS. This will let you focus on mocking up ideas quickly and getting feedback as soon as possible.

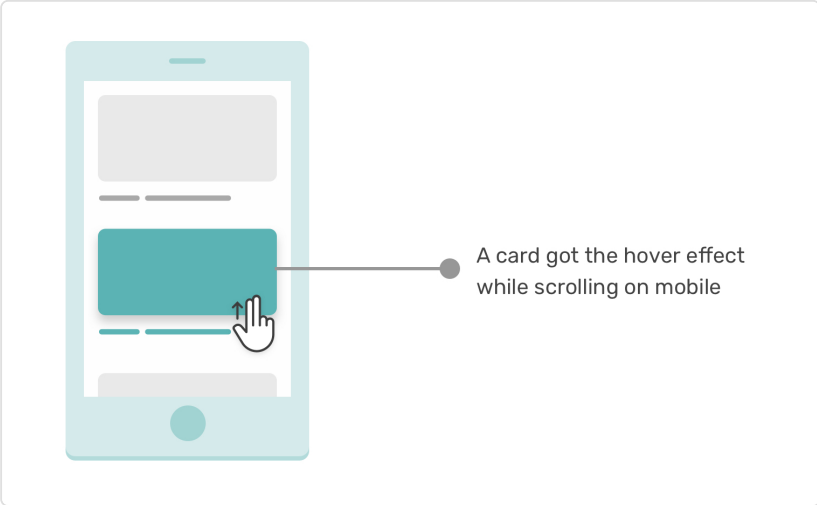
Here is how to do it:

```
* {
    color: #000;
    background: #ccc !important;
    outline: solid 1px;
}

img, video, iframe {
    background: #ccc;
    opacity: 0;
}
```

## Hover for Touch Screens

While debugging on touch devices (phones, tablets, etc.), you might notice some elements change in color or style when you scroll. This is because the `:hover` style fires on scroll. This is a problem. The solution is to use the `hover` media query. According to Mozilla Developer Network:

> The hover CSS media feature can be used to test whether the user's primary input mechanism can hover over elements.




```
@media (hover: hover) {
  .element:hover {
    color: #222;
  }
}
```

With this, we prevent `:hover` styles from firing for mobile and tablet users. At the time of writing, this feature is supported in all major browsers. The good thing is that when you active device mode in Chrome, it will be considered a touch screen, so you can test the `hover` media query there.

## Using CSS to Show Potential Errors

There is no direct way to display potential errors in CSS. However, some clever folks have devised workarounds that enable us to debug incorrect usage of HTML and CSS. Let's explore some of them.

## Using a CSS Class Out of Context

Let's say you've built a design system with your team, and you want to lint errors related to incorrect usage of a component in the design system. In his methodology named Inverted Triangle CSS (ITCSS), Harry Roberts uses the following classes to create warnings about incorrect usage of CSS classes.

```
<div class="o-layout">
  <div class="o-layout__item"></div>
  <div class="o-layout__item"></div>
  <div class="o-layout__item"></div>
</div>
```

The `.o-layout` class is for an element that acts as a layout wrapper. The `.o-layout__item` class should only be applied to elements within a parent that has the `.o-layout` class. The following usage would be incorrect:

```
<div>
  <div class="o-layout__item"></div>
</div>
```

The element with the `.o-layout__item` class shouldn't live on its own like this. We can debug this very easily:

```
.o-layout__item {
    /* Show a warning outline by default. */
  outline: solid 5px yellow;
}

.o-layout .o-layout__item {
    /* Remove the outline when item is in .o-layout. */
  outline: none;
}
```

Also, we can detect whether `.o-layout__item` is a direct child of `.o-layout`.

```
.o-layout > :not(.o-layout__item) {
    outline: solid 5px yellow;
}
```

## Adding `width` or `height` Attributes to Elements

Generally speaking, `width` and `height` attributes are not recommended for any HTML elements, except `img`.

```
:not(img):not(object):not(embed):not(svg):not(canvas)[width],
:not(img):not(object):not(embed):not(svg):not(canvas)[height] {
    outline: solid 5px red;
}
```

Going further, you can use Gaël Poupard's browser extension, a11y.css, which shows different advice, warnings, and errors.

# Acknowledgements

The book idea started as a note in April 2020. I asked Kholoud, my wife, what do you think about writing a book about debugging CSS? I told her that it will be a very short one (60 pages max). Seven months later, the book has 300 pages. Kholoud was the first person to support the book idea, and she insisted that I should move on with this, and here we are. Thank you, my dearest person!

The first person that encouraged me from the community is Mr. John Allsopp. He invited me to talk at Web Directions conference about the book topic and was one of the first supporters. Thank you very much!

I want to thank is Geoffrey Crofte. He was helpful and kind enough to proofread the whole book, and highlighting a lot of fixes. Thank you very much!

Finally, I would like to thank Bram Van Damme, who reviewed the very first draft of the book. He highlighted some important things that I should improve. Thank you, Bram!

I reduced the time I spend on debugging and fixing CSS bugs from hours to minutes. In this book, I will explain everything I learned about debugging and finding CSS issues.

## About the author

A Digital Product Designer and Front-End Developer from Palestine. He enjoy working on large scale Product Design and Front End Projects which involves solving complex design problems. He writes extensively on CSS, Accessibility and RTL (right to left) text styling.

> CSS is sadly an increasingly undervalued tool for front end developers, in no small part because developers find debugging CSS challenging. Yet, until now there's been little in depth published on debugging CSS. Ahmad Shadeed's book is long overdue, and I can't recommend it highly enough for any front end developer.
>
> **John Allsopp — Web Directions**

> Ahmad gathers in this practical book a bunch of CSS bugs encountered on a daily basis by all levels of developers, and how to solve all of them. A book that would have saved me hours of research in my early days, and well, still today! Well done!
>
> **Geoffrey Crofte — UX Designer**