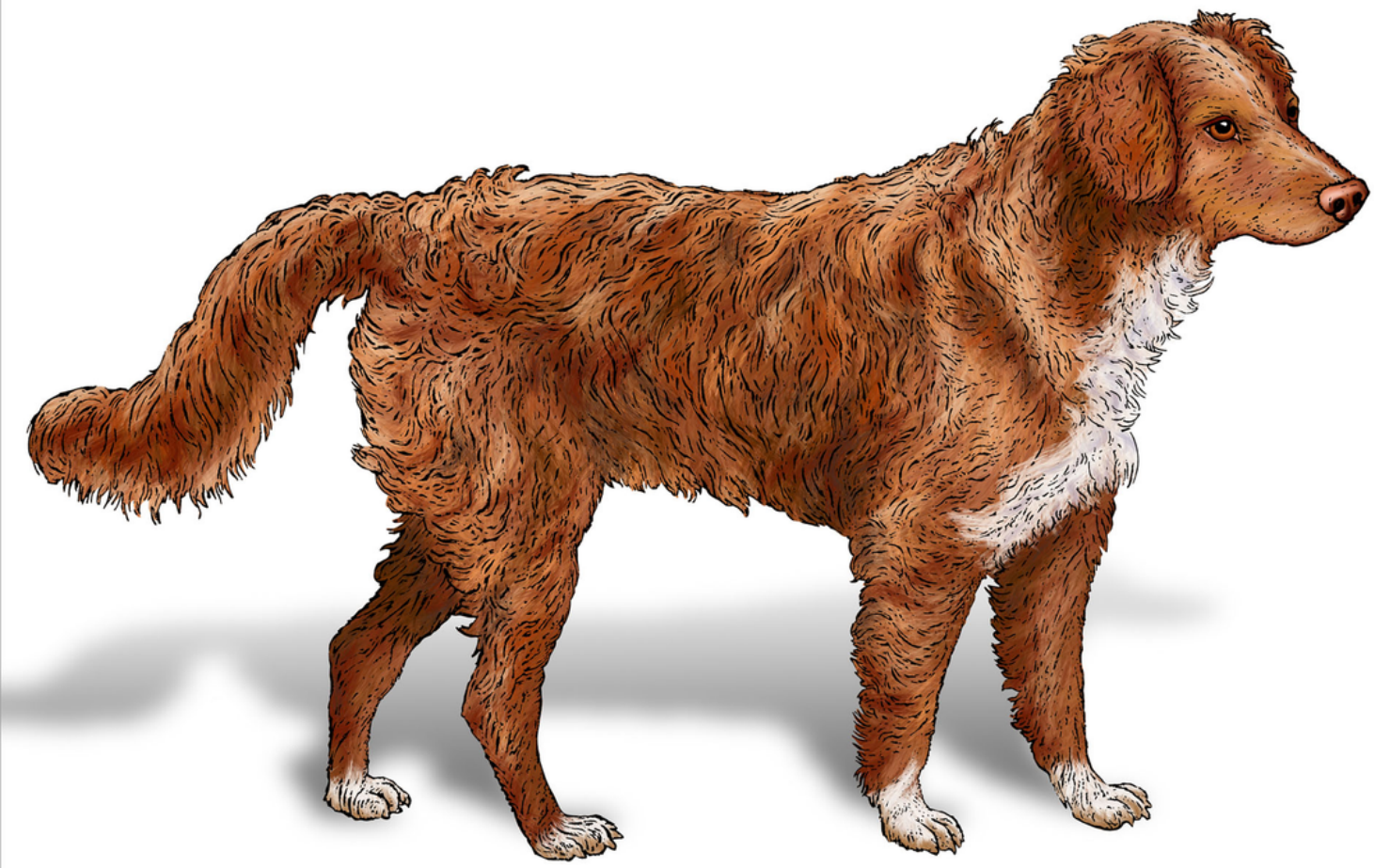
O'REILLY®

Web Accessibility Cookbook

Creating Inclusive Experiences



Manuel Matuzović

Praise for *Web Accessibility Cookbook*

Web accessibility can be daunting, but this book is not! Full of clear examples and explanations, it's very helpful for quick answers on what pattern is best to use, with good explanations for those who want to know more.

—Sara Joy Wallén, Web Developer, Pirate Ship Software GmbH

This is a much-needed and important book: the way Manuel explains the foundations of web accessibility is concise and straightforward and you are left with both practical solutions to the most common challenges as well as a solid understanding of what it takes to make any website accessible.

—Matthias Ott, Independent Web Developer

If the question is "What do I need to know about accessibility on the web?" the answer is Manuel's Web Accessibility Cookbook. This book gets straight into the code. I particularly like the problem/solution/discussion structure. The problems are laid out succinctly, and I appreciate being told why I should care before we get into the details of solving things with code. You can tell me that links should be underlined, but I'll actually listen when you tell me that a fellow human being with low vision can have trouble distinguishing links from regular text and underlining solves it. I always want to know the problem first and the literal structure of this book enforces that.

—Chris Coyier, CodePen, ShopTalk

I wish I'd had this book when I got started in accessibility. To build accessible websites, you need to understand how people use the web and get the implementation details right. This book combines the two with focused, hands-on advice. It is the perfect companion to formal accessibility standards.

—Hidde de Vries, Freelance Accessibility Specialist

Web Accessibility Cookbook helps demystify web accessibility through practical recipes for developers of all levels. Bridging coding and human-centered design, it fosters inclusiveness in every project—a must-read for those committed to digital accessibility and to building a more equitable world.

—Carie Fisher, Senior Accessibility Program Manager, GitHub

Web Accessibility Cookbook is like a helpful and knowledgeable colleague, providing just the right amount of information and explanation for all your accessibility questions. It's a deeply pragmatic book, and whether you are a new developer or a seasoned expert, Manuel has created a reference you'll want to keep close to hand as you develop for the web.

—Rachel Andrew, Content lead, Chrome Developer Relations, Google

Web Accessibility Cookbook

Creating Inclusive Experiences

Manuel Matuzović

Beijing • Boston • Farnham • Sebastopol • Tokyo

Web Accessibility Cookbook

by Manuel Matuzović

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Amanda Quinn
- Development Editor: Sarah Grey
- Production Editor: Aleeya Rahman
- Copyeditor: Dwight Ramsey
- Proofreader: Stephanie English
- Indexer: BIM Creatives, LLC

- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- June 2024: First Edition

Revision History for the First Edition

- 2024-06-13: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098145606 for release details.

978-1-098-14560-6

[LSI]

Dedication

For Vera, Sne, Claudi, Hanni, Olivi, and Phili

Foreword

Are you looking for a book that explains why you should care about web accessibility?

This is not that book.

Manuel Matuzović has too much respect for your intelligence to waste time trying to convince you of something you already know. You already know that web accessibility is important.

What you really need is a guidebook, a handy companion to show you the way through a tangled landscape.

This is that book.

If you want, you can read it cover to cover. That's what I did, and I enjoyed every moment of the journey.

But you might not have time for that. That's okay. The way that this book is subdivided means you can deep into any chapter and it will make perfect sense by itself. It really is like a cookbook. Every chapter is like a standalone recipe.

Whether you read this book linearly or dip in and out of it is up to you. Either way, Manuel is going to massage your brain until

something new takes shape in there. An understanding. Not just an understanding of web accessibility, but of the very building blocks of the web itself.

See, that's the sneaky trick that Manuel has managed with this book. It's supposed to be an accessibility cookbook but it's also one of the best HTML tutorials you'll ever find. Come for the accessibility recipe; stay for the deep understanding of markup.

Best of all, Manuel manages to do all this without wasting a word. Again, he has too much respect for you to waste your time. The only unnecessary words in your Accessibility Cookbook are the ones you're reading now. So I'm going to follow Manuel's example, respect your time, and let you explore this magnificent book for yourself.

Enjoy the journey!

Jeremy Keith

Co-founder of Clearleft and author of HTML5 For Web Designers

Brighton, England

February 2024

jeremy@adactio.com

Preface

A few years ago, I gave a talk about web accessibility at a meetup. Afterward, one of the attendees asked me why I specialize in accessibility: is it because I or someone close to me has a disability? When I answered “No,” they looked at me in surprise. They didn’t understand how someone not personally affected could be interested in making accessible websites. I explained that it’s because I care about people and about quality.

As developers, we deal with practical challenges every day. Often, we’re so focused on the technical part of development that we forget the real purpose of our jobs. It’s not solving technical issues; it’s *building products for human beings*. What we do is very technical at its core, but the consequences of our decisions are very human: they strongly affect people’s lives, their access to information, and their participation in society.

Just like people have different preferences, needs, abilities, disabilities, and habits, their approaches to accessing and consuming web content are also different. From the moment I understood that, making websites accessible wasn’t merely an option for me: it became mandatory. Caring about accessibility

means accounting for diversity and, thus, providing access to as many people as possible. That sounds obvious (because it is!), but sometimes, it's necessary to state the obvious to embrace it.

What makes accessibility technically interesting, besides its human nature, is its complexity, range, and interrelationship with other disciplines on the web. To make accessible websites, you need a core knowledge of user interface (UI) design, user experience (UX), usability, performance, content strategy, search-engine optimization (SEO), and security. A website with poor performance is inaccessible, bad UX usually means bad UX for everyone, poorly written HTML is bad for SEO and accessibility, and so on.

Accessibility doesn't just touch on the different disciplines of web development and design–it inherently connects them. Therefore, doing it well requires knowledge and interest in a broad range of disciplines. That's challenging, and many web developers see accessibility as a daunting burden. However, clearing that hurdle can be exciting and gratifying, no matter what stage you've reached in your career.

This book bridges the gap between the technical nature of web development and its impact on human beings. More than 70 recipes describe how to build the most common patterns on the

web accessibly. Each chapter outlines problems, provides solutions written in code, and explains how different pathways might affect users. You will learn how to write accessible frontend code and, most importantly, why you should. In the end, web accessibility will be less obscure to you and you'll be equipped to build and test your own accessible solutions.

My goal with the *Web Accessibility Cookbook* is to make you as excited about accessibility as I am and to give you the right tools to make the web a more inclusive place for everyone.

Who This Book Is For

This book is for anyone who writes frontend code: frontend developers, UX engineers, full stack and backend developers. It doesn't matter if you're new to web development or have been writing HTML, CSS, and JavaScript for over a decade.

You want to learn how to structure pages and components with HTML and how semantic elements affect user experience. You know CSS and want to learn how to style your websites in a way that serves your users. You are interested in ARIA and how to use it efficiently to improve the experience of interactive components written in JavaScript.

The *Web Accessibility Cookbook* provides everything you need to know to create accessible sites, pages, and components. It starts at a high level, explaining how to structure documents, then zooms in to discuss general topics like using links, buttons, tables, and forms in depth. There is a strong focus on HTML because it's the foundation of any accessible website, but the book also contains many components powered by JavaScript, like toggles, accordions, modals, filters, and navigations.

This book is for you if you want to not just copy and paste solutions, but really understand how they work and how they

benefit your users.

What's in This Book/Organization

The *Web Accessibility Cookbook* focuses on the technical side of web accessibility. You'll learn how to build common patterns written accessibly in HTML, CSS, and JavaScript. You'll also start to understand how good and bad practices affect people, especially those with disabilities. The book doesn't discuss the medical, social, or socioeconomic aspects of accessibility. It covers a variety of disabilities, like visual, motor, and learning disabilities, but not all of them, which would go beyond the constraints of this book. If you want to learn more about accessibility and the intersection between accessibility and technology, you may want to have a look at the following titles:

- *Accessibility for Everyone* by Laura Kalbag
- *Disability Visibility* edited by Alice Wong
- *Against Technoableism* by Ashley Shew
- More recommendations on *a11yproject.com*

Each chapter in this book stands on its own. You can read it from start to finish or jump directly into a specific topic. I've picked the problems and solutions based on my personal

experience auditing websites. Each recipe is oriented toward practicality and contains common frontend patterns and solutions for typical issues. You'll find references to other recipes and further resources as well.

Here is a brief overview of the content.

Chapter 1 focuses on those parts of your websites that recur and are similar or identical on every page. You will learn how to set up the `<head>` and create a base structure in the `<body>`.

In Chapter 2, you leave the base structure of your website and move into the page itself. The foundation of a well-designed page is grouping elements, landmarks, and headings. You learn how to use `<section>`, `<nav>`, or `<article>` efficiently and how to combine them with headings to create a sound document outline.

Hyperlinks are the basis of the World Wide Web. That is why Chapter 3, which is all about linking content, is one of the book's most extensive chapters. It analyzes the characteristics of the `<a>` element and helps you apply it efficiently. You'll learn how to link different types of content, images, and groups of elements. One recipe focuses on client-side routing and what

to consider when linking pages in a single application. Another addresses how to visually style links.

Chapter 4 is similar to the previous chapter, except that it puts the `<button>` element in the spotlight. You'll learn different techniques for labeling buttons and how to use them with the most common ARIA attributes.

In Chapter 5, we move from HTML to CSS and discuss color, contrast, animation, units, and sizes. You'll learn how to write CSS in a way that respects users' preferences.

Keyboard accessibility is an important topic in this book. Chapter 6 outlines everything you need to know about focus styling, focus management, and DOM order.

Almost every website has a main navigation. Chapter 7 dissects a typical site navigation and explains every part in detail, explaining why certain semantic elements can be useful for screen reader users. You'll learn how to create responsive navigation with submenus and understand the difference between navigations and menus.

There are different ways of hiding content in CSS and HTML. Chapter 8 discusses their pros and cons. You'll learn how to create disclosure widgets and accordions. The chapter also

compares the native `<details>` element with custom solutions.

Chapter 9 focuses on a complex topic: forms. It starts with general best practices for creating forms and gets more specific with every recipe. You'll learn the most important aspects of form design: labeling, description of form elements, error management, and grouping.

In Chapter 10, you'll learn how to build a filter form from start to finish. You'll also be introduced to dynamic feedback for screen reader users, pagination, and sorting.

Tables are misused so much that many developers are afraid of working with them. Chapter 11 demystifies the `<table>` element and presents best practices and guidance for using it. It also explains how to sort tables and combine them with interactive elements.

Custom elements are an exciting standard and, paired with other APIs, a powerful tool for creating web components. Chapter 12 explains everything you must consider regarding accessibility when working with them.

Trust is good; control is better. Chapter 13 introduces you to automatic testing and debugging tools that help you identify,

debug, and fix accessibility issues.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`*Constant width*`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`**Constant width bold**`

Shows commands or other text that should be typed literally by the user.

`*Constant width italic*`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples and Supported Software

Code samples for every recipe are available at *accessibility-cookbook.com* and on GitHub.

The following operating systems, browsers, and screen readers were used when developing and testing the code samples for this book:

- macOS 13.4.1
- Android 13
- Windows 11

- Chrome 121
- Safari 16.5.2
- Firefox 122
- VoiceOver on macOS and iOS
- TalkBack on Android
- NVDA 2023.3.2
- JAWS 2023.2307.37

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Web Accessibility Cookbook* by Manuel Matuzović (O'Reilly). Copyright 2024 Manuel Matuzović, 978-1-098-14560-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O’Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-889-8969 (in the United States or Canada)
- 707-827-7019 (international or local)
- 707-829-0104 (fax)
- *support@oreilly.com*
- *https://www.oreilly.com/about/contact.html*

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/web-accessibility-cookbook*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Acknowledgments

I wouldn’t have thought of writing a book on my own. Somehow, Amanda Quinn found me and convinced me to embark on this adventure. I’m grateful for that, because writing this book was an experience I wouldn’t have wanted to miss. With Amanda came Sarah Grey into my life as an author. She is an editor, as one would wish. She managed to deal with my chaotic scheduling and keep me on track without putting me under pressure. Her feedback was always clear and constructive, which shaped this book and helped me improve as a writer.

How do you know if the thousands of words you’ve written made any sense? If you have great technical reviewers, they will tell you—mine were extraordinary. Adrian Roselli, James Scholes, Joschi Kuphal, Lea Rosema, Matthias Ott, Sara Joy Wallén, and Sonja Weckenmann challenged my recipes and asked the right questions to help me improve them.

My knowledge about accessibility didn’t come from just anywhere. I learned everything I know from smart, skilled, and

talented people like Adrian Roselli, Alice Boxhall, Bruce Lawson, Carie Fisher, Dennis Lembrée, Eric W. Bailey, Eric Eggert, Heydon Pickering, Hidde de Vries, Joschi Kuphal, Karl Groves, Kitty Giraudel, Leonié Watson, Marco Zehe, Marcy Sutton, Scott Vinkle, Steve Faulkner, Rian Rietveld, Rob Dodson, Scott O'Hara, Val Head, and many more.

My mentor, Aaron Gustafson, taught me a lot, guided me in the right direction, and helped me build the confidence to blog and speak. Vitaly Friedman and the *Smashing Magazine* team, Marc Thiele from beyond tellerrand, and all the other conference organizers gave me access to a larger audience and allowed me to grow.

This book wouldn't exist without the Austrian accessibility scene. People like Jo Spelbrink, Maria Putzhuber, Wolfram Huber, and Wolfgang Leitner inspired and supported me in my early days and helped me evolve. They taught me about web accessibility, introduced me to the scene, and gave me a stage at the A-Tag conference. Michael Rederer put a lot of trust in me, guided me, and helped me become more professional. Werner Rosenberger enabled me to improve my accessibility testing skills.

My friends all over the world, with whom I've spent endless hours writing and talking about web accessibility on social media, via email, and at conferences, are the reason I enjoy sharing content and keep doing it. My friend and former office colleague Bernhard Steinbrecher unknowingly helped me build the confidence to write.

Finally, I want to thank the people who always trusted me and, by doing that, made the most significant contribution to this book: my mother Vera, my sister Sne, my fiancée Claudi and our wonderful little girls, Johanna, Olivia, and Philippa. You are everything to me and the reason this book exists.

Chapter 1. Structuring Documents

This book focuses on writing accessible components, but accessibility begins at the very first line of your HTML document. Your components live on a page, and your page is part of a document. Several elements, especially in the `<head>` of your document, affect accessibility.

1.1 Define the Natural Language

Problem

If a page doesn't contain an explicit definition of the natural language it's written in, software may not be able to translate content correctly. The term *natural language* refers to the language you use for the content on the page, not the programming language. This lack of information can result in faulty translations, wrong formatting, and content being hard to understand for screen reader users.

Solution

You can define the natural language of a page by using the `lang` attribute on the `<html>` element. See Example 1-1.

Example 1-1. English defined as the natural language of the page

```
<!DOCTYPE html>
<html lang="en">
</html>
```

You can also define a specific dialect of the base language. See Example 1-2.

Example 1-2. British English defined as the natural language of the page

```
<!DOCTYPE html>
<html lang="en-GB">
</html>
```

The `lang` attribute is global, meaning you can use it on any element, although it may not affect some of them. It can be helpful if a page is written in one language but contains text passages or even single words in other languages. See Example 1-3.

Example 1-3. Transliterated Japanese in Latin script used on a page written in English

```
<!DOCTYPE html>
```

```
<!DOCTYPE html>
<html lang="en">
  <head></head>
  <body>
    <p>
      The Wind-Up Bird Chronicle (<span lang="ja
      </span>) is a novel published in 1994 by J
    </p>
  </body>
</html>
```

Use the `lang` pseudoclass to adjust the typography and layout for specific languages. See Example 1-4.

Example 1-4. Selecting all elements in the Serbian language

```
:lang(sr) {
  font-family: 'Cyrillic font', sans-serif;
}
```

Discussion

Assistive technology and other software may not be able to determine the natural language of a page automatically. Certain features in HTML and Cascading Style Sheets (CSS) rely on that information to help localize software content and provide an

excellent overall user experience. You must set the language of each page programmatically and explicitly by using the `lang` attribute on the `<html>` element, as shown in Example 1-1. For passages of text written in a different language than the primary language of the page, you can also use the attribute, as shown in Example 1-3. That allows screen readers to improve pronunciation by switching voice profiles accordingly for certain words or sentences. Try to do this sparingly because switching voice profiles can be annoying because it interrupts the reading flow. For well-established foreign words, it might not be necessary. Examples in German are English words like "Download," "Workshop," or "Link."

Usage

The value of the `lang` attribute must be a valid BCP 47 language tag, composed from one or more subtags. A *subtag* is a sequence of alphanumeric characters distinguished and separated from other subtags by a hyphen.

The *language subtag* is a 2- or 3-character code that defines the primary language: for example, `en` for English, `de` for German, or `fr` for French, as shown in Example 1-5.

Example 1-5. Spanish defined as the natural language of the page

```
<html lang="es"></html>
```

The optional *script subtag* is a 4-character code that defines the writing system used for the language, as shown in Example 1-6.

Example 1-6. A name in Cyrillic script next to the same name in Latin script marked as such

```
Никола Јокић (<span lang="sr-Latn">Nikola Jokić<
```

The optional *region subtag* is usually a 2-character country code written in all caps and defines a dialect of the base language, as shown in Example 1-7.

Example 1-7. Austrian German defined as the natural language of the page

```
<!DOCTYPE html>
<html lang="de-AT">
</html>
```

You should use the 2-character primary language code and add region subtags only when it is necessary to differentiate content in different dialects that may not be mutually understandable. At least for screen reader users, not adding region subtags

shouldn’t make a difference because they’re typically ignored by the software.

You can find a list of all tags and subtags in the BCP 47 language subtag lookup.

Benefits

The `lang` attribute is powerful and affects many aspects of web accessibility and user experience in general. These include:

Assistive technology

Speech synthesizers that support multiple languages adapt their pronunciation and syntax to the language of the page, speaking the text in the appropriate accent with proper pronunciation. For a page with German content where the language of the page is set to English (`lang="en"`), the screen reader software may pick an English synthetic voice profile and read the German text with English pronunciation. If you set no language, screen readers may fall back to the users’ default system setting, which might not be appropriate. The result can be hard to understand, confusing, or even completely wrong. All screen readers support numerous languages. Some software switches language automatically, while for others users have to install and configure language voices or packs manually.

The attribute definition also allows Braille translation software to optimize the output and prevent it from erroneously creating Grade 2 Braille contractions.

Translation

Translation tools like Google Translate use the information from the `lang` attribute to translate content on the page. While this kind of software is usually good at automatically detecting the language of the page, a mismatch between the actual language and the defined language can yield unexpected and unwanted translations.

Quotes

Quotation marks may change depending on the natural language of the page. For example, English uses different quotation marks than German or French, and the correct `lang` helps browsers pick the proper glyphs, as illustrated in Examples 1-8, 1-9, and 1-10.

Example 1-8. Automatic quotation marks using the `<q>` element in English

```
<p lang="en">
  <q>A quote in English.</q>
```

```
  </p>

  <!-- Results in: “A quote in English.” -->
```

Example 1-9. Automatic quotation marks using the `<q>` element in German

```
<p lang="de">
  <q>Ein Zitat auf Deutsch.</q>
</p>

<!-- Results in: „Ein Zitat auf Deutsch.“ -->
```

Example 1-10. Automatic quotation marks using the `<q>` element in French

```
<p lang="fr">
  <q>Une citation en français.</q>
</p>

<!-- Results in: « Une citation en français. » -
```

Hyphenation

`lang` may affect hyphenation in CSS. See Example 1-11.

Example 1-11. A paragraph with a maximum width of 28 characters and hyphenation turned on

```
p {
  max-width: 28ch;
  hyphens: auto;
}
```

In Examples 1-12, 1-13, and 1-14, you can see how the same paragraph written in German, given a different `lang` attribute value, renders differently in Google Chrome. Words either don't break at all or break at different positions. Only the first and the second examples are correct. It's worth noting that browsers behave differently.

Example 1-12. Correctly hyphenated German text in a paragraph defined as German

```
<p lang="de">
  Weit hinten, hinter den Wortbergen, fern der L
  leben die Blindtexte. Abgeschieden wohnen sie
  des Semantik, eines großen Sprachozeans. Ein k
  fließt durch ihren Ort und versorgt sie mit de
</p>

<!-- Results in:
  Weit hinten, hinter den Wortbergen,
```

```
  fern der Länder Vokalien und Konso-
  nantien leben die Blindtexte. Abge-
  schieden wohnen sie in Buchstab-
  hausen an der Küste des Semantik,
  eines großen Sprachozeans.
-->
```

Example 1-13. No hyphenation of German text in a paragraph defined as English

```
<p lang="en">
  Weit hinten,…
</p>

<!-- Results in:
  Weit hinten, hinter den Wortbergen,
  fern der Länder Vokalien und
  Konsonantien leben die Blindtexte.
  Abgeschieden wohnen sie in
  Buchstabhausen an der Küste des
  Semantik, eines großen
  Sprachozeans.
-->
```

Example 1-14. Wrong hyphenation of German text in a paragraph defined as French

```
<p lang="fr">
  Weit hinten,…
</p>

<!-- Results in:
  Weit hinten, hinter den Wortbergen,
  fern der Länder Vokalien und Konso-
  nantien leben die Blindtexte. Abges-
  chieden wohnen sie in Buchstabhau-
  sen an der Küste des Semantik, eines
  großen Sprachozeans.
-->
```

Font selection

Browsers may select language-appropriate fonts for displaying details in ideographic characters that vary from language to language, such as Chinese, Japanese, and Korean (known as the "CJK languages.")

Search Engine Optimization (SEO)

Properly defining the natural language can improve the quality of search results by helping search engines with localization.

Form controls

In some browsers, the `lang` attribute also affects the formatting in form controls. For example, Firefox shows the correct decimal characters in number input fields depending on the language.

1.2 Describe the Document

Problem

Screen reader users navigating a website can't always tell which page they're on. They may not understand what a page is about or notice that content has changed if the page's title isn't set correctly.

Solution

You can name pages using the `<title>` element in HTML. The title must be unique and must describe the topic or purpose of each page concisely. See Example 1-15.

Example 1-15. A succinct and descriptive page title

```
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <title>Products - Johanna's Toy Store</title
  </head>
</html>
```

For social media previews, you can optionally use an open graph `meta` tag to include more or different information, as shown in Example 1-16.

Example 1-16. A catchier page title for social media previews

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Products - Johanna's Toy Store</title
    <meta property="og:title"
          content="Find dolls, toy cars, and mor
  </head>
</html>
```

Adding context depending on the state of the page can be helpful (see Examples 1-17, 1-18, 1-19, and 1-20).

Example 1-17. The title includes the current step and the total number of steps in a checkout process

```
<title>Checkout (step 3 of 4) - Johanna’s Toy St
```

Example 1-18. Number of validation errors in the title of a sign up page

```
<title>2 errors - Sign Up - Johanna’s Toy Store<
```

Example 1-19. Number of results in a search page’s title

```
<title>21 results for term “crocodile” - Johanna
```

Example 1-20. The title indicating the current search results page

```
<title>Page 2 - Product Search Results - Johanna
```

Discussion

Sometimes it takes work to get oriented on a website, such as when users land on a page coming from an external resource

like a search engine or if page changes happen unexpectedly or unannounced. That's especially true for single-page applications (SPAs), where page navigation works differently from most sites.

The page's title is one of the essential elements in an HTML document, and users benefit from a well-formed and descriptive page title.

Screen reader users can use shortcuts to announce the page title. For example, if they click a link in an SPA and content changes, but there's no announcement that changes have happened, they can use a shortcut to get oriented and check if they've landed on a different page. You can try it yourself by using one of the shortcuts in Table 1-1.

Table 1-1. Different ways to announce the page title with screen readers.

Screen reader	Command	Announcement
JAWS	`Ins` + `T`	Page title
NVDA	`Ins` + `T`	Page title
VoiceOver macOS	`VO` + `Shift` + `I`	Page summary, including page title
VoiceOver macOS	`VO` + `F`	Page title

NOTE

By default, `VO` stands for the combination of pressing `Control` and `Option` at the same time in VoiceOver on macOS. Alternatively, you can map `VO` to `Caps Lock` in the VoiceOver Utility settings.

There are other reasons for writing good page titles: they serve as labels for bookmarked pages/favorites, and search engines use the title in their results pages. Social media sites, chat and mail applications, and similar software use the title in link previews when no other title is specified. Please note that the open graph meta tag shown in Example 1-16 is no alternative to

the `<title>` element. Whether a site or application interprets the open graph title is up to the site. Ideally, the native title's content should be good enough to serve all purposes.

There are several best practices you should follow when writing page titles:

The title must be unique

The `<title>` serves as the label in tabs or browser windows. Unique titles help to distinguish one page from the other if multiple tabs of the same site are open. A common issue is that the title is the same on all pages (see Example 1-21). The result is shown in Figure 1-1.

Example 1-21. Bad practice: Three different pages with the same title

```
<!-- products.html -->
<title>Johanna's Toy Store</title>

<!-- team.html -->
<title>Johanna's Toy Store</title>

<!-- contact.html -->
<title>Johanna's Toy Store</title>
```

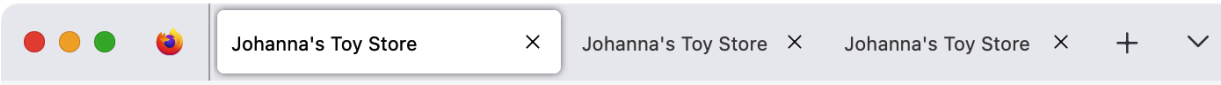


Figure 1-1. It's impossible to tell these pages apart by looking at the tabs

Use unique titles to communicate the purpose of the page and improve orientation (see Example 1-22 and the result in Figure 1-2).

Example 1-22. Three different pages, each with a unique title

```
<!-- products.html -->
<title>Products - Johanna's Toy Store</title>

<!-- team.html -->
<title>Our Team - Johanna's Toy Store</title>

<!-- contact.html -->
<title>Get in Touch - Johanna's Toy Store</title>
```

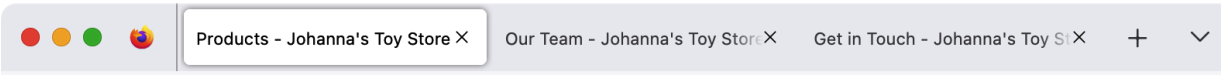


Figure 1-2. By looking at the tabs, you can tell what each page is about

That applies to websites with multiple pages but also to SPAs. You might have to take extra steps in an SPA, but if the user navigates to a different route, the page title must change as

well. Hidde De Vries explains how to do that in "Accessible page titles in a Single Page App".

The title should be concise

The title should be concise and should accurately describe the purpose of the page. It's the first information a screen reader user gets when they're accessing a page. They don't need a detailed summary of the page's content, but a succinct description.

Another reason to constrain the length of the title is that it usually gets cut off in search result pages at a certain character length (approximately 50 to 60 characters).

The title should be descriptive

When you title a page, do it with the user in mind. While SEO is important, the user experience is much more important. The title should describe the page's purpose and must not include marketing or SEO terms solely to improve page rankings.

The relevant information comes first

The title should start with the page's name, followed by the name of the site, company, or organization, as shown in

Example 1-15. Putting the relevant information first reduces repetition for screen reader users who visit multiple pages on a site, because they get the unique page-specific details first. This way of arranging content makes scanning and identifying pages easier when numerous tabs are open, as shown in Figure 1-3.

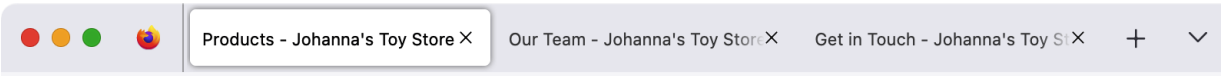


Figure 1-3. The name of the page is the first information in the browser's tab

Don't put the organization's name first (as in Example 1-23) because the relevant information might get cut off, as you can see in Figure 1-4.

Example 1-23. Bad practice: Name of the site followed by the name of the page

```
<title>Johanna's Toy Store - Products</title>
```

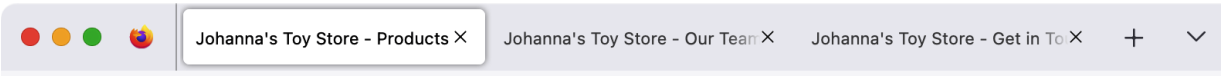


Figure 1-4. With the name of the site as the first information in the browser's tab, in some cases the name of the page is cut off

Context-dependent information

Sometimes it's helpful to add context or additional information. For example, if you split a page into multiple steps, the title should include the current step. Example 1-17 shows the title of the third of four steps in a checkout process. If there are validation errors in a form, you can indicate the number of errors (see Example 1-18). For search result pages, you can add the number of results or the current page, as shown in Examples 1-19 and 1-20.

1.3 Set the Viewport Width

Problem

Sometimes users want to zoom into a page because they can't read the text or want a closer look at something. They can't do that if the viewport settings prohibit zooming. That is especially problematic for people with low vision.

Solution

Configure the `viewport` meta tag in a way that allows for the most flexibility. Avoid restrictive settings.

The `meta` tag in Example 1-24 works for most websites and web apps. It's all you need to configure viewport settings to build a flexible, adaptive, responsive website.

Example 1-24. The page uses the available width of the device as the width for the viewport

```
<meta name="viewport" content="width=device-widt
```

There are specific properties or values you should *avoid*.

Setting the value of the `width` property to anything other than `device-width` can cause problems. If the defined width of the viewport is larger than the available width on the screen, content may overflow, resulting in horizontal scroll bars. Example 1-25 applies an explicit viewport width.

Example 1-25. Bad practice: Width of the viewport set to an absolute value

```
<meta name="viewport" content="width=500, initia
```

`maximum-scale` allows you to limit the maximum zoom level, which is `10` by default in most browsers. If you set it to `1`, you're disabling zoom in some browsers (see Example 1-26).

Example 1-26. Bad practice: Zoom disabled by setting the maximum scale to 1

```
<meta name="viewport" content="width=device-widt
```

The setting `user-scalable` defines whether zooming is allowed. Setting it to `no` or `0` disables zoom, as shown in Example 1-27.

Example 1-27. Bad practice: Zoom disabled by setting `user-scalable` to "no"

```
<meta name="viewport"
      content="width=device-width, user-scalable=
```

Discussion

Users must be able to customize their browsing experience based on their preferences and needs. Browsers offer different settings and features to support that. The ability to set larger font sizes or zoom in on the page is essential, but many websites disallow zooming on handheld devices.

Before responsive web design became a thing, websites were designed for large viewports, often using fixed width values like `960px` or `1024px` for the body or main content of the page. With the rise of smartphones and other handheld devices, this became a problem because pixel widths of the screens on these devices were usually much smaller than `960px`. The *initial containing block*, the rectangle in which the root element (`<html>`) lives, has the dimensions of the viewport. If the page is larger than the viewport, this can result in unintended layout wrapping, clipped content, and unpleasant scrollable bounds. That's why mobile browsers generally use a fixed width (typically `980px` to `1024px`) for the initial containing block. The resulting layout is then scaled down to fit the available screen space. That mitigates the issues, but it also means that the CSS pixel size on these pages will be much smaller, forcing users to zoom in.

On responsive websites, that's not an issue, since they've been built to work well on assorted viewports. However, you have to change the fixed width of the containing block on mobile devices to a width relative to the viewport's dimensions to work well with responsive designs. You can do that by using the viewport `meta` tag, as illustrated in Example 1-24.

`width=device-width` sets the viewport's width to the device's available width.

`initial-scale=1` ensures that the default zoom level is at 100%. This might not always be the default in all browsers. That's why I recommend setting it explicitly.

The fact that a page is responsive and optimized for small viewports doesn't mean users won't want to zoom. Adrian Roselli lists several reasons the ability to zoom is essential in his article "Don't Disable Zoom":

- The text may be too small for the user to read.
- The user may want to see more detail in an image.
- Selecting words to copy/paste may be easier for users when the text is larger.
- The user wants to crop animated elements out of the view to reduce distraction.
- The developer did a poor job of responsive design, and the user needs to zoom just to use the page.
- There is a browser bug that causes the default zoom level to be odd.
- It can be confounding for users when the browser interprets a pinch/spread gesture as something else.

Websites disabling zoom is a prevalent issue. According to the Accessibility chapter of the 2022 Web Almanac, 23% of desktop home pages and 28% of mobile home pages attempt to disable zoom. The report's author uses the term *attempt*, because some browsers, like Safari on iOS or Samsung Internet on Android, ignore the `maximum-scale=1` and `user-scalable=no` properties. Chrome and Firefox don't, but users can force zoom in their browser settings. In Firefox, find the browser settings, select "Accessibility," and activate "Zoom on all websites." In Chrome, find the browser settings, select "Accessibility," and check "Force enable zoom."

Justified reasons to disable zoom

On the average website, there's no good reason to turn off zoom. The same applies for app-like websites that resemble native apps. There are rare exceptions where the gestures for zooming would interfere with the website's functionality. An example would be a site that contains only an interactive map. If that's the case, it might be okay to disable the native zoom feature, but you must provide an alternative custom solution.

1.4 Optimize Rendering Order

Problem

The head of a document can contain various elements that serve different purposes: meta tags, scripts, links to other resources, and more. They can be in any order, but certain elements should come before others to ensure good loading performance. Inefficient asset loading prevents users from obtaining information quickly, if at all.

Solution

Web performance expert Harry Roberts suggests a specific order of elements within the `<head>` to ensure the best possible loading strategy, as shown in Example 1-28.

Example 1-28. The ideal order of elements in the `<head>`

```
<head>
  <!-- Character encoding -->
  <meta charset="UTF-8">

  <!-- Viewport meta tag -->
  <meta name="viewport" content="width=device-wi


  <!-- CSP headers -->
  <meta http-equiv="Content-Security-Policy" con
```

```
<!-- Page title -->
<title>Johanna's Toy Store</title>

<!-- preconnect -->
<link rel="preconnect" href="#" />

<!-- Asynchronous JavaScript -->
<script src="" async></script>

<!-- CSS that includes @import -->
<style>
  @import "file.css";
</style>

<!-- Synchronous JavaScript -->
<script src=""></script>

<!-- Synchronous CSS -->
<link rel="stylesheet" href="#">

<!-- preload -->
<link rel="preload" href="#" />

<!-- Deferred JavaScript -->
<script src="" defer></script>

<script src="" type="module"></script>
```

```
    <!-- prefetch / prerender -->
    <link rel="prefetch" href="#" />
    <link rel="prerender" href="#" />

    <!-- Everything else (meta tags, icons, open g
    <meta name="description" content="">
  </head>
```

Discussion

There are different fields within web design and development, like accessibility, usability, user experience, performance, and security. They all focus on different things, but they're not mutually exclusive. Accessibility, for example, overlaps with all of them. Improving the accessibility of form fields may result in a better user experience for everyone. If a website loads slowly, it affects users' experience and accessibility. A website that loads for too long or not at all on a slow connection is not accessible. Designing and building accessible websites means creating inclusive experiences without barriers that prevent people from interacting with the web. These barriers include physical, temporary, and situational disabilities and socioeconomic restrictions on hardware, bandwidth, and speed.

Getting the order of elements in the `<head>` right affects performance in general, but it also affects the rendering of specific elements that may contain critical information for users of assistive technology. HTML is parsed line by line, which means that a browser doesn't know that line 4 exists when line 3 hasn't finished parsing. If something blocks rendering early in a document, subsequent lines have to wait until the browser has finished parsing preceding lines. That makes the correct order of elements in the `<head>` crucial to web performance and accessibility.

Performance experts suggest several rules and optimizations, including:

- If something doesn't have to be in the `<head>`, remove it or put it in the `<body>`. That includes low-priority scripts, redirects, or any unnecessary payload.
- Self-host as much as possible and don't rely on third-party content delivery networks (CDNs). Harry Roberts explains why in his article "Self-Host Your Static Assets".
- Validate your HTML code, because invalid elements in the `<head>` can cause performance problems.
- Metadata about the page, like character encoding and information about the viewport, go first.
- Nothing render-blocking must come before the `<title>`.

- Synchronous JavaScript comes before CSS, because CSS blocks the execution of subsequent JavaScript.
- Avoid `@import` in CSS.
- SEO and social meta tags go last.

Harry Roberts has created a CSS file called *ct.css*, also available as a bookmarklet, that you can use to run tests against your `<head>` elements (see Figure 1-5).

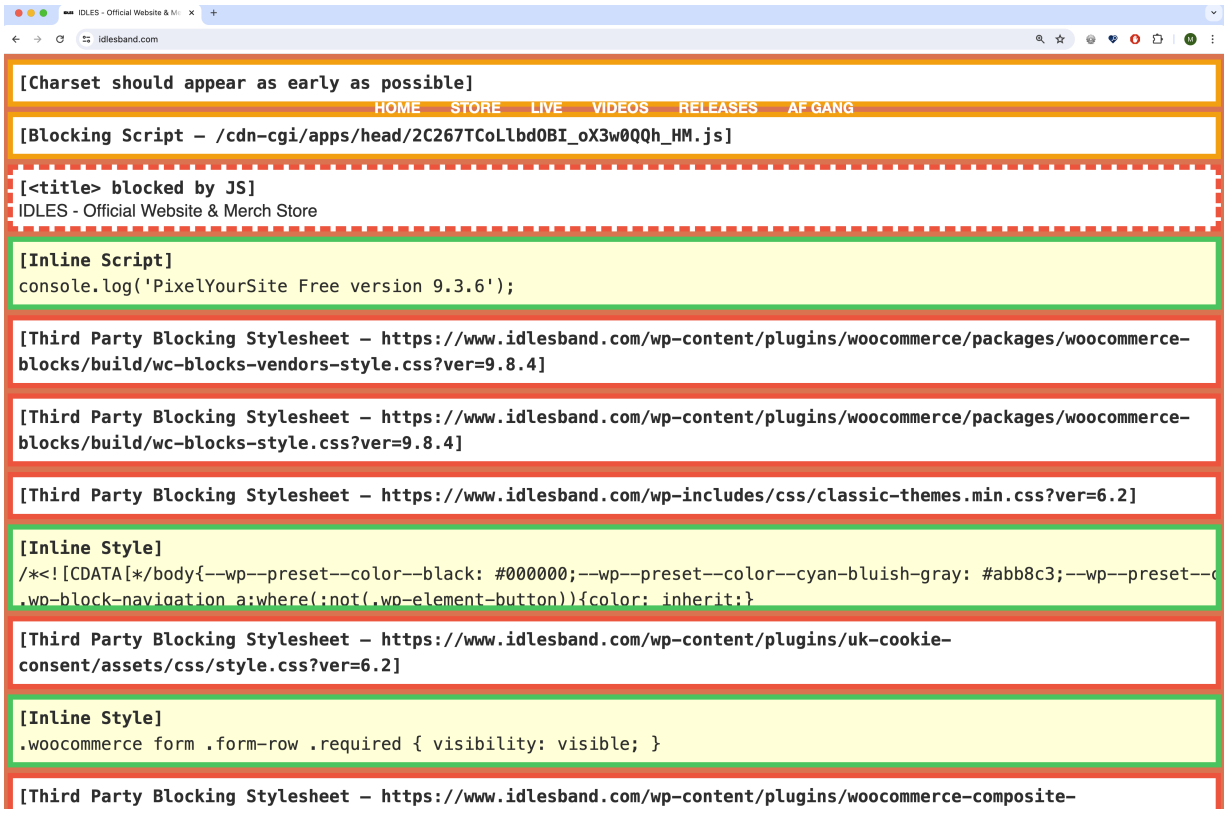


Figure 1-5. ct.css displaying messages on a page after running tests

1.5 Structure the Document

Problem

If a page doesn't contain enough semantic regions, users might not be able to understand how it's structured. That lack of semantic markup prevents them from using shortcuts to navigate more efficiently.

Solution

Use *landmarks*: regions that represent the organization and structure of a web page. They usually identify areas the user may want to access quickly.

Chapter 2, "Structuring Pages" focuses on page regions, but there are also common landmarks you will use across your site, like `<header>`, `<main>`, and `<footer>`. Every element in the page should be within one of these landmarks, as shown in Example 1-29.

Example 1-29. An exemplary structure of a web page

```
<!DOCTYPE html>
<html lang="en">
<head>

  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-wi
```

```
  <title>Products - Johanna’s Toy Store</title>
</head>
<body>
  <header> ❶
    <a href="/">
      Johanna’s Toy Store
    </a>

    <nav aria-label="Main"> ❷
      <ul>
        <li>
          <a href="/home">Home</a>
        </li>
        <li>
          <a href="/products" aria-current="page
        </li>
        <li>
          <a href="/team">Team</a>
        </li>
        <li>
          <a href="/contact">Contact</a>
        </li>
      </ul>
    </nav>


    <form role="search"> ❸
      <label for="search">Search</label>
```

```
        <input type="text" id="search">
      </form>
    </header>

    <main id="content"> ❹
      <h1>All products</h1>

    </main>

    <footer> ❺
      &copy; 2024
    </footer>
  </body>
  </html>
```

❶ The header of the site (banner landmark).

❷ The main navigation (navigation landmark).

❸ The site search (search landmark).

❹ The main content (main landmark).

❺ The footer of the site (contentinfo landmark).

Discussion

Using semantic elements within components and regions of a page is the foundation of any accessible website, but users can benefit from larger semantic groups, too. The page must communicate how it's structured and group common sitewide and page-specific elements. Landmarks in HTML help with that.

You can define landmarks with the appropriate HTML elements or use the `role` attribute when no element exists. The elements in Example 1-30 are semantically the same.

Example 1-30. Two banner landmarks

```
<!-- <header> with an implicit banner role -->
<header></header>

<!-- <div> with an explicit banner role -->
<div role="banner"></div>
```

TIP

Most semantic elements in HTML convey two bits of information: their accessible role and an accessible name. The role defines what kind of element it is: a button, link, image, etc. The accessible name is text by which software can identify a component, coming from the element's text content, another associated element like `<label>`, or an attribute like `aria-label`, `aria-labelledby`, `alt`, or `title`.

Follow the first rule of Accessible Rich Internet Application (ARIA) use and prefer elements with implicit roles over using the `role` attribute if browser support allows it. Older browsers and screen readers that don't support elements with implicit landmark roles once needed the additional explicit role, as shown in Example 1-31, but specifying both isn't necessary anymore.

Example 1-31. `<header>` with an additional explicit banner role

```
<header role="banner"></header>
```

There are different types of regions that serve other purposes in different contexts. Table 1-2 lists HTML elements, their corresponding ARIA roles, and the context in which they're exposed as landmarks.

Table 1-2. HTML landmarks and their ARIA roles

Element	ARIA role	Conditions
`header`	banner	Only in context of the body element; not when it's a descendant of `<article>`, `<aside>`, `<main>`, `<nav>`, or `<section>` (or any other element with their corresponding explicit roles).
`nav`	navigation	
`main`	main	
`section`	region	When it has an accessible name.
`form`	form	When it has an accessible name.
`search`	search	Or form with `role="search"`.
`aside`	complementary	

Element	ARIA role	Conditions
`footer`	contentinfo	Only in context of the body element; not when it's a descendant of `<article>`, `<aside>`, `<main>`, `<nav>`, or `<section>` (or any other element with their corresponding explicit roles).

Benefits

There are many reasons to use landmarks. The remainder of this section will explain several of those reasons in detail.

Orientation

Landmarks help screen reader users with orientation on the page. The software may announce landmarks when users enter or leave the enclosed content. They contain every item on the page to help users discover them.

Navigation

Screen reader users can jump from landmark to landmark using keyboard shortcuts or gestures, which provide a convenient way to skip to specific areas without interacting with the rest of the page (see Table 1-3).

In VoiceOver on iOS, you can select the "landmark" option in the rotor, which provides you with direct access to certain elements on the page, and use the swipe up and down gestures to navigate between landmarks. In TalkBack on Android, you can set the reading controls to "landmarks" and swipe up and down to navigate. In NVDA on Windows, you can press `D` or `Shift` + `D`, and in JAWS on Windows, `R` and `Shift` + `R` to do the same (see Table 1-3).

Table 1-3. Landmark navigation shortcuts

Screen reader	Command
NVDA	`D`
JAWS	`R`
Narrator	`D`
VoiceOver on iOS	Rotor
TalkBack on Android	Reading controls

It's mostly screen reader users who benefit from having landmarks, but also browser extensions like "Landmark Navigation via Keyboard or Pop-up" add keyboard shortcuts to the browser, providing access to landmarks for non-screen reader users. See Figure 1-6.

Overview

Screen reader users can list all landmarks on a page and access them directly (see Table 1-4).

Table 1-4. Shortcuts for listing all kinds of elements, such as landmarks

Screen reader	Command
NVDA	`Ins` + `F7`
JAWS	`Ins` + `Ctrl` + `R`
VoiceOver on macOS	Rotor

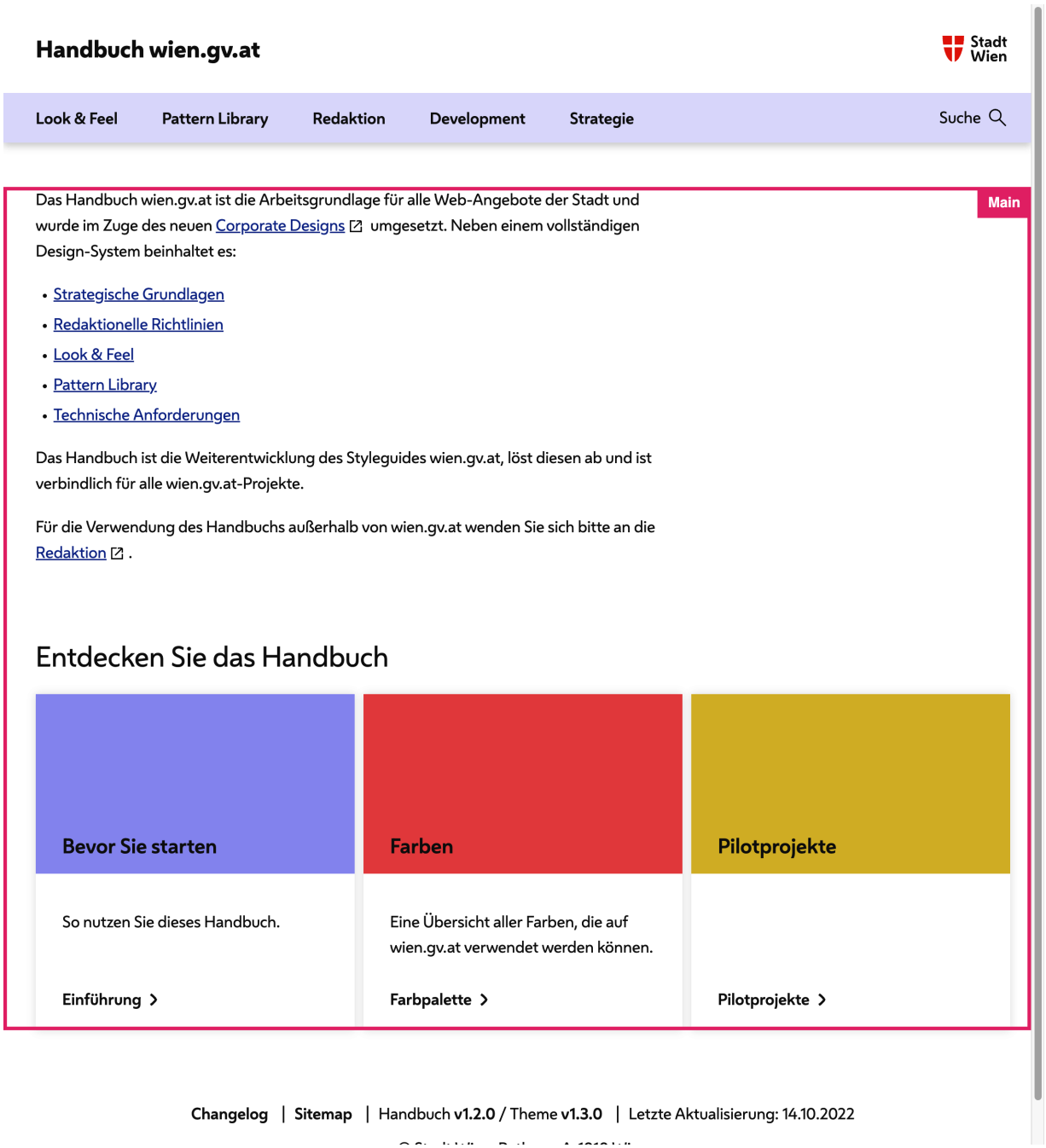


Figure 1-6. The `main` landmark on handbuch.wien.gv.at, highlighted by the "Landmark Navigation" browser extension

Site-Specific Landmarks

The three most relevant main landmarks are `<header>`, `<main>`, and `<footer>`.

banner landmark

The `<header>`, with its implicit `banner` role, contains mostly site-oriented rather than page-specific content. That's typically a logo, skip links, the main navigation, secondary navigations, a search widget, and other content that is relevant and visible on every page.

Not every `<header>` is a landmark. If it's nested inside `<article>`, `<aside>`, `<main>`, `<nav>`, or `<section>`, it's semantically similar to a `<div>` and not exposed as a landmark anymore. Having multiple `header` elements on a page is fine, but you should add only a single `banner` landmark.

Typically, you'll find `banner` landmarks at the beginning of the `<body>` in the document. Visually, it's usually at the top of the page. That is a common pattern but not a strict rule; it may also look like a sidebar. The position doesn't affect its semantic purpose. Just because it's located on the side doesn't mean its role has to change.

main landmark

The `<main>` element's implicit `main` role represents the page's core content. There should be only one visible main element on a page, and its ancestors should be limited to `html` and `body` to guarantee a hierarchically correct structure. If necessary, it's possible to wrap it in `<div>` elements.

If you're working with an SPA with multiple `<main>` elements on a page, hide all the inactive ones, as shown in Example 1-32. Having more than one visible and reachable main landmark on a page might confuse users and result in them missing content because they usually expect only one per page.

Example 1-32. Multiple `main` elements, but only one is visible

```
<main hidden>
  <h1>Home</h1>
</main>
<main>
  <h1>Products</h1>
</main>
<main hidden>
  <h1>Team</h1>
</main>
<main hidden>
  <h1>Contact</h1>
</main>
```

contentinfo landmark

The `<footer>` element's implicit `contentinfo` role also contains site-oriented content. That's typically copyright data, secondary navigations, and other links.

Similar to the `<header>`, the `<footer>` is only a landmark in the `<body>` context. If it's nested inside `<article>`, `<aside>`, `<main>`, `<nav>`, or `<section>`, it's not a landmark anymore. Having multiple `footer` elements on a page is fine, but you should add only a single `contentinfo` landmark.

See Also

- "H57: Using the language attribute on the HTML element"
- "Setting the page title in Angular"
- "React Helmet: Component for changing content in the head"

Chapter 2. Structuring Pages

In Chapter 1, you learned that landmarks can be useful for accessing major areas of the site, like the header, main content, or footer. Page-specific landmarks like the navigation (Recipe 2.1) or search (Recipe 2.2) can help users be even more efficient. Especially on complex pages with a lot of structured content and different elements, providing a shortcut to certain crucial parts of the UI can be of great help.

2.1 Create Navigation Landmarks

Problem

No matter how users access a website, they must be able to identify navigations. Otherwise, they may not be able to find the content they're looking for, orient, and navigate.

Solution

Identify navigations and enable quick access for significant groups of links, such as the main navigation (see Example 2-1), breadcrumbs (see Example 2-2), or local navigations (see Example 2-3).

Example 2-1. The main navigation of a site

```
<header>
  <nav aria-label="Main"> ❶
    <ul>
      <li><a href="/home">Home</a></li>
      <li><a href="/products" aria-current="page
      <li><a href="/team">Team</a></li>
      <li><a href="/contact">Contact</a></li>
    </ul>
  </nav>
</header>
```

❶ Unique accessible name for the landmark. More on that in Recipe 2.3.

❷ `aria-current="page"` highlights the active page.

Example 2-2. A breadcrumb navigation on deeply nested pages

```
<nav aria-label="Breadcrumb">
  <ol>
    <li><a href="/products/">Products</a></li>

    <li><a href="/products/kitchen/">Kitchen & a
    <li>
      <a href="/products/kitchen/worktops" aria-
```

```
      </li>
    </ol>
  </nav>
```

Example 2-3. A local navigation within a page

```
  <nav aria-label="Contents">
    <ol>
      <li><a href="#company">Company</a></li>
      <li><a href="#licensing">Licensing</a></li>
      <li><a href="#seealso">See Also</a></li>
      <li><a href="#References">References</a></li
      <li><a href="#externallinks">External links</
    </ol>
  </nav>
```

Please note that the use of `aria-label` attributes on navigations is not mandatory, especially if the purpose of the navigation is clear from context. In Recipe 2.3, I explain when providing an accessible name for navigation can be helpful.

Discussion

The navigation of a page must be visually and semantically distinguishable from the remaining content on the page. In

terms of styling, this usually means placing it in a prominent spot on the site or page and highlighting it using different styling than similar elements on the rest of the page. In terms of semantics, you use the `<nav>` element to define a navigation landmark and mark major navigation links. A screen reader announces something like “navigation” when the user interacts with it. However, not every group of links is automatically a navigation. For example, if your site’s footer contains only a few links, you don’t have to wrap them in a `<nav>` element. The `contentinfo` landmark itself is sufficient for that use case. That doesn’t mean you should never use the `<nav>` element inside a footer. When there are different groups of links within the footer, wrapping each group in a `<nav>` and labeling it can be helpful. It’s a judgment call, depending on the complexity and extent of the content.

The fact that `<nav>` elements are landmarks is especially useful for screen reader users because they can access them directly using keyboard shortcuts. Recipe 1.5 introduces you to landmarks.

You can use the `<nav>` element for your site’s main navigation and page-specific navigational elements like breadcrumb navigations (see Example 2-2), pagination, or content navigation (see Example 2-3). Figure 2-1 shows a breadcrumb

navigation on *ikea.com* and Figure 2-2 a local navigation on *wikipedia.org*.

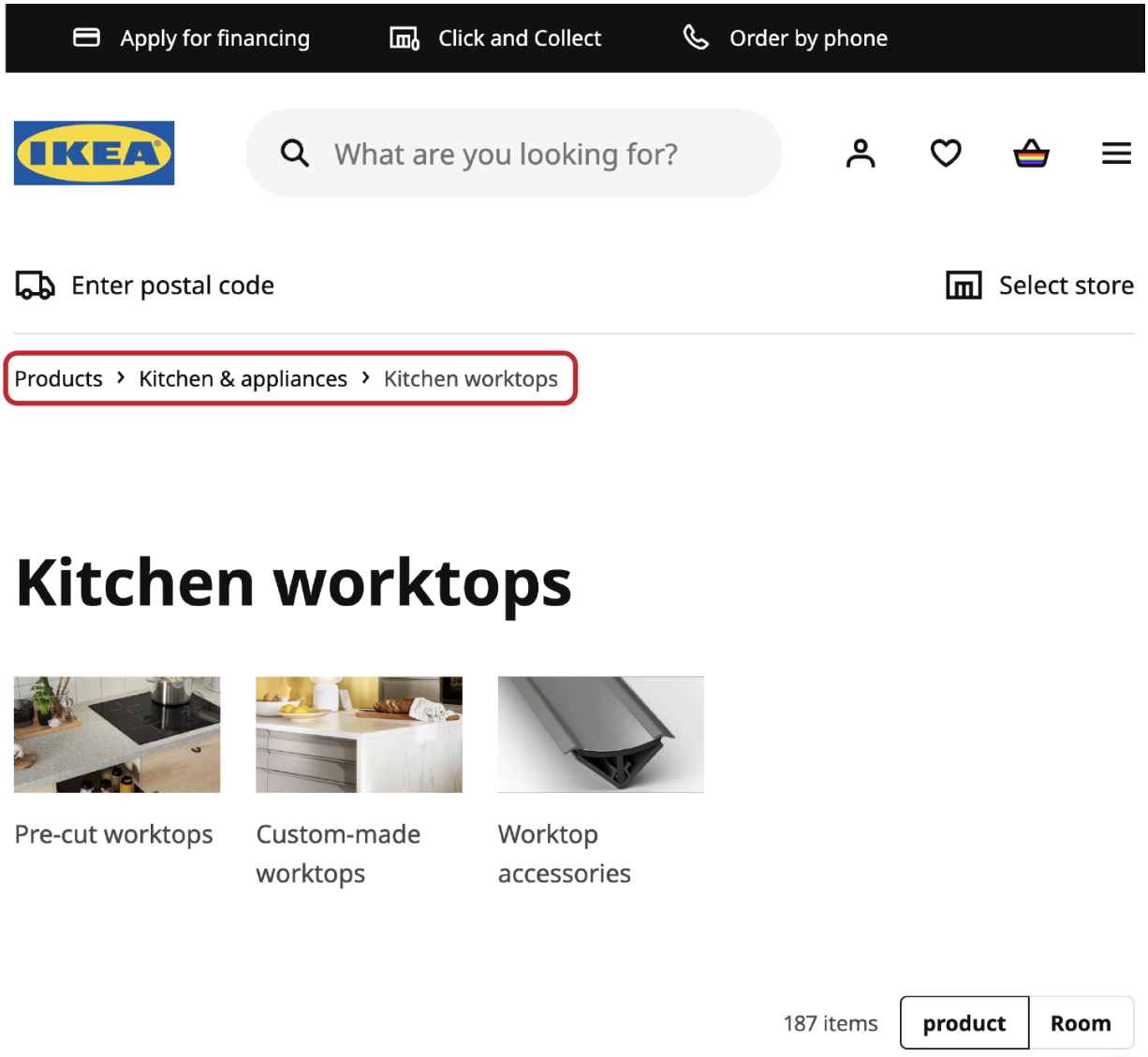


Figure 2-1. Breadcrumb navigation on ikea.com showing that the user is on the "Kitchen worktops" page within the "Kitchen & appliances" category within the Products page

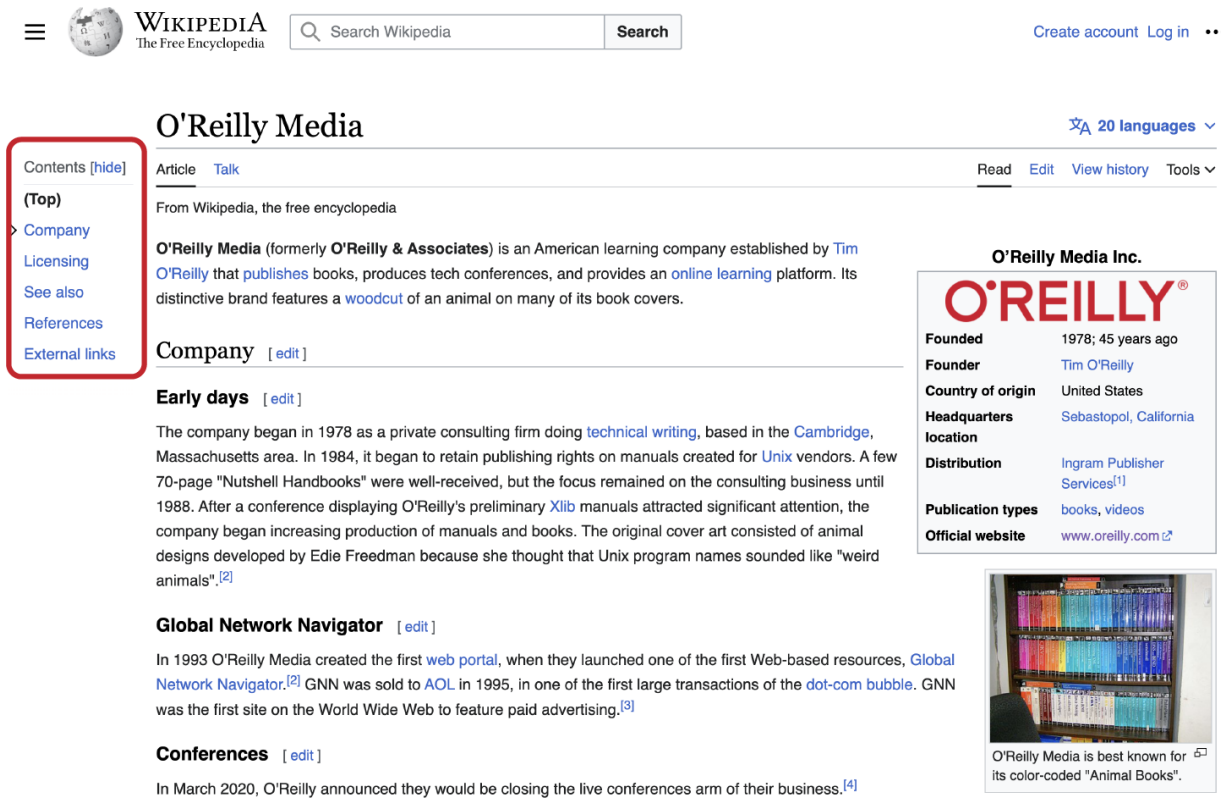


Figure 2-2. Local navigation on the Wikipedia page about O'Reilly Media shows a local navigation allowing users to access sections within the page directly

You don't have to structure the links in a navigation using a list element, but it's advisable. Recipe 7.3 discusses the benefits. You can use an ordered or unordered list. If you're unsure, defaulting to the `<ul>` (unordered list) is safe.

2.2 Create Form Landmarks

Problem

Some forms, such as search forms and filters, are central elements of a website. Especially for screen reader users, site search is often a useful alternative to conventional navigation. Users who can't easily find important forms may have difficulty obtaining information.

Solution

Promoting important forms on a page to landmarks can help with discovery. Direct access via shortcuts can be helpful for search forms (see Examples 2-4 and 2-5) and login forms (Example 2-6).

Example 2-4. Form with a search role

```
<form role="search">
  <label for="site-search">Search</label>
  <input type="text" id="site-search">
</form>
```

Example 2-5. Form within a search element

```
<search role="search">
  <form>
    <label for="site-search">Search</label>
    <input type="text" id="site-search">
```

```
    </form>
  </search>
```

Example 2-6. A login form landmark

```
<form role="form" aria-labelledby="heading_login
  <h2 id="heading_login">Login</h2>

  <div>
    <label for="username">Username</label>
    <input type="text" id="username" autocomplet
  </div>
  <div>
    <label for="password">Password</label>
    <input type="password" id="password" autocom
  </div>
</form>
```

Discussion

Site search, page search, filters, and login forms are often essential parts of a page, as shown in Figure 2-3.

You can turn a form into a `search` or `form` landmark (see Examples 2-4 and 2-6) by using the `role` attribute and labeling

the form. You can also use the `<search>` element, which has an implicit search role (Example 2-5).

Looking at Table 2-1, you'll notice that support for search and form landmarks is mixed. All tested screen readers support the `search` role well but only VoiceOver with Safari 17.3+ supports the search element. That's because the element was introduced in early 2023. Depending on your audience, you might have to combine the element with the role to get the best result. Most desktop screen readers support the `form` role when you label it. VoiceOver (on macOS and iOS), and TalkBack don't announce form landmarks. Since the role attribute doesn't harm anything when it's not supported, it's safe to use it and provide a benefit for users whose software does support it.

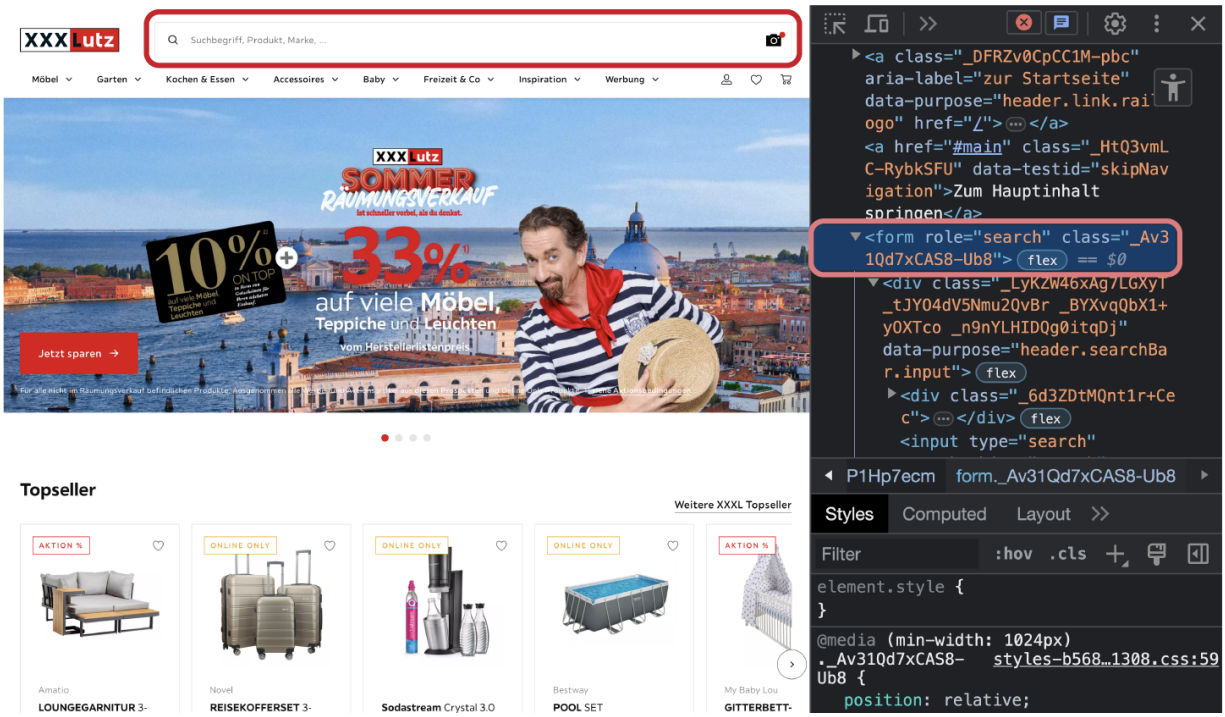


Figure 2-3. Search is the primary form of navigation on the Austrian furniture retailer lutz.at; they use a search role on the form to promote it to a landmark

I've performed these tests in early 2024, using the following shortcuts, commands, and gestures:

- `D` key + Elements list in NVDA
- Rotor in VoiceOver (VO) iOS
- Rotor + single key quick nav in VO macOS
- Swiping + landmark navigation in TalkBack
- `R` key + landmarks list (`Insert` + `Ctrl` + `R`) in JAWS
- `D` key + Landmarks List in Narrator

Table 2-1. Screen reader test: How are form and search roles exposed?

Setting	NVDA	VoiceOver	JAWS	Nar
form with no role and no label	no	no	no	no
form with no role, labeled by heading	form	no	form	form
form with form role and no label	form	no	no	form
form with form role, labeled by heading	form	no	form	form
form with no role, labeled by legend	form	no	group	form

Setting	NVDA	VoiceOver	JAWS	Nar
form with form role, labeled by legend	form	no	group	form
form with search role and no label	search	search	search	sear
form with search role, labeled by heading	search	search	search	sear
search element with no label	no	search	no	no
search element with search role and no label	search	search	search	sear

Setting	NVDA	VoiceOver	JAWS	Nar
search element with search role, labeled by heading	search	search	search	sear

[a] Bold text indicates not announced as a “form” landmark, but it’s accessible via la

2.3 Label Landmarks

Problem

Landmarks are useful only if it’s clear what they represent. When multiple landmarks of the same type exist, it can be hard for users to differentiate them, making landmark navigation difficult or even pointless.

Solution

The most reliable way to differentiate landmarks of the same type is by labeling them using the `aria-labelledby`

(Example 2-7) or `aria-label` (Example 2-8) attributes.

Example 2-7. Labeled navigation landmark via `aria-labelledby`

```
<nav aria-labelledby="pagination_heading">
  <h2 id="pagination_heading">Pages</h2>
  …
</nav>
```

Example 2-8. Labeled navigation landmarks via `aria-label`

```
<nav aria-label="Main">
  …
</nav>
<nav aria-label="Page">
  …
</nav>
```

Discussion

When you have multiple navigations on a page—like a site-wide navigation, a local navigation for the page, and a pagination—you also have three `<nav>` elements, as shown in Example 2-9.

Example 2-9. Bad practice: Three unlabeled navigation landmarks

```
<nav>
  <ul>
    <li>
      <a href="/home">Home</a>
    </li>
    ...
  </ul>
</nav>
<nav>
  <ul>
    <li>
       <a href="/category-1">Category 1</a>
    </li>
  ...
  </ul>
</nav>
<nav>
  <ul>
    <li>
      <a href="/page-1">1</a>
    </li>
  ...
  </ul>
</nav>
```

Having multiple `<nav>` elements on a page is valid, but now there are three `navigation` landmarks that all look the same. It's hard to tell them apart, unless you know the structure of the page really well (see Figure 2-4).

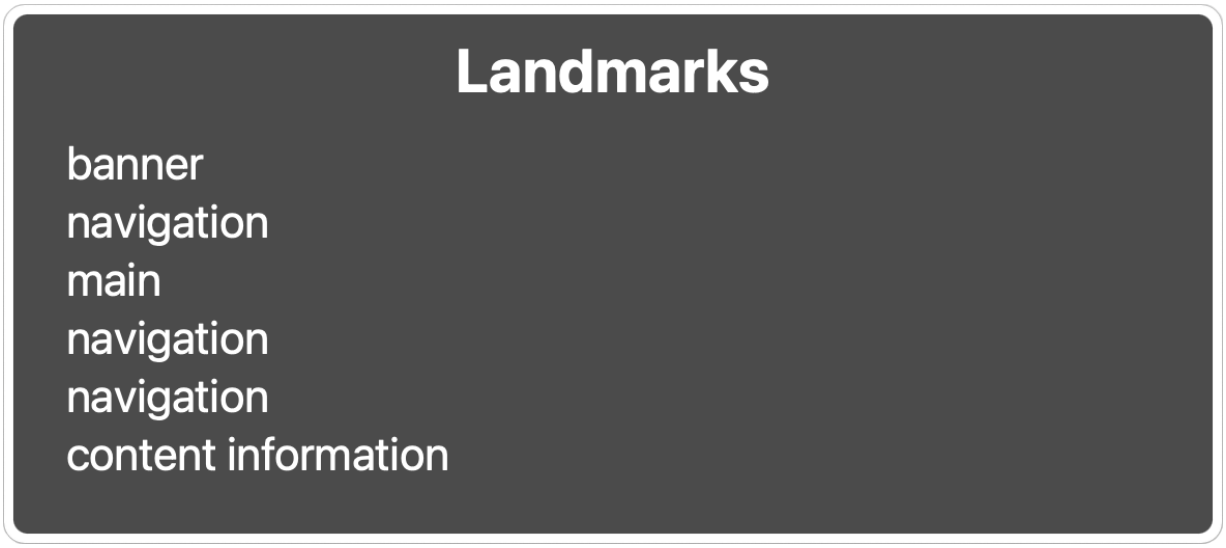


Figure 2-4. The rotor in VoiceOver listing three unlabeled navigation landmarks

To make them distinguishable, you can label them by either using `aria-labelledby` or `aria-label`. Figure 2-5 shows three labeled navigation landmarks.

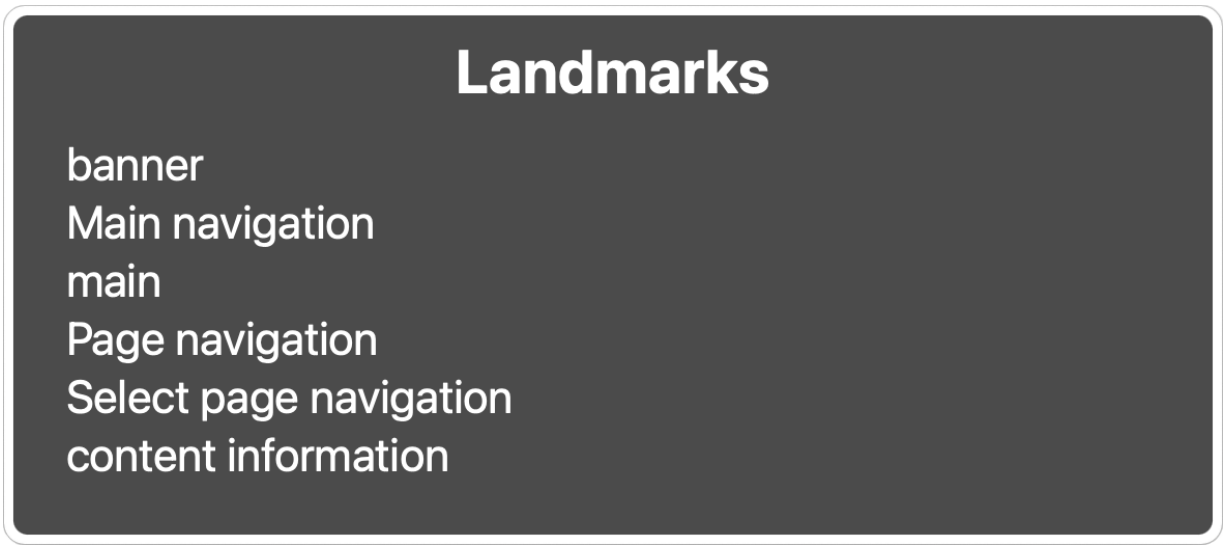


Figure 2-5. VoiceOver listing labeled and unlabeled landmarks

Picking a label

Before you pick the label for a landmark, check whether there is a useful label somewhere on the page already. If that's the case, use the `aria-labelledby` attribute to create a reference from the landmark to another element that labels it. Headings are often suitable for labeling landmarks, as Example 2-7 shows.

A good label describes the purpose of the landmark in fewer than three words. For navigations, avoid terms like "menu" or "navigation" because the element's role already conveys that. For example, a screen reader announces a label like "Main navigation" as "Main navigation navigation." Also avoid visual

names like “Burger,” “Mobile,” or “Top bar”; instead, use contextual labels like “Main” or “Content.”

Don’t label `<header>`, `<main>`, and `<footer>` landmarks because they’re supposed to be unique anyway and don’t need a label.

Generally, I recommend using `aria-labelledby` over `aria-label`, if possible, because it works with existing content instead of introducing new text, and it works better with auto-translation tools like Google Translate. If no heading or other suitable content is present, use the `aria-label` attribute, as shown in Example 2-8.

2.4 Structure the Main Content

Problem

On complex pages with a lot of content and interactive elements, it can be difficult and time-consuming for users to orient, navigate, and figure out how the page is structured. That’s especially true for users who cannot quickly scan a page for an overview, like keyboard and screen reader users and people who rely on zooming.

Solution

Use landmarks and other elements like headings and lists to provide structure for easier orientation and navigation.

The summary at the beginning of a blog post can be a region to give screen reader users direct access, as shown in Example 2-10.

Example 2-10. A quick summary landmark in a blog post

```
<h1>
  &lt;article&gt; vs. &lt;section&gt;: How To Ch
</h1>

<div role="region" aria-labelledby="summary"> ❶
  <h2 id="summary">
    Quick summary
  </h2>
  <p>
    In this article, Olushuyi explores a mental
    the <code>&lt;article&gt;</code> and <code>&
    writing documents. You will explore how grou
    and how you can make it all count for users.
  </p>
</div>
```

❶ The role and label turn this `div` into a landmark.

Focusable containers need a role and accessible name. Example 2-11 combines the `tabindex`, `role`, and `aria-label` attributes.

Example 2-11. A focusable, labeled region

```
<div role="region" aria-label="Code Demo" tabind
  <article>
    …
  </article>
</div>

<style>
[role="region"][tabindex="0"] { ❷
  overflow: auto;
}
</style>
```

❶ `tabindex` makes the region focusable.

❷ Makes focusable regions scrollable if necessary.

You can use the `section` element to group larger thematic regions of a page, as shown in Example 2-12.

Example 2-12. Thematic regions on a home page

```
<h1>The Agency</h1>

<section> ❶
  <h2>Latest News</h2>
</section>

<section>
  <h2>About Us</h2>
</section>

<section>
  <h2>Selected projects</h2>
</section>
```

❶ Unlabeled sections with no semantic meaning but useful for structuring and styling.

Labeling the `section` element turns it into a landmark (Example 2-13).

Example 2-13. A labeled section for the search region on a page

```
<section aria-label="Product Search"> ❶
  <!-- Search form/filters -->
```

```
    <!-- Search results -->
  </section>
```

- `aria-label` turns the section into landmark.

The `aside` element marks content related to the page's main content, as shown in Example 2-14.

Example 2-14. An aside element within a blog post listing related articles

```
<article>
  <h1>Cascade Layers are useless*</h1>

  <p>*if you don't understand the problems they

  <!-- rest of the blog post -->

  <aside aria-labelledby="relatedheader">
    <h2 id="relatedheader">Related Articles</h2>
    <ul>
      <li>
        <a href="/blog/2022/100daysof-day37/">
          cascade layers
        </a>
      </li>
      <li>
        <a href="/blog/2023/100daysof-day68">
```

```
            cascade layers and browser support
          </a>
        </li>
        <li>
          <a href="/blog/2023/100daysof-day74">
            using !important in cascade layers
          </a>
        </li>
      </ul>
    </aside>
  </article>
```

Structuring content using list elements, as shown in Example 2-15, can improve understanding, overview, and navigation.

Example 2-15. A recipe using unordered and ordered lists

```
<h1>Iced latte</h1>

<p>A refreshing iced latte recipe.</p>

<h2>Ingredients</h2>

<ul>
  <li>2 espresso shots (60ml)</li>
  <li>2 teaspoons sugar or honey</li>
  <li>ice</li>
```

```
    <li>110ml milk</li>
  </ul>

  <h2>Steps</h2>

  <ol>
    <li>
      Mix the hot espresso with the sugar.
    </li>
    <li>
      Fill a glass with ice and stir in the coffee
    </li>
    <li>
      Pour over the milk and stir.
    </li>
  </ol>
```

Discussion

HTML provides several elements to structure content. Whether and how you should use them depends on what you're trying to achieve.

Sections

The `<section>` element represents a generic page region used to group content thematically. It typically starts with a heading. Unlabeled `<section>` elements are semantically equal to the `<div>` element, but that doesn't mean you can always use them interchangeably. You use the `div` mostly for styling purposes or as a convenience for scripting. In contrast, the `<section>` element marks thematic groups of your page, like different sections on a home page, as illustrated in Example 2-12. Granted, you could use `divs` or no wrapper to achieve the same, but the `<section>` element has two major advantages:

- You can use the section element selector in CSS to define general styling rules for these elements.
- You can turn section elements into landmarks by labeling them, as shown in Example 2-13.

The sections in Example 2-12 have no accessible name provided via `aria-labelledby` or `aria-label` because they're not important enough to be promoted to landmarks. As soon as the `<section>` has an accessible name, its role changes from `generic` to `region`. The region role represents a generic landmark role that you can use to promote a generic element to a landmark. Use this role sparingly; limit it to regions of your page that your users feel are important enough to be able to navigate directly to them and list them in a summary of the

page. Use `aria-labelledby` or `aria-label` to describe the purpose of the region.

You should use a generic landmark only when no other landmark, like `navigation` or `complementary`, qualifies. Example 2-10 uses `region` to give users quick access to the summary of a blog post before reading the entire post.

You can also use the `region` role to identify scrollable areas. Keyboard users cannot interact with scrollable content in any browser except Firefox. You can fix that by adding `tabindex="0"` to the parent element that contains the content. That makes it interactive, meaning it also needs an accessible role and name. The `region` role can be an option, as Example 2-11 shows. As a side note: generally, it's bad to nest interactive elements (for example, a `<button>` inside an `<a>`), but it's okay to put interactive elements in scrollable and focusable regions.

A region (`<section>` or `<div role="section">`) typically starts with a heading. As a general rule, the element is appropriate only if its contents would be listed explicitly in the document's outline (more about that in Recipe 2.5). When you need an element only for styling purposes or as a convenience for scripting, use a simple `<div>` instead.

It's valid to nest `<section>` elements, especially if you have a page with chapters and many subchapters. If you were to keep using h1s as the headings for all sections, as shown in Example 2-16, something particular happens that usually doesn't come up in practice but is still good to know: the font size of each heading decreases as you nest them deeper.

In Figure 2-6, you can see how the h1 in the first section looks like an h2, the h1 in the second section like an h3, etc. That's because the WHATWG used to hold the idea that sectioning content should influence the document's outline. The WHATWG included this concept for many years, but spec authors finally removed it in July 2022 because it didn't work in practice and no browsers ever implemented it. What browsers did implement, though, were user agent styles that influenced the rankings of headings visually. In hindsight, that was a mistake. It makes it look like nesting sectioning content controls the heading level, but in reality, it affects only styling, not semantics. There is discussion of removing the user agent styles, but stakeholders are still determining whether this change would break the styling of too many websites.

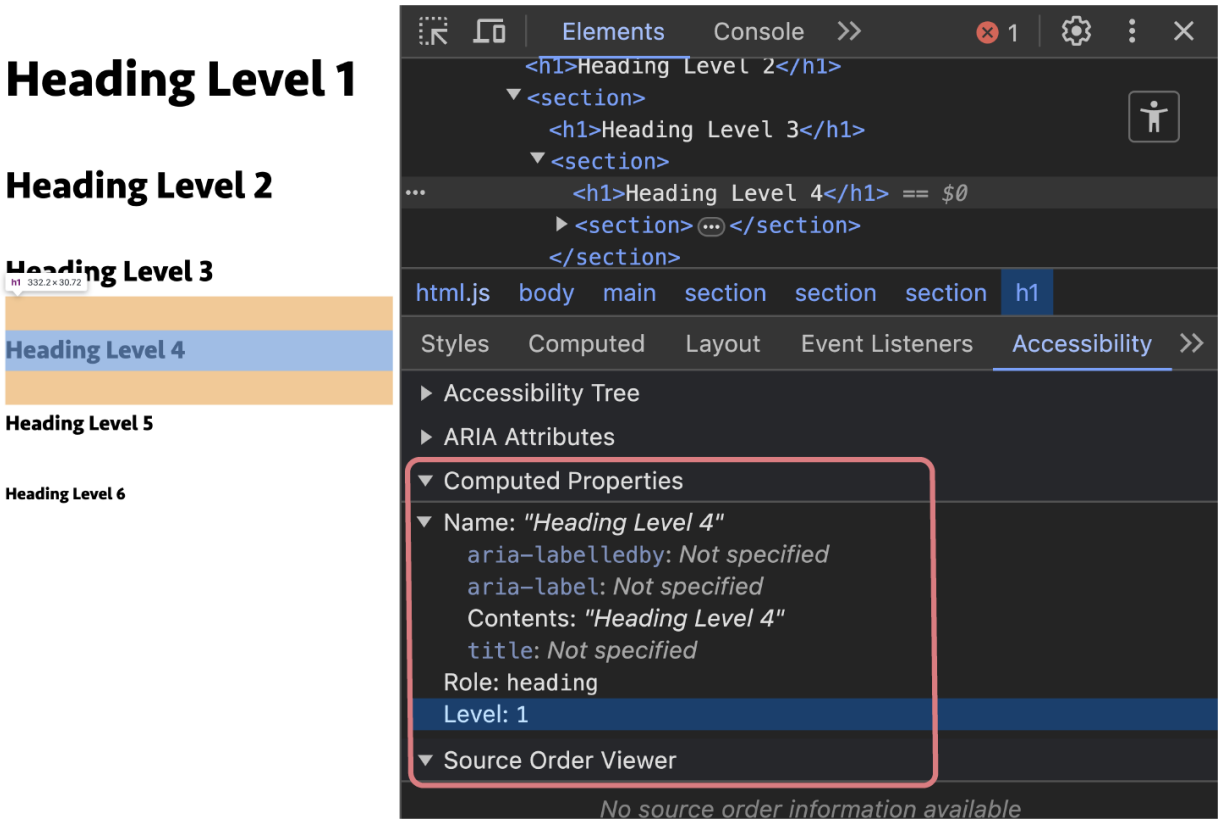


Figure 2-6. The accessibility panel in Chrome showing that the ARIA level of the heading that looks like an h4 is still 1

Example 2-16. Bad practice: Multiple sections nested, each starting with an `<h1>`

```
<h1>Heading Level 1</h1>

<section>
  <h1>Heading Level 2</h1>

  <section>
    <h1>Heading Level 3</h1>

    <section>
```

```
        <h1>Heading Level 4</h1>

        <section>
          <h1>Heading Level 5</h1>

          <section>
            <h1>Heading Level 6</h1>
          </section>
        </section>
      </section>
    </section>
  </section>
```

Even though user agent styles tell a different story, there is no outline algorithm. Avoid nesting headings with the same level. There should be only one h1, followed by h2s, and so on.

Asides

You can use the aside element to mark content tangentially related to the nearby main content that could be considered separate, such as pull quotes, sidebars, advertising, or related links, as shown in Example 2-14.

If you're testing with a screen reader, it's good to know that the implicit role of the aside element is `complementary.`

Articles

You can use the article element for any group of content that you could in theory independently distribute or reuse elsewhere, and it would still make sense. A classic example is a news article or blog post that you can read on a website or in an RSS feed reader.

However, the definition of the article element is not a literal news article but *a particular item or separate thing*. An article can be a forum post, a comment on a blog post (article nested in article), an interactive widget, a product listed on an ecommerce site, or any other complete or self-contained composition in a document. The article element can be useful for a few reasons:

- You can use the article element selector in CSS to define general styling rules for these elements.
- Third-party software, for example, RSS feed readers or reader mode in browsers, can extract content wrapped in the article tag and display it differently.
- Screen reader users can use shortcuts to access articles, but screen reader support for the article element is diverse because each software applies different heuristics, as you can see in Table 2-2.

Table 2-2. Screen reader test: Is the article element exposed?

Type	NVDA	Jaws	VoiceOver (macOS)	N
Virtual cursor/swipe	no	yes	yes	n
Landmark list	no	no	no	n
Custom article list	no	yes	yes	n
Default quick nav key	no	yes	no	n

NVDA doesn't announce the article's role when you use the arrow keys or list it in the elements list, but you can enable it in the document formatting settings and add a custom quick nav shortcut for article navigation.
JAWS announces labeled and unlabeled articles when you use the arrow keys or the `O` key to navigate. They're not included in the list of landmarks, but you can list all articles by pressing `Ctrl` + `Insert` + `O`, as displayed in Figure 2-7.

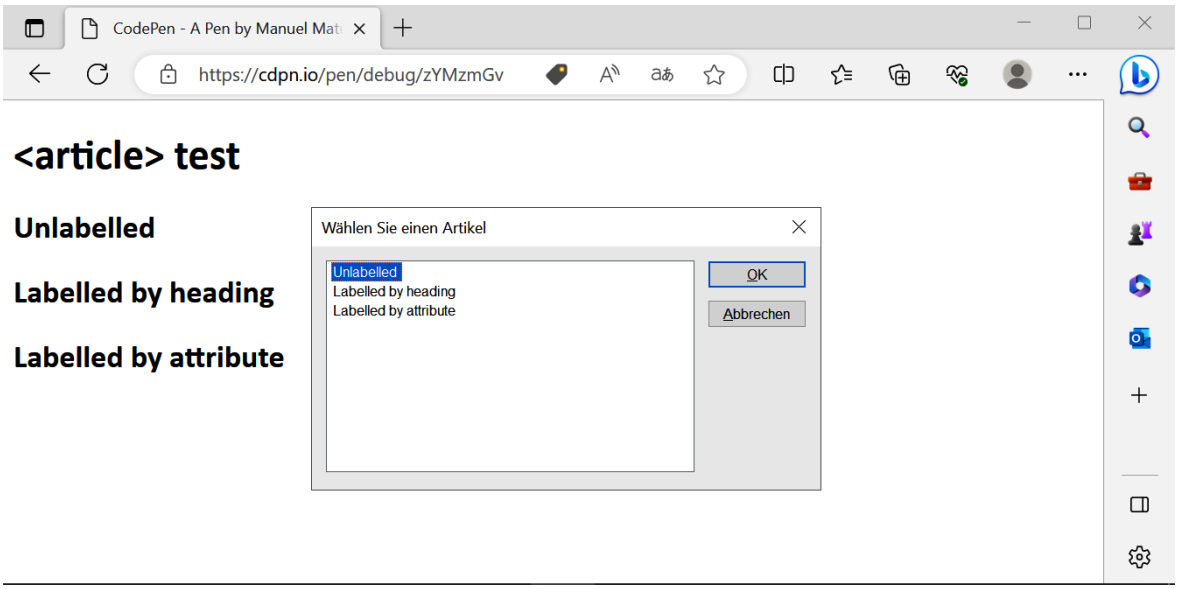


Figure 2-7. JAWS listing an unlabeled article and two labeled articles in the articles list

VoiceOver on macOS announces labeled and unlabeled articles when you use the virtual cursor. It also adds a new list of articles to the rotor. The landmarks list in the rotor doesn't include articles.

Narrator doesn't announce the article's role or list it in the elements list.

VoiceOver on iOS announces articles when you select contained items or swipe. It includes articles in the landmarks list and adds a new list of articles to the rotor.

TalkBack doesn't announce articles when you select contained items or swipe, but labeled and unlabeled articles are accessible via the landmark navigation.

Lists

You can use ordered or unordered lists to group elements visually and semantically. Besides the visual unity they form, lists provide users with many additional features:

- A screen reader can announce the total number of items when the user finds the list.
- A screen reader can announce the index of the current item (for example, “list item, two of four”).
- When you access an item in the list for the first time, a screen reader can announce that the item belongs to a list of *n* total items.
- Users can use shortcuts to jump from list to list or list item to list item.
- Users can use shortcuts to list all lists on a page and access them directly.

You can also use definition lists (`dl`), but please note that what and how much screen readers announce varies across different screen readers.

Ordered versus unordered lists

Use the `<ol>` tag when the order in your list of items matters and `<ul>` when it doesn’t. In Example 2-15, you can see the recipe for an iced latte. In this case, the ingredients are listed in

an unordered list because it doesn't matter whether you list ice or milk first. The recipe is an ordered list because the correct order of steps is relevant.

Testing lists with screen readers

You can use shortcuts in screen readers to access lists and list items:

NVDA

Use the `L` key to jump from list to list and the `I` key to jump from list item to list item.

JAWS

Use the `L` key to jump from list to list and the `I` key to jump from list item to list item. To show a list of all lists, press `Insert` + `F3`.

VoiceOver on macOS

Use `VO` + `Cmd` + `X` to jump from list to list.

2.5 Create a Sound Document Outline

Problem

When a page reaches a certain size, it can be difficult to get an overview of its contents. Users must be able to quickly tell where they are, what information is there, and how it's structured. Headings in HTML can help with that, but if you don't use them correctly, they can disorient all user groups and make navigation harder for screen reader users.

Solution

Use headings to create an outline for the document, as shown in Example 2-17.

Example 2-17. Document outline of a home page of an ecommerce site

```
<h1>Johanna's Toy Store</h1>
<h2>Toys</h2>
   <h3>Toys for babies</h3>
   <h3>Toys for kids</h3>
   <h3>Outdoor toys</h3>
<h2>Books</h2>
<h2>Special occasions</h2>
<h2>Contact</h2>
<h2>Payment</h2>
```

Discussion

Headings play an important role in web pages' overall quality, usability, and especially accessibility. Kelley Gordon, of the Nielsen Norman Group, lists headings as one of the key ingredients for creating a clear visual hierarchy. Besides other factors, they can guide the eye to the most important elements on the page and help users scan it. In Figure 2-8, you can see how combining headings and white space on the Set Studio websites creates a clear structure and enables users to get an overview quickly.

According to the latest screen reader survey conducted by WebAim in 2024, navigating headings is the predominant method for finding page information, and most respondents find proper heading structures very or somewhat useful.

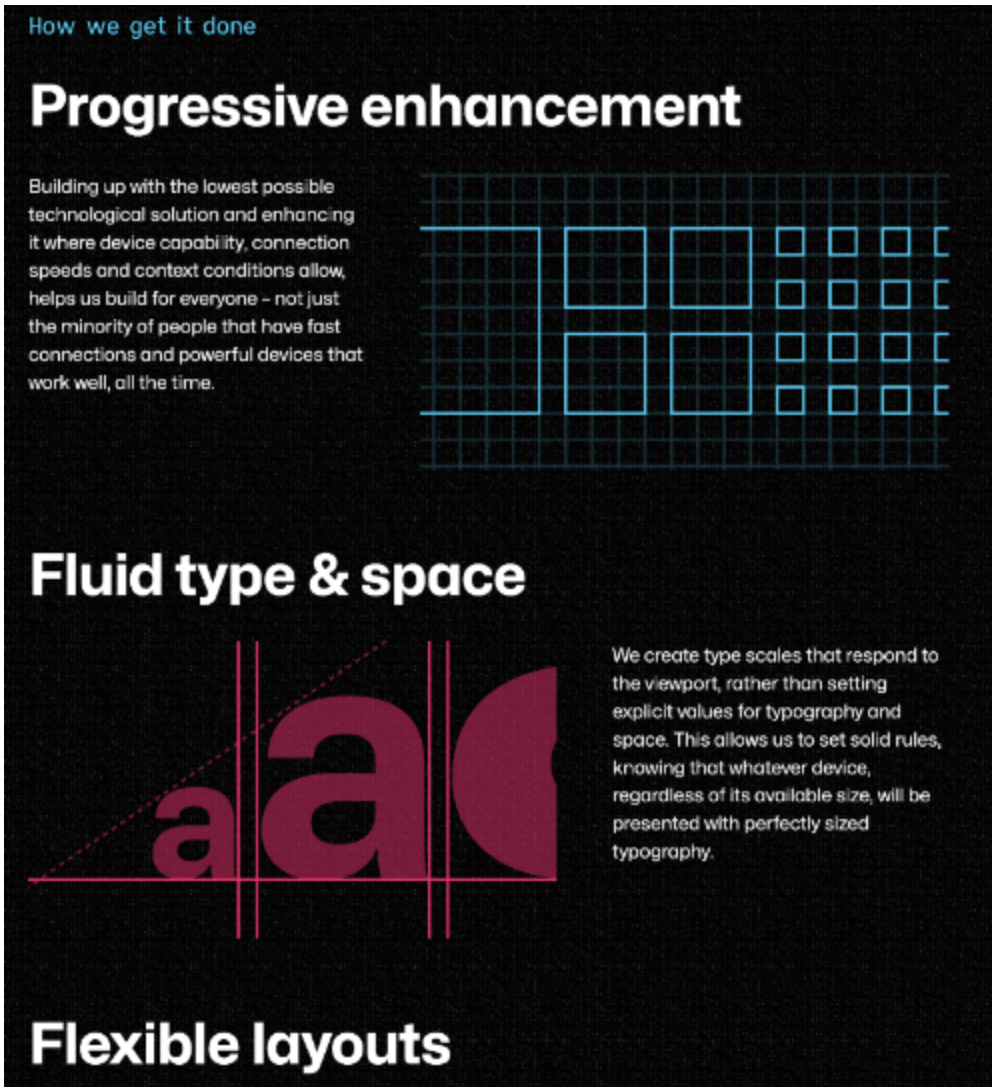


Figure 2-8. Sections on the home page of set.studio

Headings communicate how content is organized on the page. Besides the visual aspects that make orientating easier, a good document outline has many benefits for screen reader users. They can list all the headings on the page using shortcuts, as shown in Figure 2-9, select them and navigate to them directly. Table 2-3 lists those shortcuts.

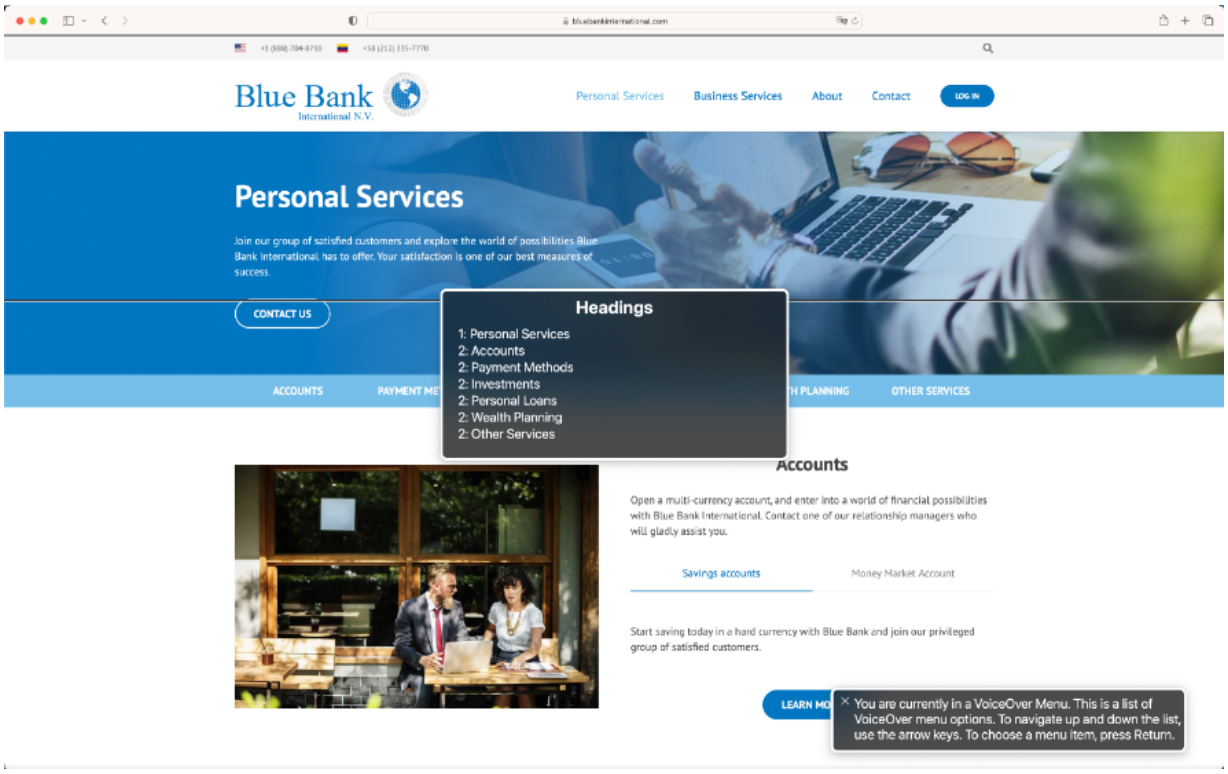


Figure 2-9. Document outline on the “Personal Services” page of the Blue International Bank, listed in VoiceOver on macOS

Table 2-3. Screen reader shortcuts for listing headings on a page

Screen reader	Command
NVDA	`Ins` + `F7`
JAWS	`Ins` + `Ctrl` + `R`
VoiceOver on macOS	Rotor
Narrator	Narrator + F6

All screen readers enable users to navigate by heading, using shortcuts without listing them first. For example, in NVDA, you can press the `H` key to jump from heading to heading or the `2` key to jump from h2 to h2 (see Table 2-4 for more commands).

Table 2-4. Screen reader shortcuts for navigating by heading

Screen reader	Command
NVDA	`H` / `1` - `6`
JAWS	`H` / `1` - `6`
VoiceOver on macOS	`Cmd` + `VO` + `H` or `H` / `1` - `6` with Quick Nav
Narrator	`H` / `1` - `6`
VoiceOver on iOS	Rotor
TalkBack	Quick Nav Menu

When a screen reader user finds a heading, the software usually announces the content of the heading along with its level (one to six). That's why it's important to keep a sound document outline.

Nest headings by their rank, starting with h1, the most important heading on the page. The `<h1>` tells users and machines what the page is about. The `<h2>`s split the page into large sections; `<h3>` through `<h6>` structure these larger sections into subsections. Once you reach the end of a subsection, you can start a new, larger section, as illustrated in Example 2-17.

Skipping levels can be confusing and should be avoided where possible, because it obscures the structure of the page. Example 2-18 shows a disorganized document outline on the home page of a clothing brand.

Example 2-18. Bad practice: Disorganized document outline with skipped headings

```
<h2>New arrivals</h2>
<h2>Shoe 1</h2> <!-- (redacted) -->
<h2>Shoe 2</h2>
<h2>Shoe 3</h2>
<h2>Shoe 4</h2>
<h4>E-Mail</h4>
<h4>Address</h4>
<h4>Hours</h4>
<h2>Store Locator</h2>
<h2>Follow us</h2>
```

An exception to this rule is headings in landmarks that come before the main content of your page, where the h1 is usually located. In that case, it's okay if the outline starts with an h2, as shown in Example 2-19.

Example 2-19. Document outline with a slightly disordered heading structure

```
<h2>Navigation</h2>
<h1>Johanna's Toy Store</h1>
<h2>Toys</h2>
<h3>Toys for babies</h3>
<h3>Toys for kids</h3>
<h3>Outdoor toys</h3>
<h2>Books</h2>
<h2>Special occasions</h2>
<h2>Contact</h2>
<h2>Payment</h2>
```

Besides the correct structure, headings must also be short and descriptive. They label sections or subsections and should clearly describe what they're about. Hoa Loranger lists five tips for headlines that convert, which also benefit accessibility:

- Make sure the headline works out of context.
- Tell readers something useful.

- Don’t succumb to cute or faddish vocabulary.
- Omit nonessential words.
- Front-load headings with strong keywords.

Although HTML headings aren’t a huge factor in SEOs anymore, SEO can also benefit from a well-structured and well-labeled document outline. The algorithms for determining the most important content on a page by search engines are obscure. However, Google recommends that you put accessibility first when you design outlines for your pages.

When you work on a document outline, imagine that you’re writing a scientific paper. A paper always needs one title (`<h1>`). There are usually several main chapters (`<h2>`), and sometimes there are also subchapters (`<h3>` through `<h6>`). By looking at the table of contents of your paper, you should be able to determine its purpose, structure, and main topics. On a webpage it should be the same. I like this analogy, but there’s one crucial difference between a paper and a website. The paper has one main title, but a website’s main title (the `<h1>`) changes on every page. It must be unique and always describe the current page. Don’t use the website’s title or the home page link in the header as the `<h1>` on every page.

When you create a document outline, it's much better to imagine how the page should be structured than to look at a design and follow only visual cues. Sometimes it makes sense to add headings to the page that aren't visible in the design, if it helps improve the document outline. You can hide these headings by using a custom visually hidden class (see Recipe 8.1). However, while this is possible and sometimes necessary, I generally don't recommend hiding content from only some users. The best solution is to show headings to everyone.

You can use tools to visualize the heading structure. Matthias Ott lists several of them in "Level Up Your Headings Game".

2.6 Present Content in Order

Problem

Users may not be able to make sense of content if you don't present it in a meaningful sequence or if the visual order of elements doesn't match the order of elements in the document. Screen reader users may need time to understand how the page is structured. A disorganized page makes the experience for keyboard and screen reader users unpredictable and confusing.

Solution

Structure page content from top to bottom to make sense, even when presented without CSS. Example 2-20 shows how you can structure a page.

Example 2-20. A typical web page structure, ordered from top to bottom

```
<header>
  <a class="skip-link" href="#content">Jump to ma

  <a href="/">
    Johanna's Toy Store
    <svg>…</svg>
  </a>

  <nav aria-label="Main"> ❷
    <ul>
      <li>
        <a href="/home">Home</a>
      </li>
      …
    </ul>
  </nav>
</header>
```

```
<main id="content">
  <h1>Johanna’s Toy Store</h1>

  <h2>Toys</h2>
  <h3>Toys for babies</h3>
  <h3>Toys for kids</h3>
  <h3>Outdoor toys</h3>
  <h2>Books</h2>
  <h2>Special occasions</h2>
  <h2>Contact</h2>
  <h2>Payment</h2>
</main>

<footer>
  &copy; 2024
</footer>
```

1. A skip link (more about skip links in Recipe 6.6).
2. Main navigation of the site.

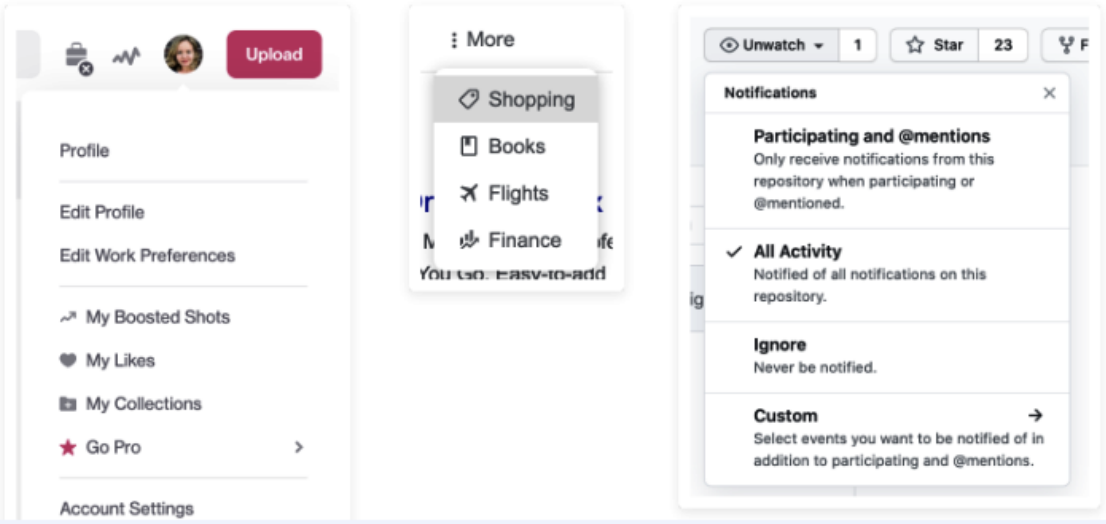
Discussion

When you arrange the content on a page, there are two things you need to consider: its general structure and order and the visual presentation of elements.

You have learned how to structure a site using high-level landmarks in Recipe 1.5 and how to structure a page using low-level landmarks (Recipe 2.4) and headings (Recipe 2.5). The order of elements in the examples in these recipes is not arbitrary. Work through the document from top to bottom and start with the most relevant content, as shown in Example 2-20. Usually, that's the header, which contains essential navigational elements like a skip link (Recipe 6.6), a link to the home page, the main navigation (Chapter 7), and a search (Recipe 1.5).

Suppose your site contains interface elements that users must interact with before accessing the content, like a cookie consent banner or a region and language selection. In that case, these elements should come first so users can deal with it first, no matter how they access the site. Following the header comes the main content, which puts the most important and relevant information first, structured by headings. The `<footer>` follows the `<main>` and contains tangentially related content.

A typical mistake is following the visual representation of elements in the design without paying enough attention to the logical order within the page. If you take the teaser card in Figure 2-10 as an example and strictly follow the visual presentation, you'd get the markup in Example 2-21.



When CSS Isn't Enough: JavaScript Requirements For Accessible Components

by Stephanie Eckles

Spoiler alert: tooltips, modals, tabs, carousels, and dropdown menus are some of the user interface components that require more than CSS. To ensure accessibility of your interface, JavaScript is a necessary addition to accomplish focus management, respond to keyboard events, and toggle ARIA attributes.

Read "When CSS Isn't Enough".

Figure 2-10. A card with a heading, text, image, and a call-to-action-link

Example 2-21. A teaser for a blog post with the image preceding the heading

```
<img src="teaser.png"
     alt="Example dropdown menus from Dribbble, (

<h3>
  When CSS Isn’t Enough: JavaScript Requirements
</h3>

<p>by Stephanie Eckles</p>

<p>
  Spoiler alert: tooltips, modals, tabs, carousel
  menus are some of the user interface component
  than CSS. To ensure accessibility of your inte
  a necessary addition to accomplish focus manag
  keyboard events, and toggle ARIA attributes.
</p>

<p>Read <a href="#">“When CSS Isn’t Enough”</a>.
```

That looks all right, but considering the purpose of the heading (it introduces a new subchapter), the image is misplaced

because it comes before the heading. Starting with the heading first makes much more sense, as shown in Example 2-22.

Example 2-22. Improved order: A teaser for a blog post with the image following the heading

```
<h3>
  When CSS Isn’t Enough: JavaScript Requirements
</h3>

<img src="teaser.png"
     alt="Example dropdown menus from Dribbble, (

<p>by Stephanie Eckles</p>

<p>
  Spoiler alert: tooltips, modals, tabs, carousel
the user interface components that require more
your interface, JavaScript is a necessary additi
respond to keyboard events, and toggle ARIA attr
</p>

<p>Read <a href="#">“When CSS Isn’t Enough”</a>.
```

You can then use CSS to reorder elements visually. Arguably, it would be better to design a page that matches the order in the

document or component in the first place. That's not always doable, but an ideal you should follow.

Refrain from reordering content, especially interactive elements, using CSS properties like `flex-direction`, `order`, `grid-auto-flow` or techniques like explicit placement, absolute positioning, or negative margins. When you deal with interactive elements, visual order should always represent Document Object Model (DOM) order as well as possible (Recipe 6.5 deals with that topic in depth) because tab order follows DOM order no matter how you arrange items visually.

The easiest way to get page structure and content order right is to work HTML-first and design a document that works well even when presented without CSS. If you can look at a final HTML document without any styling and understand how the page is structured and how contents relate, you have done a great job. Now you're ready to start writing CSS.

See Also

- "Accessibility of the section element" by Scott O'Hara
- "Under-Engineered Responsive Tables" by Adrian Roselli
- "Keyboard-Only Scrolling Areas" by Adrian Roselli

- “Why You Should Choose HTML5 article Over section” by Bruce Lawson
- “article vs. section: How To Choose The Right One” by Olushuyi Olutimilehin
- “On the dl” by Ben Myers

Chapter 3. Linking Content

Hyperlinks serve different purposes; they link to other web pages, sections within a page, external resources, email addresses, telephone numbers, and files. Users have certain expectations of what happens when they find and click a link, and you should meet these expectations with your designs and technical implementations.

A well-designed link has a concise and descriptive text label. It informs users up front what will happen when they click it, and users can activate it with any input modality. When it doesn't, users might be unable to identify the link as such, use it, or understand its purpose.

This chapter closely examines the anatomy of hyperlinks and explains how to label, style, and create links in server-side and client-side rendered environments.

3.1 Pick the Right Element

Problem

If a link to a resource doesn't meet basic requirements, you may exclude one or more groups of users from being able to access it or understand its purpose. The discussion section of this recipe provides more detail, but the essential criteria for a link are:

- It must convey its link role.
- It has an accessible name.
- The label is unique, concise, and straightforward.
- It's accessible to assistive technology.
- It's focusable with the keyboard.

TIP

Some elements are focusable; for example, links, buttons, or form fields. You can access them using the keyboard by pressing `Tab` or `Shift` + `Tab`. By default, browsers highlight focused elements visually.

Solution

HTML offers a simple and powerful way of linking to other resources: the `<a>` element in combination with the `href` attribute. Example 3-1 shows links to different kinds of targets.

Example 3-1. Different kinds of links

```
<!-- Link to an external site -->
<a href="https://www.oreilly.com/products/books-
  O'Reilly books and videos
</a>

<!-- Link to an internal page -->
<a href="/blog">
  Blog
</a>

<!-- Link to an email address -->
<a href="mailto:support@johannastoys.com">
  support@johannastoys.com
</a>

<!-- Link to a telephone number -->
<a href="tel:+17078277019">
  (707) 827-7019
</a>

<!-- Link to an anchor within the page -->
<a href="#content">
  Skip to content
</a>
```

Discussion

A hyperlink must meet specific requirements to be accessible and provide great UX.

It links to an internal or external resource

Sometimes there is confusion about whether it's better to use a link or a button for a task. All types of links have in common that they take you somewhere else when you click them—to another page, another site, a part of your page, or another application. The button element submits a form or runs JavaScript code.

TIP

If it takes you somewhere else, use a link. If you submit a form or run JavaScript, use a button.

There are exceptions to this rule, but it's true for most links and buttons. When you find yourself using one of the inaccessible examples in Example 3-3, you likely want to use a button instead.

It conveys its semantic link role

When you link to another resource, your link should convey its semantic role. The safest and most reliable option is to use the

`<a>` element with the `href` attribute because it has an implicit `link` role.

A screen reader announces the role alongside the text; for example, “O’Reilly books and videos, link.”

It has an accessible name

A meaningful link text is essential for a good user experience. When you write the text for a link, keep it short, understandable, and meaningful, and avoid repetition. Kate Moran argues that a good link is *sincere*, *substantial*, *succinct*, *specific*.

Sincere means that links should meet users’ expectations because when they don’t, they slowly erode the user’s trust in the site and the organization it represents. *Substantial* says that links must stand out and be *succinct* to increase the likelihood that users will quickly understand them as they scan and process the page. *Specific* means that it should describe its purpose. You should avoid generic link text like “learn more,” “read more,” “click here,” “download,” or “here” because vague or repetitive text makes it difficult for users to anticipate what these links lead to.

Instead of writing something like:

- *Click here to learn how to create attributes in HTML.*

Use a more specific phrasing:

- *Learn how to create attributes in HTML.*

In addition to Moran's point, there's another good reason to avoid generic phrasing: There are different ways of accessing links on a page. A blind screen reader user might use the `Tab` key to jump between interactive elements on a page. When they focus a link, the software announces its role and accessible name, for example "click here, link." Without any context, it's impossible to tell where this link will bring you. This also applies for other ways of accessing content: some screen reader users might use shortcuts to list all the links on a page, as illustrated in Figure 3-1. If several links are called "Shop now," "Read more," or "Read the story," it's impossible to know without more context what's behind each link.

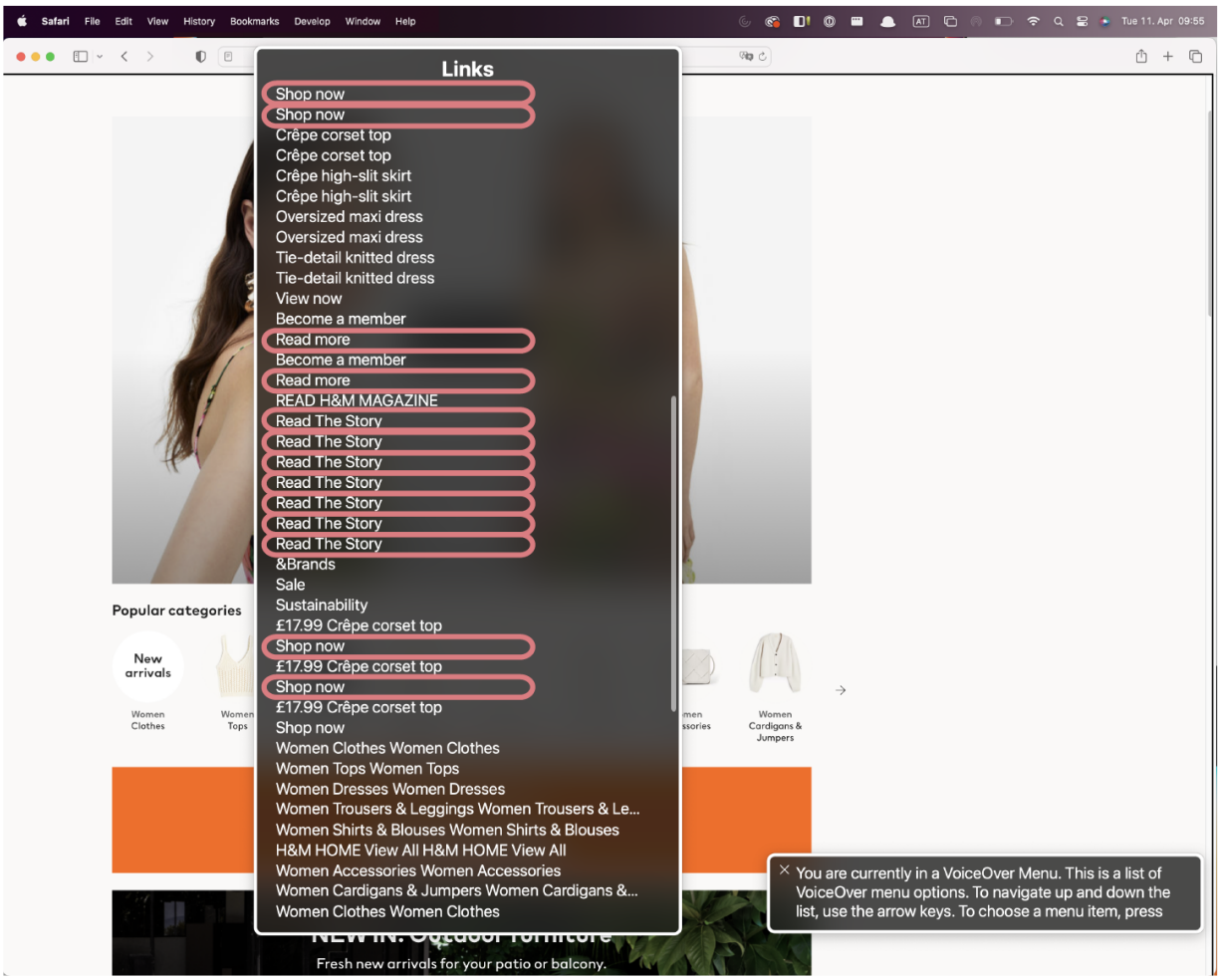


Figure 3-1. Many generic and repetitive links on hm.com

Rian Rietveld adds another good reason in her article "Creating the perfect link": Sighted people often scan pages quickly until they find something of interest. Links usually stand out in the text and should serve as a teaser for scanning users. Generic link text is neither appealing nor informative.

It communicates its current state

A link can have different states: unvisited, visited, hovered, active, and focused. For each state, you should provide suitable styles, as shown in Example 3-2.

Example 3-2. Styling links in different states

```
a:link {
  color: blue;
  text-decoration: underline;
}

a:visited {
  color: rebeccapurple;
}

a:focus-visible { ❶
  outline: 2px solid currentColor;
  outline-offset: 2px;
}

a:hover {
  text-decoration: none;
}

a:active {
  color: red;
}
```

- An alternative to `:focus`. You can read about the difference in Recipe 6.1.

On top of that, ARIA attributes can communicate additional state. You can find an example in Recipe 7.2.

It must be tabbable and allow activation via click, touch, and key events

A link is an *interactive* element, which means that if you can click it, you must also be able to perform the same action using the keyboard by pressing `Enter`. For that, it must be tabbable (reachable via the `Tab` key), which the `<a>` element is by default, provided that it has an `href` attribute.

Some people like to use modifier keys, like `Cmd`, when clicking, or right- or middle-click links to perform other actions, like opening the link in a new tab or copying the URL. Depending on the browser and operating system, these actions also come with the `<a>` out of the box.

Creating well-designed links is challenging, but links are one of the essential features of the web. Try to follow the best practices in this recipe and avoid several common alternative solutions that don't meet the requirements, listed in Example 3-3.

Example 3-3. Bad practice: Inaccessible links

```
<!-- Link with a click event.
Wrong role. This should be a <button>. -->
<a href="javascript: void(0)" onclick="[JS funct
  O'Reilly books and videos
</a>

<!-- Link with a click event.
Wrong role. This should be a <button>. -->
<a href="#" onclick="[JS function]">
  O'Reilly books and videos
</a>

<!-- Placeholder link with click event.
Wrong role. This should be a <button>. -->
<a onclick="[JS function]">
  O'Reilly books and videos
</a>

<!-- Focusable placeholder link with click event
Wrong role. This should be a <button>. -->
<a onclick="[JS function]" tabindex="0">
  O'Reilly books and videos
</a>

<!-- span with link role and click event.
Not focusable, no key events, and no middle- and
```

```
<span role="link" onclick="[JS function]">
  O'Reilly books and videos
</span>

<!-- Non-focusable link.
Not focusable. -->
<a href="https://www.oreilly.com/products/books-
  O'Reilly books and videos
</a>

<!-- Focusable but hidden from the accessibility
Inaccessible to screen reader users. -->
<a href="https://www.oreilly.com/products/books-
  O'Reilly books and videos
</a>
```

3.2 Style Links

Problem

Inaccessible custom styling of links can influence users' ability to navigate, get oriented, and perceive the website in several ways, including:

- Keyboard users might not know where on the page they're located.
- People with low vision and those who use forced colors (more on that in Recipe 5.2) might be unable to distinguish links from regular text.
- Users might not recognize a link or understand what to expect from it if its styling doesn't meet their expectations.
- People with disabilities that affect their dexterity and motor movements can have difficulty clicking links. So might anyone who's moving while using a website, for example, walking, riding the subway, or multitasking.

Solution

You should style links in a way that serves your users. Underline links, and don't rely on color alone. Avoid removing the default underline in blocks of text and provide different styles for different states, as shown in Example 3-4.

Example 3-4. Custom styling for four different states a link can be in

```
a:link {
  color: blue;
  text-decoration: underline;
```

```
}

a:visited {
  color: rebeccapurple;
}

a:hover {
  color: green;
  text-decoration: none;
}

a:active {
  color: red;
}
```

Use prominent styling for keyboard-focused links. In Example 3-5, you can see a combination of the `outline` and `outline-offset` properties used for the `focus-visible` or `focus` states.

Example 3-5. A 2-pixel-wide outline with extra padding

```
a:focus-visible {
  outline: 2px solid currentColor;
  outline-offset: 2px;
}
```

```
/* or */

a:focus {
  outline: 2px solid currentColor;
  outline-offset: 2px;
}
```

Don't use `box-shadow` alone for link or focus styling because shadows will not be displayed in `forced-colors mode`. A workaround is to use it in combination with a transparent outline, as shown in Example 3-6.

Example 3-6. A combination of box-shadow and a transparent outline

```
a:focus-visible {
  box-shadow: 0 3px 0 0 currentcolor;
  outline: 2px solid transparent;
}
```

Discussion

Color must not be the only means of distinguishing a visual element, because many users have difficulty perceiving color. That includes older adults who may not see well, color-blind people, users of older or low-quality displays, and people with

partial sight who experience limited color vision. Figure 3-2 shows how *wikipedia.com* might look for someone who doesn't perceive color.

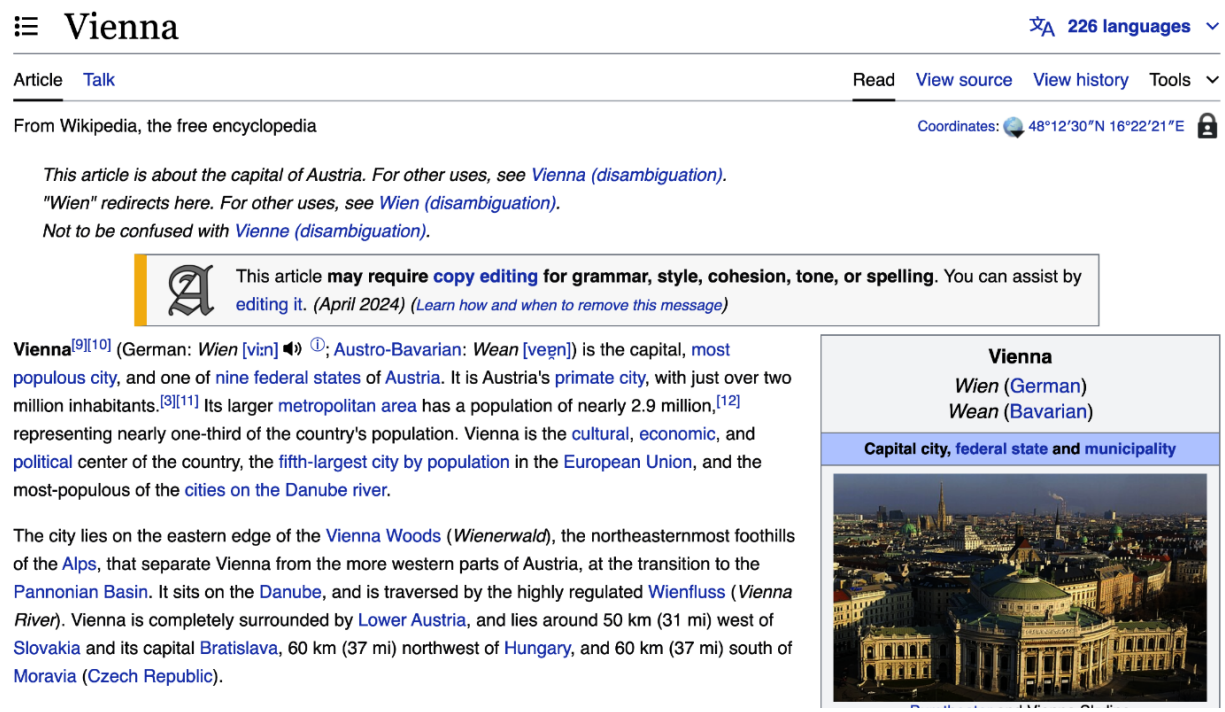


Figure 3-2. Wikipedia presented without colors; links are hard to distinguish from regular text

A different text color from the regular text color in combination with an underline works best for most users. Make links look like links (as in Figure 3-3), and don't try to reinvent the wheel.

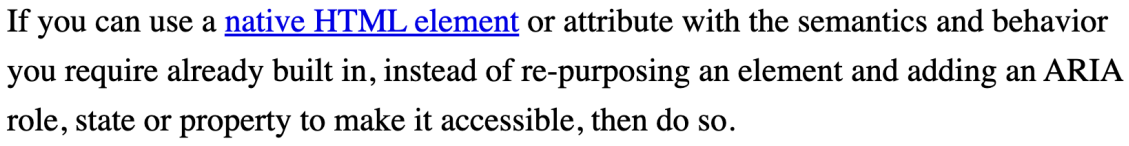


Figure 3-3. Default link styling in most browsers

That's especially true for links in blocks of text, but there are exceptions to this rule. Sometimes links look like buttons (for example, call-to-action links), and sometimes they don't need an underline, like links in a navigation. Accessibility expert Eric Eggert argues that's fine if the functionality is clear from the context.

States

Links have different states that you can style with CSS. Use the `:link` `:hover`, `:focus`, `:focus-visible`, `:visited`, and `:active` pseudoclasses to give users more context and help them get oriented. Your UI must provide users with feedback when they hover over and focus on a link, and it can be beneficial to know whether they've clicked a link before.

Focus styling

Depending on the browser, default focus styles may or may not be clearly visible and may or may not work with the colors on

your site. If that's the case, provide custom styling that works with your design. The gov.uk website is an excellent example of focus styles that are clearly visible and work well with the corporate design (see Figure 3-4).

It's possible to create focus styles that work well for users and look nice. However, aesthetics shouldn't be your primary focus—but usability should.

Target size

Target sizes, the clickable areas of interactive elements, must be large enough for users to activate them easily on small touchscreen devices or on larger screens. Small targets can be hard to activate for people who have limited dexterity or other difficulties using mice and similar pointing devices.

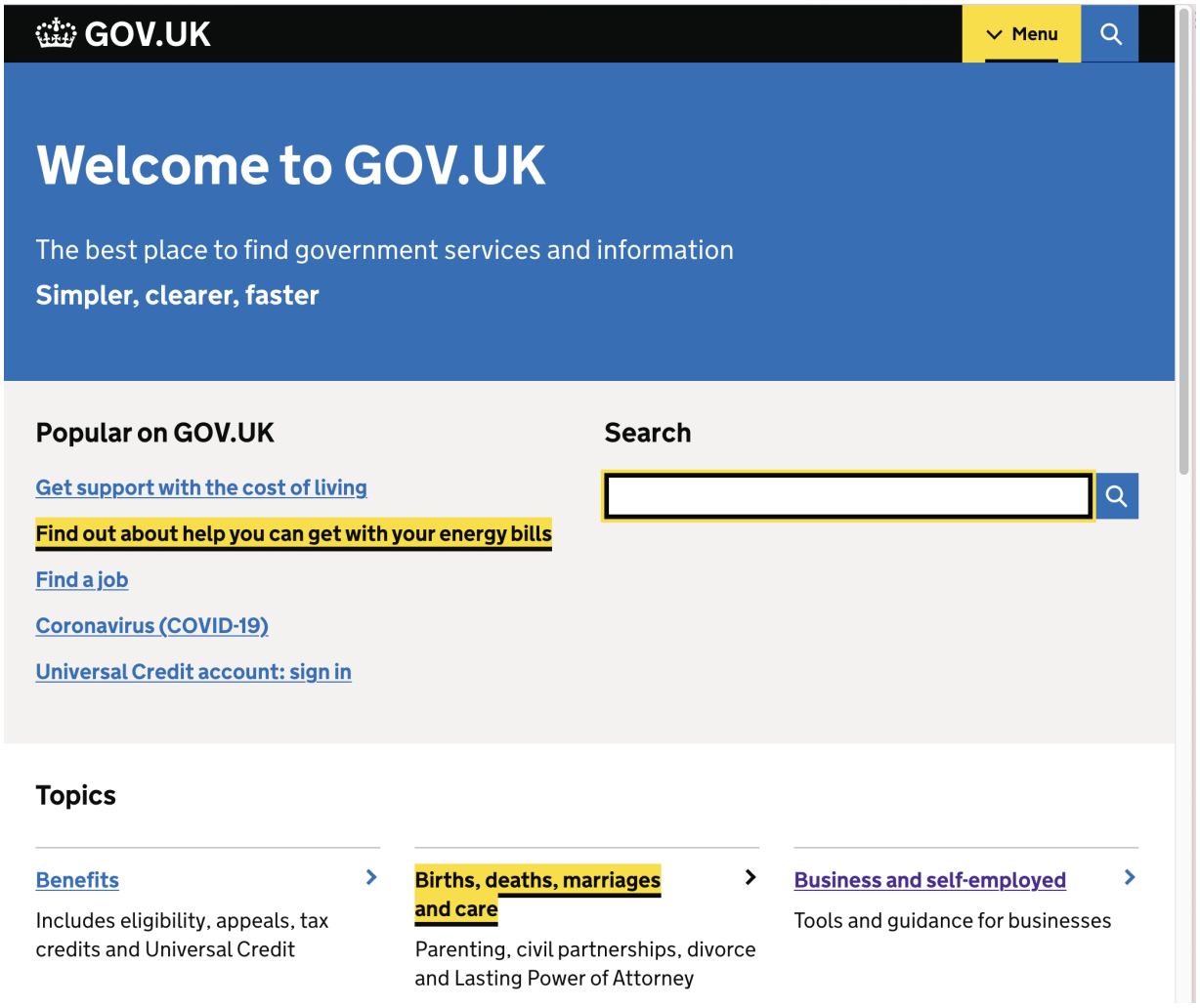


Figure 3-4. gov.uk uses different custom focus styles that work well with its overall design

Guideline 2.5.8 Target Size (Minimum) of the Web Content Accessibility Guidelines (WCAG) 2.2 suggests that touch targets should be at least 24 x 24 CSS pixels.

NOTE

Thanks to advanced display technology, screen resolutions on mobile devices are sometimes comparable to resolutions on desktop devices. In CSS, you can query the pixels available on the screen to apply different styles, but you must consider the screen resolution and size to provide users with a suitable layout on every device. Manually doing that is a tedious task, so the *device pixel ratio* (DPR) is used to define the relationship between "device pixels" and "CSS pixels" on a particular device. For example, if the device pixels are twice as large as the CSS pixels, you would say the device has a DPR of 2.0.

High contrast mode

For styling underlines, I recommend `text-decoration`, and, for focus styles, `outline`. Developers like to use `box-shadow` to create similar effects. It's easy to work with since it doesn't affect the width or height of the element, unlike `border`, and it allows authors to create nested borders. However, it doesn't work well with forced-colors mode, an accessibility feature intended to increase the readability of text through user-defined colors and a limited palette, that may be high or low contrast, as illustrated in Figure 3-5. Example 3-6 shows a workaround for that issue. In CSS, `transparent` is a color that is normally transparent but can be visible in forced-colors mode. So, if users don't use forced colors, they see the box shadow; if they do, they get a 2px outline.

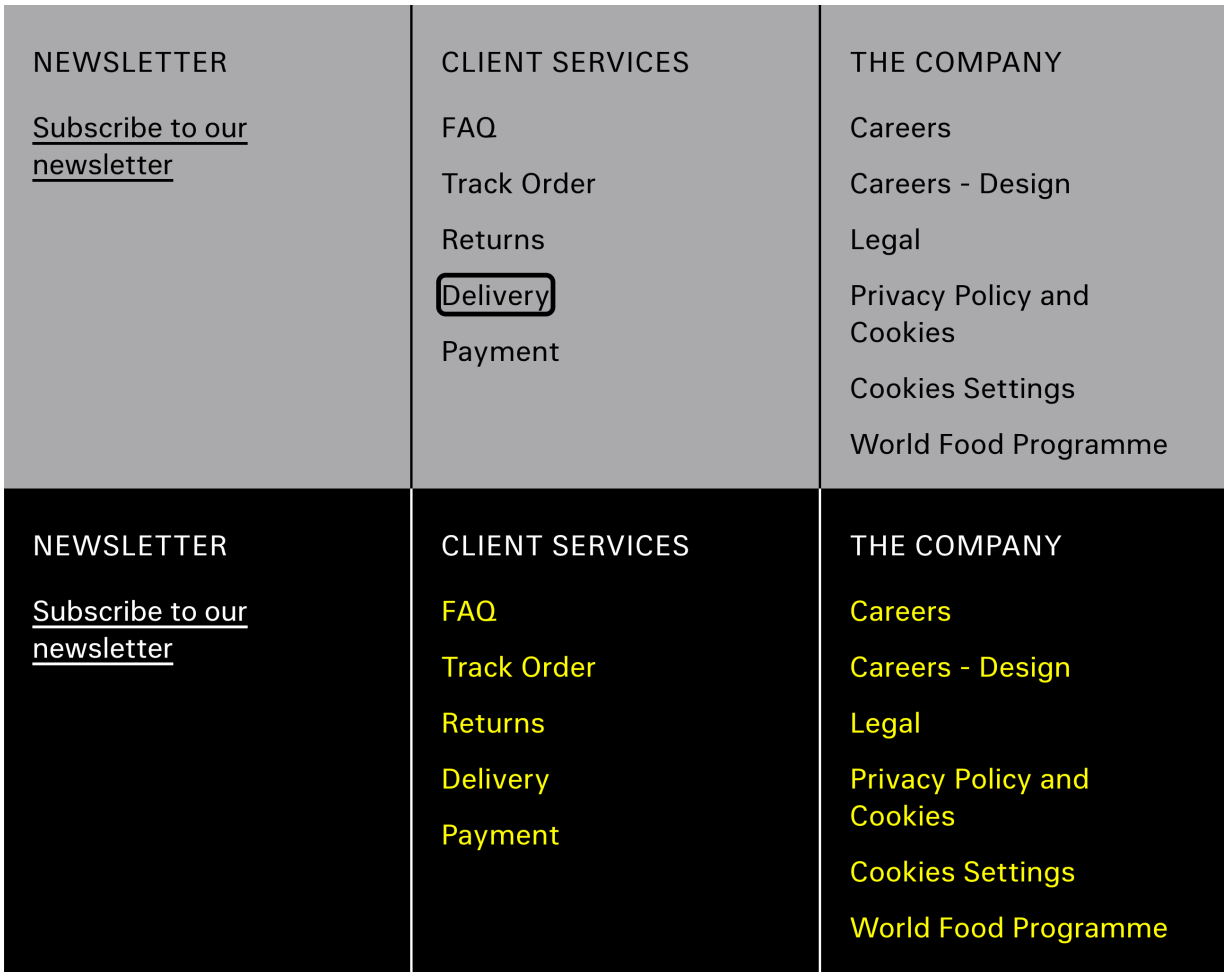


Figure 3-5. Focusing the “Delivery” link in normal and forced colors mode on balenciaga.com. Focus styles are not visible in forced colors mode.

3.3 Create Download Links

Problem

The `<a>` element’s versatility requires that it conveys, before the user clicks it, its purpose, functionality, and what action a click will perform. When you link to a file instead of a document, users should know ahead of time the format and size

of the file and how they will access it. Unexpected changes in context can be irritating, and downloading large files can have financial consequences for users.

Solution

The links in Example 3-7 answer critical questions:

- How will I access the file?
- What information does the file contain?
- Which format does it have?
- How large is it?

Example 3-7. Links to files

```
<!-- Link to a pdf -->
<a href="/menu.pdf">
  View our delicious menu (PDF, 1.2MB)
</a>

<!-- Download link to a pdf -->
<a href="/report.pdf" download>
  Download the sustainability report 2022 (PDF, 
</a>
```

For download links, you can also provide different or additional styling; for example, you could use an icon that symbolizes a download, as shown in Example 3-8 and Figure 3-6.

Example 3-8. Using the download attribute as a hook in CSS

```
[download]::after {
  content: "";
  background: url('../icons/download.svg') no-re
  block-size: 1em;
  display: inline-block;
  inline-size: 1em;
}
```

Figure 3-6. A “download” icon next to a link to a PDF

You can also use the `download` attribute to rename a file. Instead of the original filename, the name provided as the value of the download attribute will be used when it’s downloaded, as shown in Example 3-9.

Example 3-9. A link to a file with a custom filename

```
<a href="/hdj588pwW-1312d-oek92x.xls" download="
```

```
    Download sales Q1 2023 (Microsoft Excel, 0.3MB
  </a>
```

Discussion

The text of a link should describe its purpose to allow users to distinguish it from other links on a web page and determine whether to follow it. You can help users understand what will happen when they click a link to a file by including specific actions in the link text. You can use phrases like “view” or “download,” depending on the purpose, but always in combination with the target, to avoid generic link texts. Examples 3-10 and 3-11 show two different ways of linking to an image.

Example 3-10. Bad practice: A link to an image and a download link for an image with generic text

```
<a href="/images/flower.jpg">
  View
</a>

<a href="/images/flower.jpg" download>
  Download
</a>
```

You know what you can expect by reading the link text in Example 3-11.

Example 3-11. A link to an image and a download link for an image with specific text

```
<a href="/images/flower.jpg">
  View a photo of the flower
</a>

<a href="/images/flower.jpg" download>
  Download a photo of the flower
</a>
```

Users may prefer to know the type of a linked file to avoid unnecessary downloads if their operating system or software doesn't support that kind of file, as illustrated in Example 3-12.

Example 3-12. A link to a potentially unsupported file type

```
<a href="/files/logo.ai" download>
   Download raw file (Adobe Illustrator)
</a>
```

Another reason is that they may want to avoid opening specific types of files; for example, PDFs are often inaccessible, and

users may want to find other ways of obtaining the same information.

Instead of text, you can use icons to communicate the file type, as shown in Example 3-13. Figure 3-7 illustrates how the City of Amsterdam uses icons to highlight links to PDFs. Depending on your audience, this may or may not work well. The biggest drawback can be that your users don't understand specific icons. Adrian Roselli lists other points against using icons instead of text in "Showing File Types in Links". He argues that you have to maintain a library of images for all file types you intend to support and that styling and aligning icons can be cumbersome. If you want to play it safe, I recommend using text.

Example 3-13. Using an attribute selector and a pseudoelement to display icons depending on the linked file type

```
a[href$=".pdf"]::after {
  background-image: url(../icons/pdf.svg);
}

a[href$=".tiff"]::after {
  background-image: url(../icons/tiff.svg);
}
```

- Attach a photocopy of a valid identification document. For example a passport, identification card or driving licence.
- a copy of the original lease contract or contract of sale for your residence or a consent form (PDF, 160 KB) (PDF) signed by the primary resident, including a copy of their ID. The copy must include the photo and the signature.
- in the case of anti-squat: the user agreement.

Figure 3-7. The City of Amsterdam website puts an icon of a document next to links to PDFs

For some people, it's critical to know the file size, especially those with slow connections or who don't have unlimited data plans. You can include an approximate size alongside the file type in parentheses after the link text, as shown in Example 3-14.

Example 3-14. A link to a large file

```
<a href="/images/garden.tiff">
  High-resolution version of the image (TIFF, 6M
</a>
```

How file links are treated varies by browser, user settings, installed applications, file type, and other factors. A link to an image usually opens in the same window or tab, while a link to

a document may prompt a download or open an external application automatically.

You can force a download by using the download attribute when placing the link. However, remember that the presence of the attribute doesn't convey any information to users, including screen reader users, so the link text must undertake that task, as illustrated in Example 3-7. Whether you want to use the download attribute depends on the context. If you link to a file the browser supports, like an HTML file, adding the attribute would make downloading it easier for users. On the other hand, using the attribute can limit users' choice of how to access a file.

WARNING

The `download` attribute works only for same-origin URLs or the `blob:` and `data:` schemes.

3.4 Create Email Links

Problem

Clicking an email link may initiate a change of context as the user moves from the browser to a mail application if there's one

installed. That can be useful but isn't always desirable because switching between applications can be tedious for some users.

Solution

Show the email address directly and link it using the `mailto:` URI scheme, as illustrated in Example 3-15.

Example 3-15. A link to an email address

```
<a href="mailto:support@johannastoys.com">
  support@johannastoys.com
</a>
```

You can link to multiple email addresses, as shown in Example 3-16.

Example 3-16. A link to multiple addresses

```
<a href="mailto: manuel@matuzo.at, office@matuzo
  manuel@matuzo.at and office@matuzo.at
</a>
```

You can even prefill the subject and body of the email, as shown in Example 3-17.

Example 3-17. Subject and body prefilled

```
<a href="mailto:support@johannastoys.com?subject=
        &body=Customer%20number%3A%20068303">
  support@johannastoys.com
</a>
```

Discussion

It can be convenient for users when an email address is linked because clicking it brings them to their mail application, and they can start writing immediately. Some users may want to avoid that change of context and only copy an email address. You can serve both requirements by using the email address as the link text.

I recommend avoiding generic link text like “Click here to get in touch” or “Contact us,” as shown in Example 3-18, because if the user wants only to know the email address, they would have to click the link, move to the email application or webmail browser tab, and copy it from the address field.

Example 3-18. Bad practice: Generic link text

```
If you want to learn more about our products,
```

```
<a href="mailto:support@johannastoys.com">contac
```

If you want to learn more about our products, contact us.

It's more user-friendly if the email address is immediately visible:

If you want to learn more about our products, contact us at support@johannastoys.com.

If you want the email to have a particular subject or include data about the user or their inquiry, you can reduce the amount of manual typing they have to do by prefilling the email. The `mailto:` URI scheme supports several parameters. You can prefill the subject by appending a `?` to the email address, followed by `subject=` and the value, as shown in Example 3-19.

Example 3-19. Subject prefilled

```
<a href="mailto:support@johannastoys.com?subject=
  support@johannastoys.com
</a>
```

You can also prefill the message of the email by appending the `body` field, as shown in Example 3-20.

Example 3-20. Body prefilled

```
<a href="mailto:support@johannastoys.com?body=Me
  support@johannastoys.com
</a>
```

In Example 3-17, you can see how to prefill more than one parameter, and Figure 3-8 shows how the result looks in Apple Mail.

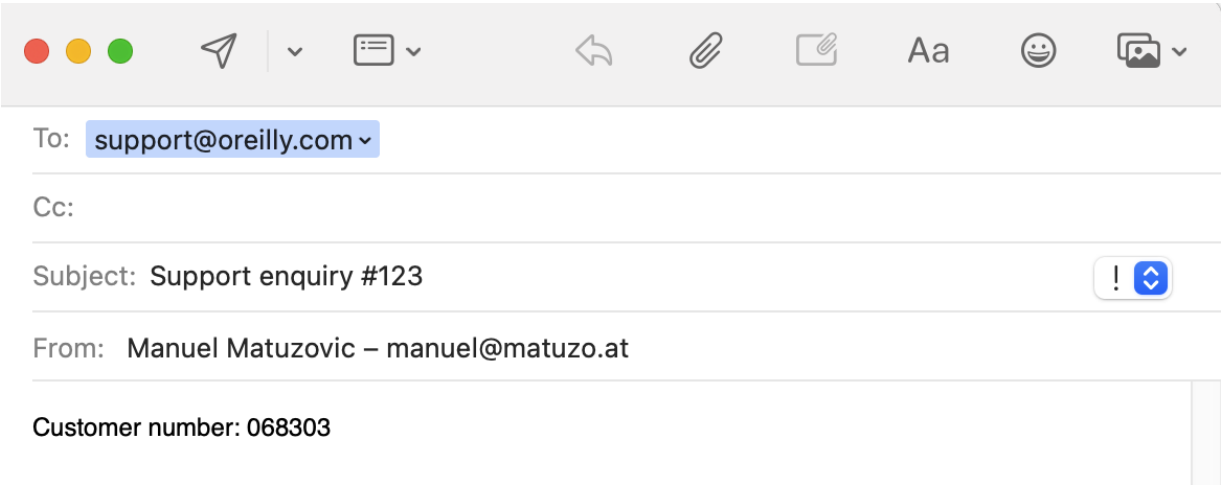


Figure 3-8. Subject and body prefilled in an email opened via click on a link

NOTE

Special characters, line breaks, and spaces must be converted to a format that all browsers and servers understand (URL encoding).

3.5 Link Images

Problem

Incorrect or missing descriptions of linked images result in confusing or broken links, which can prevent screen reader users from understanding the purpose of the link.

Solution

Use the `alt` attribute of the image to provide the link's label or accessible name. Example 3-21 shows a linked logo in the header of a page. The image's description is "Home page" because that's where the link points to. A common error is to use a description like "Company Logo." This image is functional and thus must describe the link's purpose, not the image's content.

Example 3-21. The alternative text of the linked logo serves as the link text

```
<header>
  <a href="/">
    <img src="/images/logo.svg" alt="Home page">
  </a>
</header>
```

Instead of an `img` you can also use a scalable vector graphic (SVG) with a `<title>` element. For cross-browser support, you should use `aria-labelledby` on the SVG and create a reference to the title. Note that some browsers show the content of the `<title>` element in a tooltip when you hover the SVG, as shown in Example 3-22.

Example 3-22. The title of the SVG serves as the text of the download link

```
<a href="/report.pdf" download>
  <svg aria-labelledby="save_title" role="img">
    <title id="save_title">Save</title>
    <path d="…"/>
  </svg>
</a>
```

You can also remove the graphic from the accessibility tree by defining an empty `alt` attribute on the `img` or `aria-`

`hidden="true"` on the SVG. If you do that, the link still needs a text alternative, which you can provide with `aria-label` or `aria-labelledby` on the link itself, as shown in Example 3-23.

Example 3-23. The `aria-label` attribute labels the link

```
<a href="https://mastodon.social" aria-label="Ma
  <img src="/images/mastodon-logo.svg" alt="">
</a>

<a href="https://mastodon.social" aria-label="Ma
  <svg aria-hidden="true">
    <path d="…"/>
  </svg>
</a>
```

Discussion

I'm dedicating an entire recipe to this issue because it's so prevalent and affects users significantly. If you don't use the image to provide an accessible name for the link or label the link in any other way, a screen reader announces the URL, parts of it, something like "unlabeled image," or the image's filename.

Images on the web can be roughly divided into three categories: informative, decorative, and functional.

Informative images represent information that is relevant to the main content of the page or a section. Describe those kinds of images briefly by explaining the meaning or the content displayed. How long the description gets depends on the information and context you want to convey. In Figure 3-9 and Example 3-24, you can see an example of an informative image and its description.

Figure 3-9. A photo of the St. Charles Church (Karlskirche) in Vienna

Example 3-24. Description of an informative image

```
<img src="images/karlskirche.jpg" alt="The St. C
Vienna, a baroque building containing a dome and
```

Decorative images purely serve the visual design and contribute little or nothing in terms of content, so they do not require an alternative text. In Figure 3-10 and Example 3-25, you can see an example of a decorative image and its description.

Figure 3-10. An image that contributes to the visual aesthetics but doesn't contain relevant information

Example 3-25. The blank `alt` attribute hides the decorative image from screen readers

```
<img src="images/headphones.jpg" alt="">
```

In some situations, decorative images don't contribute to understanding the content, but they communicate emotions. In these cases, when you want to create a particular *atmosphere*, it can be helpful to describe decorative images. As Léonie Watson puts it in her blog post, "Text descriptions and emotion rich images":

> *Like sighted users, we'll skip around the content of the page until we find something that interests us. If the first few syllables of an alt text sound promising, we'll pause to read. If they don't, we'll move on to the next element on the page. Also like sighted users, we're often likely to pause on something unimportant, but which captures our imagination.*
>
> —Léonie Watson

Functional images are images in links or buttons whose alternative text serves as the text for the link or button. With

this type of image, you don't describe the image's content but the action it initiates.

According to the WebAIM Million 2024 report, which is the result of a yearly automated accessibility evaluation of the top 1 million websites, 44.6% of tested websites contained empty links.

There are at least two root causes for these issues. One is literal empty links—those not containing any content, as shown in Example 3-26.

Example 3-26. Bad practice: An empty link with no accessible name

```
<a href="/products"></a>
```

The other cause is wrapping images in links or buttons without providing any text alternative, as shown in Examples 3-27 and 3-28.

Example 3-27. Bad practice: An image without an `alt` attribute wrapped in a link

```
<a href="https://webaim.org/projects/million/">
  <img src="/images/screenshot-04-03-23_copy.jpg
```

```
  </a>
```

I tried the code in Example 3-27 with different screen readers and browsers. Table 3-1 shows how they announce it.

Table 3-1. Screen reader test: Linked image with no `alt`

Screen reader	Browser	Screen reader narration
NVDA	Firefox	**million, graphic, link**
JAWS	Firefox	**H T T P S colon slash slash webaim dot org slash projects slash million slash**
JAWS	Edge	**images slash screenshot dash 04 dash 03 dash 23 underline copy, link, graphic**
TalkBack	Chrome	**screenshot the 3rd of April 23 underscore copy, link**
VoiceOver macOS	Safari	**link, million**
VoiceOver macOS	Chrome	**unlabeled image**
VoiceOver macOS	Firefox	**link, image, million**

Screen reader	Browser	Screen reader narration
VoiceOver iOS	Safari	**million, link**

If you add an `alt` attribute to an image but leave it empty, a screen reader announces the URL (or parts of it).

Table 3-2 shows how different screen readers and browsers announce the code in Example 3-28. At best, screen reader users can only guess what they can expect when they click an empty link.

Example 3-28. An image with an empty `alt` attribute wrapped in a link

```
<a href="https://webaim.org/projects/million/">
  <img src="/images/screenshot-04-03-23_copy.jpg
</a>
```

Table 3-2. Screen reader test: Linked image with empty `alt`

Screen reader	Browser	Screen reader narration
NVDA	Firefox	**million, link**
JAWS	Firefox	**link, H T T P S colon slash slash webaim dot org slash projects slash million slash**
JAWS	Edge	**million, link**
TalkBack	Chrome	**million, link**
VoiceOver macOS	Safari	**link, million**
VoiceOver macOS	Chrome	**link, million**
VoiceOver macOS	Firefox	**link, million**
VoiceOver iOS	Safari	**million, link**

With informative images, you explain the content, and with functional images, you describe what the link or button does. Whether a specific description of an image is appropriate depends on the context, but both informative and functional images must have a text alternative.

3.6 Inform Users of Changing Context

Problem

When a link opens in a new tab or window unexpectedly, it can cause several problems for users:

- It can confuse and disorientate people with difficulty perceiving visual content or with cognitive disabilities.
- It's not always clear that a link has opened in a new tab, especially on mobile browsers, which can be disorientating.
- The back button might not work because not all browsers share the browsing history of a session across tabs. If the user can't use the back button, they must find their way back to the previous page.
- Less-technical users might not understand how to navigate across windows and tabs.
- On top of that, it clutters the user's information space.

Solution

If opening a link in a new tab or window will improve the user experience and you use `target="_blank"` on a link, you should inform users in the link text that the link will open in a new tab, as shown in Example 3-29.

Example 3-29. A link text that includes a warning in parenthesis

```
<a href="https://www.mxb.dev" target="_blank">
  Max Böck's website (opens in new tab).
</a>
```

If you can't add warnings manually and have no server-side automation logic in place, you can use CSS. It's better than having no warning, but there are downsides to this solution:

- On sites with multiple languages, you must manage translations of the warning text in CSS and not in HTML or a database. That's unintuitive and a source of error because you usually don't expect text content in CSS.
- Auto-translation tools might not translate pseudocontent.
- The information is not available if CSS fails to load or the page is presented without CSS.

Example 3-30 looks like an elegant solution, but I recommend using text in the element directly instead of text coming from a pseudoelement.

Example 3-30. A warning added via a pseudoelement

```
<a href="https://www.mxb.dev" target="_blank">
  Max Böck's website.
</a>
```

```
[target="_blank"]::after {
  content: " (opens in new tab)";
}
```

NOTE

A regular element in HTML is one you add to a page directly in your HTML document or via JavaScript. A pseudoelement is not part of your HTML document but gets added to the DOM via CSS.

Discussion

Is it a good idea to force open links in new tabs or windows? Depending on who you ask, you will get very different answers. Some people favor opening external links in new windows

because it keeps users on your site. What usability and accessibility experts agree on is that you should avoid it (with few exceptions) and inform users if you don't.

That links opening in new windows or tabs can be problematic for usability, and accessibility is not a new insight. Jakob Nielsen spoke against it early on in "The Top 10 Web Design Mistakes of 1999". Since then we've seen far more handheld devices like smartphones and tablets, where space is limited, and keeping track of open tabs and windows is even more challenging.

Besides the issues listed in the problem section of this recipe, one of the strongest arguments against opening links in a new tab is that when you apply `target="_blank"` you're taking away the user's ability to decide how to open a link. It will always open in a new tab or window (depending on browser settings). Rian Rietveld argues that the browser's default behavior is always the most predictable and, in this case, provides users with more options (open in the same tab or a new tab or window).

Exceptions

The (WCAG) lists scenarios where forcing links to open in a new tab is better for the user experience:

- If a linked page contains context-sensitive information meant to persist alongside the main content, such as instructions for filling out a form or other reference documents, opening the page in the same tab can disrupt a multistep workflow.
- If a link opens widgets in a pop-up window, such as a calendar-based date picker or a login, you may want users to perform these actions outside the main page.
- If a user is logged in, linking to a page outside the secured area could terminate the user's session.

In those cases where opening a new tab by default helps users, provide a warning. This allows them to decide whether they want to leave the current window, and it will help them with orientation and navigation.

The most reliable way to add a warning is using text and a quick heads-up, like "Opens in new tab," as shown in Examples 3-29 and 3-30. Both solutions add content to the original label and don't overwrite it. A common bad practice is using `aria-label` for the task, as shown in Example 3-31. `aria-label` overwrites the original text and makes the link pretty much useless because its accessible name is now "Opens in new tab."

Example 3-31. Bad practice: `aria-label` overwrites the original label

```
<a href="https://www.mxb.dev" target="_blank" ar
  Max Böck's website.
</a>
```

You might be tempted to use an icon instead of text, but finding an appropriate one isn't easy. A typical icon for that use case is a rectangle with an arrow pointing out the top-right corner (shown in Figure 3-11).

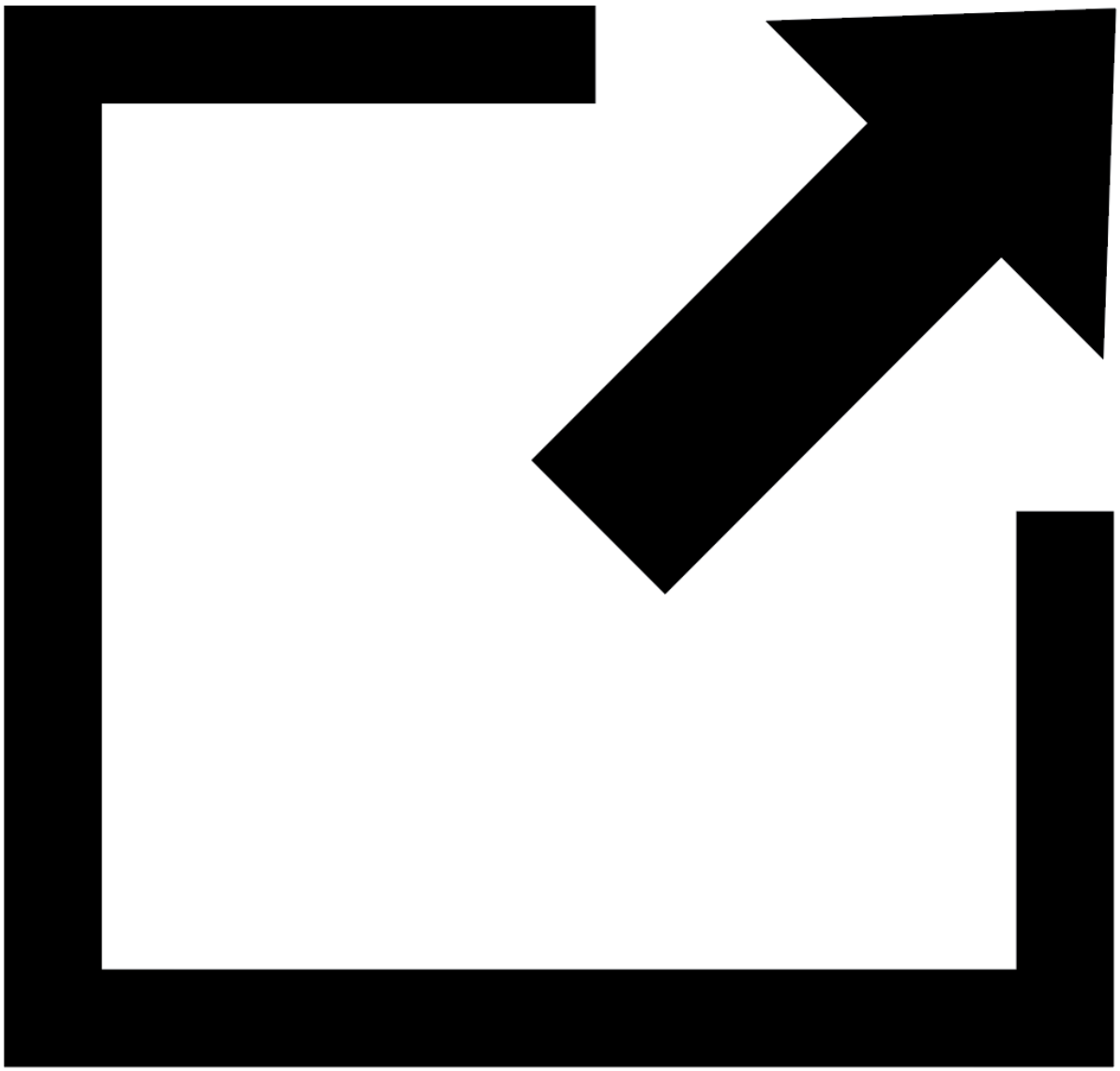


Figure 3-11. Icon often used to indicate external links and/or links opening in new tabs

The problem with that icon is that people interpret it differently. It can mean “opens in new tab,” “opens external site,” “opens external site in new tab,” “fullscreen,” “pop-out,” “share,” and more. Researchers at *gov.uk* found that the icon doesn’t provide users any value, so they removed it.

If you set a link to open in a tab, always base your decision on what's best for your users. SEO or key performance indicators (KPIs) shouldn't be a factor. If possible, back up your choices with usability testing or contextual inquiries.

3.7 Fix Client-Side Rendering

Problem

Different JavaScript frameworks and plug-ins implement client-side routing differently, but they all have in common that they disable the browser's default site-navigation mechanism and inject content into the DOM. Managing routing manually can result in unexpected or indistinguishable changes. For users who primarily rely on keyboard accessibility, focus might not be where they would expect it. It can even be lost altogether, which can make it much harder to navigate. Screen reader users might not notice that the content of a page has changed due to the lack of feedback.

Solution

The most obvious answer to this problem is to avoid causing it in the first place and rely on the default site navigation

behavior of the browser. But depending on the requirements or the technology stack, that's not always possible or desirable.

Let's start with a simple example. You disable the browser's default behavior, and instead of reloading the page, you append content from a template into the main element (see Examples 3-32 and 3-33).

Example 3-32. A navigation, an empty main element, and the page content of both linked pages stored in a `template` tag

```
<header>
  <nav>
    <ul>
      <li><a href="#page-home">Home</a></li>
      <li><a href="#page-about">About</a></li>
    </ul>
  </nav>
</header>

<main>
</main>

<template id="page-home">
  <h1 id="heading">Home</h1>
</template>
```

```
<template id="page-about">
  <h1 id="heading">About Us</h1>
</template>
```

Example 3-33. Incomplete solution: A click event prevents the default behavior of links and clones the content of the template with a matching `id` into the main element

```
const main = document.querySelector("main");
const nav = document.querySelector("nav");

nav.addEventListener("click", (e) => {
    if (e.target.nodeName === "A") {
        e.preventDefault();
        const id = e.target.getAttribute("href")
        const page = document.querySelector(id);

        main.innerHTML = "";
        main.appendChild(page.content.cloneNode(
    }
});
```

That works well for mouse and touch users, but screen reader users don't get feedback when DOM content changes, and the focus is still on the link.

Focus management

One solution to this problem is to move focus from the link to a labeled semantic element, which could be the main element, as in Examples 3-34 and 3-35.

Example 3-34. A simple client-side routing solution with focus management

```
<header>
  <nav>
    <ul>
      <li><a href="#page-home">Home</a></li>
      <li><a href="#page-about">About</a></li>
    </ul>
  </nav>
</header>

<main tabindex="-1" aria-labelledby="heading"> ❶
</main>

<template id="page-home">
  <h1 id="heading">Home</h1>
</template>

<template id="page-about">
  <h1 id="heading">About Us</h1>
</template>
```

- ❶ The main element is focusable and labeled by its child `<h1>`.

Example 3-35. The script focuses the main element after appending the template content

```
const main = document.querySelector("main");
const nav = document.querySelector("nav");

nav.addEventListener("click", (e) => {
    if (e.target.nodeName === "A") {
        e.preventDefault();
        const id = e.target.getAttribute("href")
        const page = document.querySelector(id);

        main.innerHTML = "";
        main.appendChild(page.content.cloneNode(
        main.focus(); ❶
    }
});
```

- ❶ Moves focus to the main element

When a user clicks a link, the focus moves to the main content area of the page. A welcome side effect of focusing an element

with JavaScript is that screen readers automatically announce its role and accessible name. Depending on the screen reader and browser, this example announces something like “Main, Home” or “Main, About.”

Live regions

Another solution to this problem is to report page changes using a live region. Live regions are perceivable regions that announce changes to their content to screen readers due to an external event when user focus may be elsewhere. They can be useful for communicating DOM changes, user interaction results, waiting state, process progress, errors, or page changes. Examples include chat logs, stock tickers, form validation, or client-side routing.

Every time you change the content of the element with `role="status"` (this attribute and value turns it into a *live region*), a screen reader automatically announces it. Examples 3-36 and 3-37 show a simple client-side routing solution using a live region.

Example 3-36. A simple client-side routing solution with a live region

```
  <header>
```

```
    <nav>
      <ul>
        <li><a href="#page-home">Home</a></li>
        <li><a href="#page-about">About</a></li>
      </ul>
    </nav>
  </header>

  <main></main>
  <div role="status" class="visually-hidden"></div

  <template id="page-home">
    <h2>Home</h2>
  </template>

  <template id="page-about">
    <h2>About Us</h2>
  </template>
```

```
  <style>
  .visually-hidden { ❷
    clip-path: inset(50%);
    height: 1px;
    overflow: hidden;
    position: absolute;
    white-space: nowrap;
    width: 1px;
```

```
}
</style>
```

1. Additional element to announce changes on the page (the live region).
2. Moves an element visually out of the viewport keeping it in the accessibility tree. You can learn more about hiding content in Recipe 8.1.

Example 3-37. The script changes the content of the live region after appending the template content

```
const main = document.querySelector("main");
const nav = document.querySelector("nav");
const region = document.querySelector('[role="st

nav.addEventListener("click", (e) => {
    if (e.target.nodeName === "A") {
        e.preventDefault();
        const link = e.target;
        const id = link.getAttribute("href");
        const page = document.querySelector(id);


        main.innerHTML = "";
        main.appendChild(page.content.cloneNode(
        region.textContent = `${link.textContent
```

```
    }
});
```

1. Changes the content of the live region every time the user clicks a link

The focus is still on the link, so consider combining both solutions if necessary, but remember that this might cause redundancy: a screen reader might convey the content in the live region *and* feedback from focusing an element.

These are just examples. There are many different ways of implementing and combining both solutions, and you can only determine what works best by testing with users.

Discussion

Most elements in HTML are accessible by default, meaning that they provide the semantic information and functionality needed for most users out of the box. Aside from a few exceptions, relying on the default behavior of HTML elements is a safe bet. When you disable these defaults, you must ensure that your custom solution provides a similar or better user experience.

Let's take page navigation as an example. When a user clicks on a link, the browser loads a new page, changes the document displayed in the viewport, and puts focus on the document. If they're using a screen reader, it also announces the page title (`<title>`), which should be unique on each page. No matter how they access the website, they usually always know what happened, that something happened, and where they are. None of these things happen when you prevent the default click behavior on a link. You must either re-create them manually or provide a similar or better alternative.

Focus management

Is the default browser behavior of focusing on the document when it loads a new page best for the user experience? That's debatable, according to research done by Marcy Sutton. Comparing different solutions, she found that it depends on the user's browsing method. Still, in a custom client-side rendered solution, you have to manage focus somehow and move it to a focusable and semantic element. To do that, put the `tabindex` attribute on it and set the value to `-1`, which ensures you can focus it via JavaScript's `focus()` method but not with the `Tab` key.

TIP

Set `tabindex="-1"` to make an element focusable via JavaScript and `tabindex="0"` to make it focusable and tabbable. `tabindex="-1"` is what you want to use in most cases because you should make a noninteractive element interactive only as a last resort.

The focusable element mustn't be a generic element, like the `div` because screen readers handle them differently and you cannot name them. See Example 3-38.

Example 3-38. Bad practice: Don't put focus on an empty `div`

```
<div tabindex="-1"></div>
```

The focusable element could be the `<main>` element, as shown in Example 3-34, a `section`, or a heading. It ideally has an accessible name, either coming from its content or `aria-label`, as shown in Example 3-39. You can also use `aria-labelledby` to create a reference to an existing element. In the case of Example 3-34, that would be the heading of each page.

Example 3-39. A focusable labeled section

```
<section tabindex="-1" aria-label="Products"></s
```

Live regions

You can create and configure live regions in different ways. The most relevant attributes are `aria-live`, `aria-atomic`, and `aria-relevant`.

You can turn an element into a live region by setting the `aria-live` attribute to `assertive` or `polite`. Assertive updates have the highest priority and should be presented immediately, interrupting ongoing announcements if necessary. Polite updates are less aggressive and wait for the next possible opportunity, like the end of a sentence or typing pauses.

By default, only the changed DOM nodes will be presented because `aria-atomic` is `false`. In Example 3-40, you change the value of a `<span>` element within a live region on click. The screen reader only announces the content of the changed DOM node; for example, "Olivia," not the whole sentence. It also only does that if the text changes. There is no announcement if the current value is "Philippa" and the new value that overrides it is the same.

Example 3-40. Updating a nested node in a live region

```
<button>Change name</button>
```

```
<button>Change name</button>

<div aria-live="polite">
  My name is <span>Claudia</span>
</div>

<script>
const button = document.querySelector("button");

const names = [ ❶
        "Moritz",
        "Valentina",
        "Johanna",
        "Magdalena",
        "Victoria",
        "Philippa",
        "Olivia",
];
const name = document.querySelector("span");

button.addEventListener("click", (e) => {
   const random = Math.floor(Math.random() * name
   name.textContent = names[random]; ❷
});
</script>
```

❶ A list of names

2. Populates the `<span>` element with a randomly picked name

If you add the `aria-atomic` attribute and set it to `true`, you make the live region *atomic*, meaning that screen readers present the entire region, for example, "My name is Moritz," in Example 3-41.

Example 3-41. Updating a nested node in an atomic live region

```
<button>Change name</button>

<div aria-live="polite" aria-atomic="true">
  My name is <span>Claudia</span>
</div>
```

There are also dedicated live region roles, for example, `status` and `alert`, and a live region element, `<output>`. `role="status"` is a shorthand for `aria-live="polite"` and `aria-atomic="true"` (see Example 3-42) and `role="alert"` is a shorthand for `aria-live="assertive"` and `aria-atomic="true"`.

Example 3-42. A status-live region

```
<button>Change name</button>

<div role="status">
  My name is <span>Claudia</span>
</div>
```

The output element (Example 3-43) has an implicit `aria-live="polite"` and `aria-atomic="true"`.

Example 3-43. Using the output element as a live region

```
<button>Change name</button>

<output>
  My name is <span>Claudia</span>
</output>
```

Eric Eggert provides more details in "We're ARIA Live", where he also explains `aria-busy` and `aria-relevant`.

The page title

It's not enough to announce a page change: you also have to change the document's `title`. Screen reader users use shortcuts to read the page title, which can be helpful for orientation. (You can learn more about page titles in Recipe 3.3.)

Client-side page navigation is a complex topic. The solutions in this recipe are not ready-made snippets you can implement on your websites. They merely illustrate the underlying problem and its complexity. How you solve it depends on your stack, your framework, and, most importantly, what works best for your users. When you use a framework or pick a router for your application, ensure that they're doing it accessibly or at least provide instructions on how to do it. If you want to dig deeper into this topic, read "What we learned from user testing of accessible client-side routing techniques with Fable Tech Labs" by Marcy Sutton.

3.8 Add Links to Groups of Elements

Problem

It's sometimes desirable to increase a link's click-and-touch area to span multiple elements. That can improve the user experience for mouse and touch users, but can negatively affect the accessibility and UX for keyboard and screen reader users. Depending on the solution, the following problems can arise:

- Text in a link can be lengthy, making using an interface tedious.

- Redundant links pollute the interface and make orientation and navigation more complex for screen reader users.
- Additional tab stops make it more physically demanding for keyboard users to use an interface.
- Empty or verbose links can be hard or impossible for voice users to select.
- Browser defaults like selecting the text, middle-clicking to open a link in a new window/tab, opening the context menu, or previewing URLs might not work.

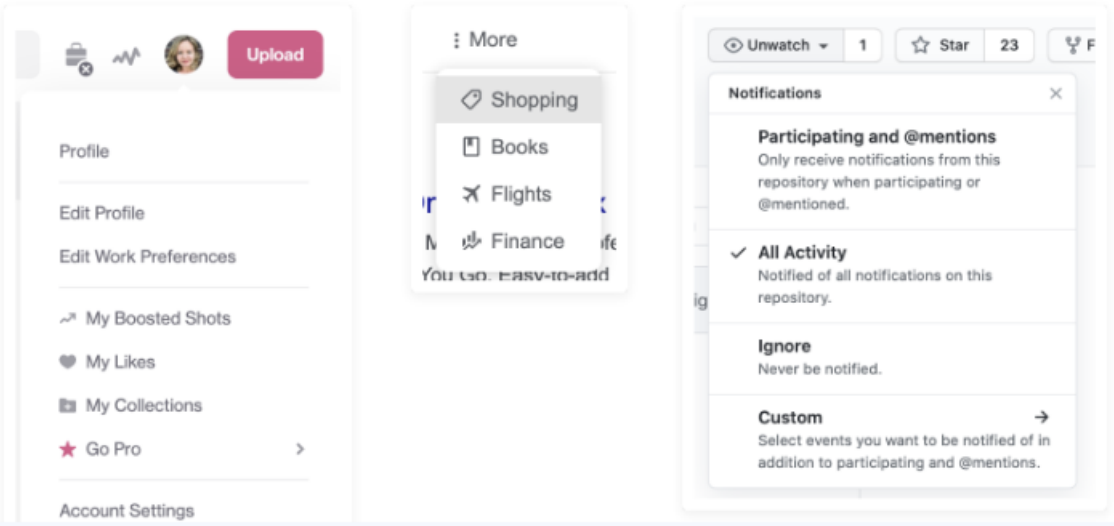
Solution

Unfortunately, there is no perfect way to link groups of elements; the solutions all come with trade-offs. This recipe lists five ways of linking multiple elements, starting with the simplest. If you don't want to go through all of them, skip to solution 4, which I recommend for most use cases.

To illustrate the issue, I have created a card component, a page teaser usually consisting of a heading, image, and text, that I link differently in each example (see Figure 3-12). The code in Example 3-44 is the baseline. The requirements for the card are:

- The whole card should be clickable.
- The announcement in screen readers should be succinct.

- The card should contain as few tab stops as possible.
- The solution should support multiple different links.
- The HTML should be valid.
- The context menu and other browser defaults should work as expected.
- The text should be selectable.



When CSS Isn’t Enough: JavaScript Requirements For Accessible Components

by Stephanie Eckles

Spoiler alert: tooltips, modals, tabs, carousels, and dropdown menus are some of the user interface components that require more than CSS. To ensure accessibility of your interface, JavaScript is a necessary addition to accomplish focus management, respond to keyboard events, and toggle ARIA attributes.

Read “When CSS Isn’t Enough”.

Figure 3-12. A card with a heading, text, image, and a call-to-action link

In the Discussion section of this recipe, you can learn how many of these requirements each solution meets.

Example 3-44. A basic card component

```
<div class="card">
  <h3>
    When CSS Isn't Enough: JavaScript Requiremen
  </h3>

  <p>by <a href="https://thinkdobecreate.com/">S

  <img src="teaser.png"
       alt="Example dropdown menus from Dribbble

  <div>
    <p>
      Spoiler alert: tooltips, modals, tabs, car
      menus are some of the user interface compo
      than CSS. To ensure accessibility of your
      is a necessary addition to accomplish focu
      to keyboard events, and toggle ARIA attrib
    </p>
  </div>


  <p>
```

```
      Read
      <a href="/css-javascript-requirements-access
      “When CSS Isn’t Enough”
      </a>.
    </p>
  </div>

  <style>
    .card {
      --padding: 1rem;

      background-color: hsl(222deg 100% 98%);
      box-shadow: 0 0 7px 3px hsl(0deg 0% 0% / 10%
      display: grid;
      font-family: Seravek, 'Gill Sans Nova', Ubun
      grid-template-columns: var(--padding) 1fr va
      line-height: 1.5;
      max-width: 23rem;
      padding-block-end: var(--padding);
    }

    .card > * {
      grid-column: 2 / -2;
      margin: 0 0 0.6em;
    }


    .card h3 {
      line-height: 1.3;
```

```
    margin-block-end: 0;
  }

  .card h3 a {
    color: initial;
    text-decoration: initial;
  }

  .card :is(img, .img) {
    grid-column: 1 / -1;
    margin-block-end: var(--padding);
    max-width: 100%;
    order: -1;
  }

  .card :last-child {
    margin-block-end: 0;
  }
</style>
```

This basic solution meets all of the requirements except the core requirement of being clickable as a whole.

Let's try to link the card.

Solution 1: Wrapping all elements in an <a> element

You could wrap all elements in the card in a single `<a>` element. That is a common pattern, but it has significant negative consequences for screen reader users. That's why I've not included the code in the Solutions section. You can read more about it and look at the code in the Discussion section.

Solution 2: Separate links

Instead of wrapping all elements in a link, link most elements separately, as shown in Example 3-45.

Example 3-45. Card component with separate links

```
<div class="card">
  <h3>
    <a
    href="/css-javascript-requirements-accessibl
      When CSS Isn't Enough: JavaScript Requirem
    </a>
  </h3>

  <p>by <a href="https://thinkdobecreate.com/">S

  <a class="img"
  href="/css-javascript-requirements-accessible-

    <img src="teaser.png"
         alt="Example dropdown menus from Dribbb
```

```
    </a>

    <div>
      <p>
        Spoiler alert: tooltips, modals, tabs, car
        some of the user interface components that
        accessibility of your interface, JavaScrip
        accomplish focus management, respond to ke
        ARIA attributes.
      </p>
    </div>

    <p>
      Read
      <a
      href="/css-javascript-requirements-accessibl
      “When CSS Isn’t Enough”
      </a>.
    </p>
  </div>
```

Example 3-46 is a variation that reduces the number of tab stops.

Example 3-46. Card component with separate links with `tabindex="-1"` and `role="presentation"`

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

```
<div class="card">
  <h3>
    <a tabindex="-1" role="presentation"
    href="/css-javascript-requirements-accessibl
      When CSS Isn’t Enough: JavaScript Requirem
    </a>
  </h3>

  <p>by <a href="https://thinkdobecreate.com/">S

  <a class="img" tabindex="-1" role="presentatio
  href="/css-javascript-requirements-accessible-
    <img src="teaser-image.png"
    alt="Example dropdown menus from Dribbble, G
  </a>

  <div>
    <p>
      Spoiler alert: tooltips, modals, tabs, car
      of the user interface components that requ
      accessibility of your interface, JavaScrip
      accomplish focus management, respond to ke
      toggle ARIA attributes.
    </p>
  </div>

  <p>
    Read <a href="/css-javascript-requirements-a
         “When CSS Isn’t Enough”</a>.
```

```
    </p>
  </div>
```

Using `tabindex="-1"` and `role="presentation"` on redundant links sounds like a clever solution, but there are caveats, addressed in the Discussion section of this recipe.

Solution 3: Empty link

To avoid verbose announcements and invalid HTML, put an empty link on the same level as the card content and position it above the card, as shown in Example 3-47.

Example 3-47. Card component with an empty link covering the card

```
<div class="card">
  <a class="solution-3-link"
     href="/css-javascript-requirements-accessib
  aria-labelledby="solution3-heading"></a>

  <h3 id="solution3-heading">
    When CSS Isn't Enough: JavaScript Requiremen

  </h3>

  <p>by <a href="https://thinkdobecreate.com/">S
```

```
    <img src="teaser.png"
         alt="Example dropdown menus from Dribbble

    <div>
      <p>
        Spoiler alert: tooltips, modals, tabs, car
        menus are some of the user interface compo
        than CSS. To ensure accessibility of your
        is a necessary addition to accomplish focu
        respond to keyboard events, and toggle ARI
      </p>
    </div>

    <p>
      Read
      <a
      href="/css-javascript-requirements-accessibl
      “When CSS Isn’t Enough”
      </a>.
    </p>
  </div>

  <style>
    .solution-3-link {

      grid-column: 1 / -1;
      width: 100%;
      height: 100%;
```

```
      position: absolute;
      inset: 0;
    }
  </style>
```

Solution 4: Pseudoelement

To avoid redundancy, verbosity, and empty links, add a pseudoelement to an existing link and position the pseudocontent above the card content, as shown in Example 3-48.

Example 3-48. Card component with a pseudoelement within the link covering the card

```
<div class="card">
  <h3>
    When CSS Isn’t Enough: JavaScript Requiremen
  </h3>

  <p class="author">
     by <a href="https://thinkdobecreate.com/">S
  </p>

  <img src="teaser-image.png"
       alt="Example dropdown menus from Dribbble
```

```
    <div>
      <p>
        Spoiler alert: tooltips, modals, tabs, car
        menus are some of the user interface compo
        than CSS. To ensure accessibility of your
        is a necessary addition to accomplish focu
        respond to keyboard events, and toggle ARI
      </p>
    </div>

    <p>
      Read <a class="readmore"
              href="/css-javascript-requirements-a
      “When CSS Isn’t Enough”
      </a>.
    </p>
  </div>

  <style>
    .card .readmore::after {
      content: "";
      display: block;
      inset: 0;
      position: absolute;


    }
  </style>
```

To make specific elements selectable, you can use `z-index` to put them above the pseudoelement. Keep in mind that they won't be linked anymore when you do that, as shown in Example 3-49.

Example 3-49. Moving selected elements up a layer

```
.card .author,
.card > div
  z-index: 1;
}
```

Solution 5: JavaScript

The markup is the same as in the base component in Example 3-44 and you use JavaScript to make the entire card clickable. I've picked two different solutions to show you how you can achieve that.

In this Example 3-50, a solution by Heydon Pickering, you save the current time on *mousedown* and *mouseup*. On *mouseup*, you subtract the former from the latter. If the result is less than 200 ms, it's likely that the user didn't select the text but just clicked it; and if it's more, they're probably selecting it.

Example 3-50. Using a timing threshold to detect whether a text was clicked or selected

```
const card = document.querySelector(".card");
let down,
    up,
    link = card.querySelector(".readmore");

card.onmousedown = () => (down = +new Date());
card.onmouseup = () => {
  up = +new Date();
  if (up - down < 200) {
    link.click();
  }
};
```

Vikas Parashar uses the `window.getSelection()` API in his solution to detect whether text has been selected, as shown in Example 3-51.

Example 3-51. Using the selection API to detect whether a text was clicked or selected

```
const card = document.querySelector(".card");
const link = card.querySelector(".readmore");
```

```
  card.addEventListener("click", (e) => {
    const noTextSelected = !window.getSelection().
    if (noTextSelected) {
      link.click();
    }
  });
```

Discussion

As already mentioned, none of the solutions is perfect. Let's discuss their pros and cons.

Solution 1: Wrapping all elements in an <a> element

It's valid to wrap multiple elements in an `<a>` element, as shown in Example 3-52.

Example 3-52. Bad practice: Card component with all elements nested in a link

```
  <div class="card">
    <a
    href="/css-javascript-requirements-accessible-
      <h3>
        When CSS Isn't Enough: JavaScript Requirem
      </h3>
```

```
    <p>by Stephanie Eckles</p>

    <img src="teaser.png"
    alt="Example dropdown menus from Dribbble, G

    <div>
      <p>
        Spoiler alert: tooltips, modals, tabs, c
        menus are some of the user interface com
        more than CSS. To ensure accessibility o
        is a necessary addition to accomplish fo
        respond to keyboard events, and toggle A
      </p>
    </div>

    <p>Read “When CSS Isn’t Enough”</p>
  </a>
</div>

<style>
  /* Card styles same as in baseline component *

  .card {
    display: block;

  }

  .card > a {
```

```
    color: initial;
    display: grid;
    grid-template-columns: inherit;
    text-decoration: none;
  }

  .card > a > * {
    grid-column: 2 / -2;
    margin: 0 0 0.6em;
  }
</style>
```

This solution is simple to implement, but it has serious downsides.

Pros

- The whole card is clickable.
- Browser defaults like right- or middle-click work as expected.
- There are no redundant links.

Cons

- If you access the link with the `Tab` key, a screen reader reads the entire text contained within the link, the heading, the alt text of the image, the teaser text, and the call to action. Some screen readers tell users it's a link only when they're done announcing. NVDA, for example, reads:

  > *When CSS Isn't Enough: JavaScript Requirements For Accessible Components by Stephanie Eckles Example dropdown menus from Dribbble, Google search, and GitHub Spoiler alert: tooltips, modals, tabs, carousels, and dropdown menus are some of the user interface components that require more than CSS. To ensure accessibility of your interface, JavaScript is a necessary addition to accomplish focus management, respond to keyboard events, and toggle ARIA attributes Read "When CSS Isn't Enough," link.*

- Using the virtual cursor, a screen reader might announce pieces of text as separate links. Again, in NVDA:

> *link, Spoiler alert: tooltips, modals, tabs, carousels, and dropdown menus are some of the user interface. link, components that require more than CSS. To ensure accessibility of your interface, JavaScript is a link, necessary addition to accomplish focus management, respond to keyboard events, and toggle ARIA attributes*

- You have to remove the nested links because it's not valid to nest interactive elements, and it breaks the layout.

- The text isn't selectable.

- It's not clear how to access the link via voice.

- Especially on mobile, users might be unable to access the nested elements separately.

The verbosity you get when you access the link using the `Tab` key can be mitigated by labeling the link with `aria-label` or

`aria-labelledby`, as shown in Example 3-53, but all the other issues persist.

Example 3-53. Card component with all elements wrapped in an `<a>` element with explicit accessible name

```
<a href="/css-javascript-requirements-accessible
   aria-labelledby="solution2-heading">
  <h3 id="solution2-heading">
    When CSS Isn't Enough: JavaScript Requiremen
    Components
  </h3>
  …
</a>
```

Solution 2: Separate links

The two significant downsides of this solution are that it creates many redundant links with the same target and it increases the number of tab stops.

You could put `tabindex=-1` on the additional links to reduce the number of tab stops, as illustrated in Example 3-46, but this breaks the third rule of ARIA. All interactive element controls must be usable with the keyboard. You could break the second rule of ARIA and change the role of the link to `none` or

`presentation`. In theory, you would get a nonfocusable, generic text element that behaves like a link for mouse, touch, and voice users. In practice, that doesn't work, because accessibility APIs ignore `role=presentation` and `role=none` on hyperlinks. That results in a confusing mishmash where links are not discoverable via `Tab`, but they are with the virtual cursor.

Pros

- There is no verbose announcement when using the `Tab` key with a screen reader.
- Parts of the card are clickable.
- Browser defaults like right- or middle-click work as expected.
- The card supports multiple links.
- It's accessible via voice.

- The text is selectable.

Cons

- Only parts of the card are clickable.
- The solution adds additional tab stops.
- The card contains many redundant links with the same target.

Solution 3: Empty link

This solution is similar to Solution 1 except that it doesn't wrap all the other elements. The link is empty, and you position it above them using CSS. That solves the verbosity issue, but text is still not selectable, and the solution doesn't support multiple links.

Pros

- There is no verbose announcement when using the `Tab` key with a screen reader.

- The whole card is clickable.
- There’s only one tab stop.
- Browser defaults like right- or middle-click work as expected.
- There are no redundant links.
- It’s accessible via voice.
- Other elements within the card are accessible/discoverable.

Cons

- The solution doesn’t support multiple different links.
- The text isn’t selectable.

- The non-clickable *Read "When CSS Isn't Enough"* text can be confusing.

- Empty elements are a bad practice.

Solution 4: Pseudoelement

This solution is my favorite for most use cases because it doesn't require additional markup, and it ticks off most items on the list of requirements. The only problem is that the pseudoelement overlays all the other elements, which means that text isn't selectable. You can move parts above the pseudolayer by applying a higher `z-index`, making them selectable, but they're not clickable anymore (see Figure 3-13).

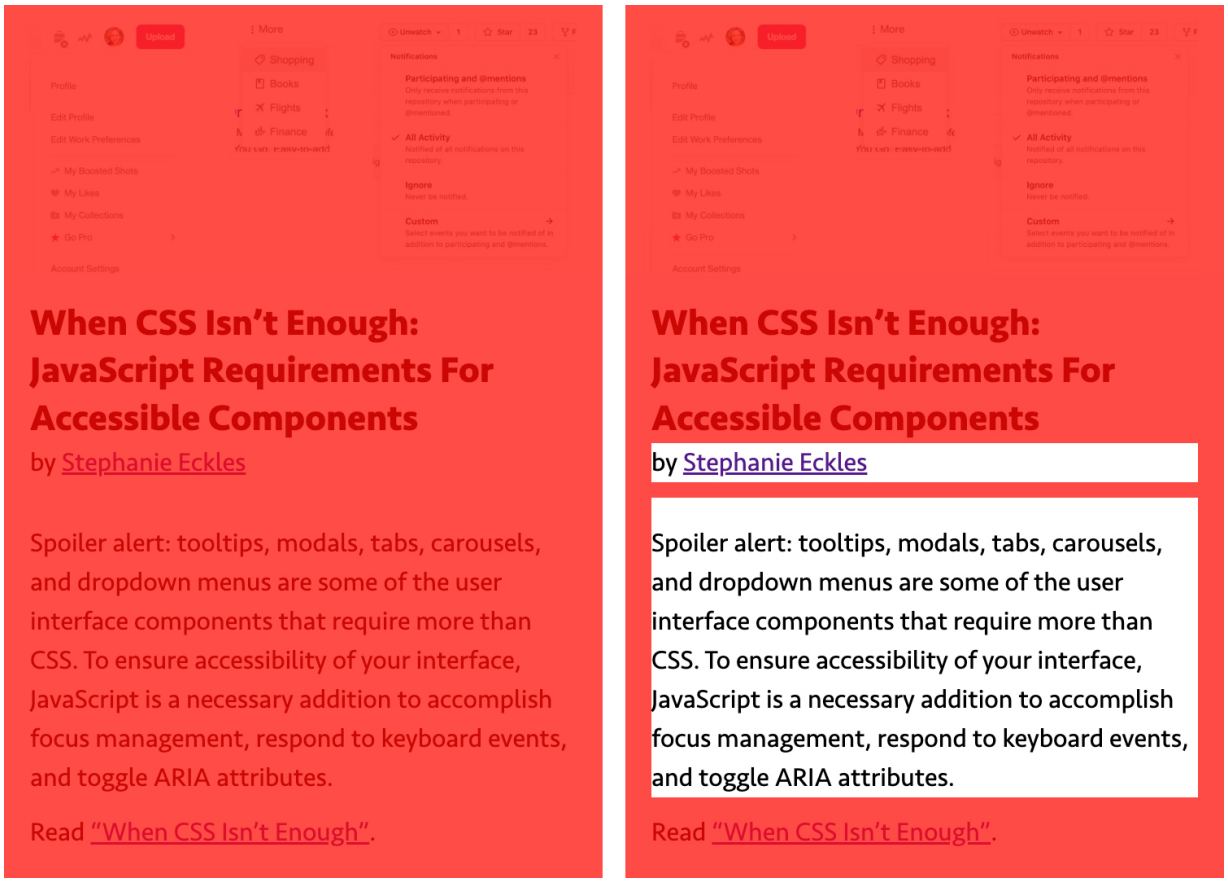


Figure 3-13. On the left, the pseudoelement (shown in this screenshot with a translucent background color) overlays all elements; on the right, two paragraphs lie above the pseudoelement

Pros

- There is no verbose announcement when using the `Tab` key with a screen reader.
- The whole card can be clickable.
- There are only two tab stops.

- Browser defaults like right- or middle-click work as expected.

- The solution supports multiple different links.

- It’s accessible via voice.

- Other elements within the card are accessible/discoverable.

- You don’t need an additional HTML element.

Cons

- Text isn’t selectable except when you move elements up a layer using `z-index`, but then you might have large nonclickable areas.

- If parts of the card promoted to a higher layer contain interactive elements like links, it’s possible that people mistap and click the card instead of the link.

Solution 5: JavaScript

The biggest downside of this solution is that browser default behavior, like right- or middle-click or URL previews on *hover*, don't work anymore.

Pros

- There is no verbose announcement when using the `Tab` key with a screen reader.
- The whole card can be clickable.
- There are only two tab stops.
- The solution supports multiple different links.
- The HTML would be valid even if there were additional links.
- It's accessible via voice.

- Other elements within the card are accessible/discoverable.

- You don't need an additional HTML element.

- Text is selectable.

Cons

- The context menu and other browser defaults only work on links directly.

- It's a *magic* JavaScript solution. Using clever JavaScript hacks to work around default browser behavior is prone to error.

Depending on your requirements, you probably want to try solution 4 or 5.

See Also

- "Better Link Labels: 4 Ss for Encouraging Clicks" by Kate Moran

- “Creating the perfect link” by Rian Rietveld
- “Block Links, Cards, Clickable Regions, Rows, Etc.” by Adrian Roselli
- “Block Links Are a Pain (and Maybe Just a Bad Idea)” by Chris Coyier
- “Teaser with multiple links” by Michael Scharnagl
- “Block Links: The Search for a Perfect Solution” by Vikas Parashar
- “Cards” by Heydon Pickering
- “Building a Good Download…Button?” by Eric Bailey
- “Showing File Types in Links” by Adrian Roselli
- Accessibility in Angular: Routing
- SvelteKit: Route announcements
- NextJS: Route announcements

Chapter 4. Performing Actions

This whole chapter focuses on a single HTML element, the `<button>`. There are two reasons I give buttons so much space.

First, the button is a common element; you'll find it on most pages: in a contact or newsletter sign-up form, to toggle a mobile navigation, or to close a pop-up dialog.

Second, although web developers need buttons so often, they're bad at implementing them correctly. That's not just based on years of my experience auditing sites others have built; there's also data to confirm it.

That's why this chapter focuses on the most essential requirements of the button element and how to fulfill them.

4.1 Pick the Right Element

Problem

If a button doesn't meet basic requirements, you may exclude one or more groups of users from being able to access it or understand its purpose.

The discussion section of this recipe provides more detail, but the essential accessibility criteria for a button are:

- It must convey its semantic button role programmatically.
- It has a concise and straightforward accessible name (visible text or a text alternative).
- It communicates its state (pressed, expanded, etc.) if necessary.
- It's recognizable as a button.
- Its colors must have enough contrast.
- It must be focusable and allow activation via click, touch, and key events.

Solution

Use the native button element and HTML and ARIA attributes to adjust its features according to the requirements, as shown in Examples 4-1, 4-2 and 4-3.

Example 4-1. A native submit button used in a form to submit data

```
<form>
  <label for="email">Email address</label>
```

```
    <input type="email" id="email" name="email">

    <button>Sign up</button>
  </form>
```

Example 4-2. A button that executes JavaScript

```
<button type="button" onclick="print()">
  Print
</button>
```

Example 4-3. A button communicating its pressed state

```
<button type="button" aria-pressed="true">
  Mute
</button>
```

Discussion

The button element has two main use cases: submitting a form and running JavaScript when a user interacts with it.

A button becomes a submit button when you set its type to *submit* or in the context of a form, as shown in Example 4-1. You usually use a submit button to send form data to a server.

If you set the type to *button*, the button does nothing. You do that if you want to run JavaScript when the user activates the button. For example, when you click the button in Example 4-2, it opens your browser's print dialog. Other common uses are toggling the visibility of other elements (see Chapter 8), opening modal dialogs (see Recipe 11.3), and running other JavaScript functions.

According to the WCAG, a button must meet at least six specific requirements to be accessible and provide great UX. Let's look at each one in turn:

It must convey its semantic button role programmatically.

Your buttons must convey their semantic roles. The safest and most reliable option is to use the `<button>` element, because it has an implicit `button` role.

A screen reader will announce the button's role alongside its name; for example, "Print, button."

It has an accessible name (visible text or a text alternative).

Regardless of whether a button contains text, it must have an accessible name. There are no exceptions to this rule. If you don't label buttons, screen reader and voice users won't know what to do with them.

There are different ways of labeling buttons. Recipe 4.2 describes some of them, but the best thing you can do in most cases is include text that is visible to everyone.

It communicates its state (pressed, expanded, etc.) if necessary.

A button may have different states or control another element's state: If so, it must convey its state or relationship using ARIA attributes. The button in Example 4-3 conveys that some media is muted. You can find more examples in Recipe 4.5.

It's recognizable as a button.

According to Jakob's law, users spend most of their time on other sites, which means that users prefer your site to work the same way as all the other sites they already know. For buttons, this means they should meet the user's visual expectations of a button. If a button looks like a button, it makes it easier for users, especially those with cognitive disabilities, to understand its purpose. That also applies to its accessible name. If identical functions have different accessible names on different pages, the site will be more challenging to use.

Its colors must have enough contrast.

Just like most of the text or images of text on a web page, buttons must also meet minimum contrast requirements. The text color in a button must have a contrast ratio of at least 4.5:1 for normal text or 3:1 for large-scale or bold text.

Other colors used in the button, like the background color or colors of focus indicators, should also meet requirements for minimum contrast ratio against adjacent colors. In other words, the button's background color or the outline of a focus indicator or border should also have a contrast ratio of at least 3:1 against the background, which typically is a parent component or the page itself. Low contrast controls are more difficult to perceive and may be completely missed by people with low vision.

It must be tabbable and allow activation via click, touch, and key events

A button is an *interactive* element, which means that if you can click it, you must also be able to perform the same action using the keyboard by pressing `Enter` or `Space`. For that, it must be tabbable (reachable via the `Tab` key), which is true by default for the `<button>` element.

Most websites contain buttons. Try to follow the best practices in this chapter and avoid several common alternative solutions that don't meet the requirements, listed in Example 4-4. It's possible to re-create the native button's default functionality using a different element, like the `<div>`, but usually it's not worth the effort because the `<button>` comes with most of the features described in this recipe.

Example 4-4. Bad practices: Common inaccessible button alternatives you should avoid

```
<!-- Div with a click event -->
<div onclick="[JS function]">
  O'Reilly books and videos
</div>

<!-- Non-focusable button -->
<div role="button" onclick="[JS function]">
  O'Reilly books and videos
</div>

<!-- Link with a click event -->
<a href="javascript: void(0)" onclick="[JS funct
  Menu

</a>

<!-- Link with a click event -->
```

```
<a href="#" onclick="[JS function]">
  Menu
</a>
```

4.2 Label Buttons Clearly

Problem

Buttons can take various shapes and forms. The content wrapped in the button may be text, an icon, or both. When you don't name a button, regardless of whether the design anticipates it, screen reader users may be unable to tell its purpose.

Solution

First, if the button contains text, that text serves as its label, as shown in Example 4-5.

Example 4-5. Button with the accessible name coming from its content

```
<button type="button">
  Download
```

```
  </button>
```

Second, if the button contains an image, the image's `alt` attribute should provide the label, as in Example 4-6.

Example 4-6. Button with the accessible name coming from the `alt` attribute of the image

```
  <button type="button">
    <img src="/images/download.svg" alt="Download"
  </button>
```

Instead of an `img`, you can also use a an SVG with a `<title>` element. For cross-browser support, use `aria-labelledby` on the SVG and create a reference to the title, as shown in Example 4-7.

Example 4-7. Button with the accessible name coming from the SVG

```
  <button type="button">
    <svg viewBox="0 0 39 44" aria-labelledby="titl
      <title id="title">Download</title>

      <path d="M19.5 36.5 1.6 26.1v-3.6l16.3 9.4V1
      <path d="M1 41.5h37" style="stroke:#000;stro
    </svg>
```

```
  </button>
```

You can also remove the graphic from the accessibility tree by defining an empty `alt` attribute on the `img` or `aria-hidden="true"` on the SVG. If you do that, the button still needs a text alternative, which you can provide with visually hidden text (see Example 4-8), `aria-labelledby`, or `aria-label` (see Example 4-9).

Example 4-8. The visually hidden text labels the button

```
<button type="button">
  <span class="visually-hidden">Download</span>
  <img src="/images/download.svg" alt="" width="
</button>

<!-- or -->

<button type="button">
  <span class="visually-hidden">Download</span>
  <svg viewBox="0 0 39 44" aria-hidden="true" wi
    <path d="M19.5 36.5 1.6 26.1v-3.6l16.3 9.4V1
    <path d="M1 41.5h37" style="stroke:#000;stro
  </svg>

</button>
```

```
<style>
  .visually-hidden {
    clip-path: inset(50%);
    height: 1px;
    overflow: hidden;
    position: absolute;
    white-space: nowrap;
    width: 1px;
  }
</style>
```

Example 4-9. The `aria-label` attribute labels the button

```
<button type="button" aria-label="Save">
  <svg aria-hidden="true" viewBox="0 0 39 44" wi
    <path d="M19.5 36.5 1.6 26.1v-3.6l16.3 9.4V1
    <path d="M1 41.5h37" style="stroke:#000;stro
  </svg>
</button>
```

Removing nested graphics from the accessibility tree is also helpful when you combine an icon with text because the text eliminates the need for an extra label for the icon, as shown in Example 4-10.

Example 4-10. Combination of text and icon

```
<button type="button">
  Save
  <img src="/images/download.svg" alt="" width="
</button>

<button type="button">
  Save
  <svg aria-hidden="true" viewBox="0 0 39 44" wi
    <path d="M19.5 36.5 1.6 26.1v-3.6l16.3 9.4V1
    <path d="M1 41.5h37" style="stroke:#000;stro
  </svg>
</button>
```

Discussion

According to the WebAIM Million 2024 report, which is the result of a yearly automated accessibility evaluation of the top 1 million websites, 28.2% of tested websites contained empty buttons, which makes it one of the top five issues the study found. An *empty button* is either a button with no children or a button that contains unlabeled graphics and with no other source that labels it.

There are different methods for labeling buttons. Which one you choose depends on the requirements. When you have

multiple options, I recommend following the priority order Adrian Roselli describes in "My Priority of Methods for Labeling a Control":

1. Native HTML techniques
2. `aria-labelledby` pointing at existing visible text
3. Visibly hidden content
4. `aria-label`

If the button contains only text or a combination of text and icons, put the label as text between the element's start and end tag, as shown in Example 4-5. If there's an icon that doesn't provide additional information, remove it from the accessibility tree to avoid redundancy (see Example 4-10).

If there's no visible text but only an image or icon, then its alternative text can serve as the name for the button (see Examples 4-6 and 4-7). These kinds of images are called functional images. Their alternative text should describe not what they show, but their purpose. Alternatively, you can use visually hidden text (Example 4-8), `aria-labelledby`, or the `aria-label` attribute (Example 4-9).

Missing accessible names are probably the most common issue, but wrong ones can be problematic, too. Sometimes they're in

the wrong natural language, like English on a French site. Sometimes they contain unresolved variables or placeholder text. These issues usually arise with icon-only buttons. You can easily miss visually hidden text or labels provided via attributes like `aria-label` since they're visible only in the code, not the rendered UI. You should favor buttons with visible text because they are universally understandable and less error-prone.

The label should be informative and concise. Don't label a button "Click here to open or close the navigation." "Navigation" is sufficient.

4.3 Remove Default Button Styles

Problem

Even when a button doesn't *look* like a button, it must still meet most of the requirements described in Recipe 4.1. If it doesn't, it might not be accessible to keyboard and screen reader users. If you pick a generic element with fewer default styles instead of the `<button>` element, users might not be able to access the fake button or understand its purpose.

Solution

If you want to use a button but don't want to make it look like a button, you should still use the `<button>` element but remove the default button styles. CSS offers three efficient strategies for removing default button styles, as shown in Examples 4-11, 4-12, and 4-13.

Example 4-11. Removing or resetting properties manually

```
button {
  background: none;
  border: 0.1em solid transparent; ❶
  font: inherit;
  padding: 0;
}
```

❶ Shows an outline in forced-colors mode, which can be helpful.

Example 4-12. Resetting all button properties to their initial value

```
button {
  all: initial;
}

button:focus-visible {
  outline: 0.1em solid;
```

```
    outline-offset: 0.1em;
  }
```

Example 4-13. Resetting all button properties to their initial value except for inheritable properties

```
  button {
    all: unset;
  }

  button:focus-visible {
    outline: 0.1em solid;
    outline-offset: 0.1em;
  }
```

Discussion

Button elements in operating systems and web pages have a particular default shape and styling: a rectangle with a border, a background color, and some padding between the text and the border. When you style a page, you usually stick to these default characteristics. You change the default values and maybe add properties, as shown in Example 4-14. You might also have versions of that button that vary in size, color, and shape.

Example 4-14. Custom styles for buttons

```
button {
  --_l: 0.47;

  background: oklch(var(--_l) 0.05 195.6);
  color: #fff;
  font-size: 1.2rem;
  font-family: inherit;
  padding: 0.4em 0.9em;
  border-radius: 3px;
  border: 0;
  min-inline-size: 7rem;
}

button:is(:hover, :focus-visible) {
  --_l: 0.27; ❶
}
```

❶ The background color gets darker on `hover` and `focus-visible`.

Some buttons don't look like buttons because they consist of only an icon. How you implement such a button is crucial. One of the most common accessibility issues on most websites I audit is fake buttons: many developers assume that if a control doesn't *look* like a button, it doesn't have to *be* a `<button>` element. Their reasoning is: if there are no button styles in the

first place, you don't have to remove them. That often results in code like that in Example 4-15, which looks harmless but makes a considerable difference to accessibility.

Example 4-15. Bad practice: A fake `div` button

```
<div class="button" aria-label="Navigation">
  <svg width="24" height="24" aria-hidden="true":
    <path d="M3 18h18v-2H3v2zm0-5h18v-2H3v2zm0-7
  </svg>
</div>

<script>
const button = document.querySelector(".button")
button.addEventListener("click", (e) => {
  console.log("do something");
});
</script>
```

There are several issues with this "button:"

- It's not focusable via keyboard.
- Even if it was focusable, activation via Enter or Space wouldn't work.
- Screen readers don't announce it as a button.
- Some screen readers don't announce it at all.

- You shouldn't use `aria-label` on generic elements because it's invalid to name them.

This "button" not being focusable affects keyboard and screen reader users, but the latter can use the virtual cursor to access the fake button. If a click event listener is attached to the `div`, some screen readers will hint that you can click it. Nevertheless, if you look at Table 4-1, you can see that using a `div` or any other noninteractive element isn't reliable. If you have need to remove default button styles, sticking to the `<button>` element and resetting styles using CSS is the safest choice.

Table 4-1. Screen reader test: accessibility of a `div` button

Screen reader	Browser	Screen reader narration
NVDA	Firefox	Navigation
JAWS	Chrome	N/A
Narrator	Edge	N/A
TalkBack	Chrome	Navigation
VoiceOver macOS	Safari	Navigation, empty group
VoiceOver iOS	Safari	N/A

It wouldn't be viable to use a generic element instead of a button or put event listeners on noninteractive elements. You need an interactive element if the user can interact with a component.

The four lines of CSS you see in Example 4-11 are all you need to remove buttons' default user-agent styles—assuming you haven't added more rules, like we did in Example 4-14. In that case, you would have to reset those styles as well. That's the problem with this manual approach: you must keep track of changes to your custom default styles and adjust the reset styles accordingly, or else reset every possible property in advance.

The approach illustrated in Examples 4-12 and 4-13 is much more efficient. You can use the `all` property in CSS to reset all properties of an element, except `unicode-bidi`, `direction`, and CSS Custom Properties. Depending on your needs, you can set its value to `initial` or `unset` (see Figure 4-1).

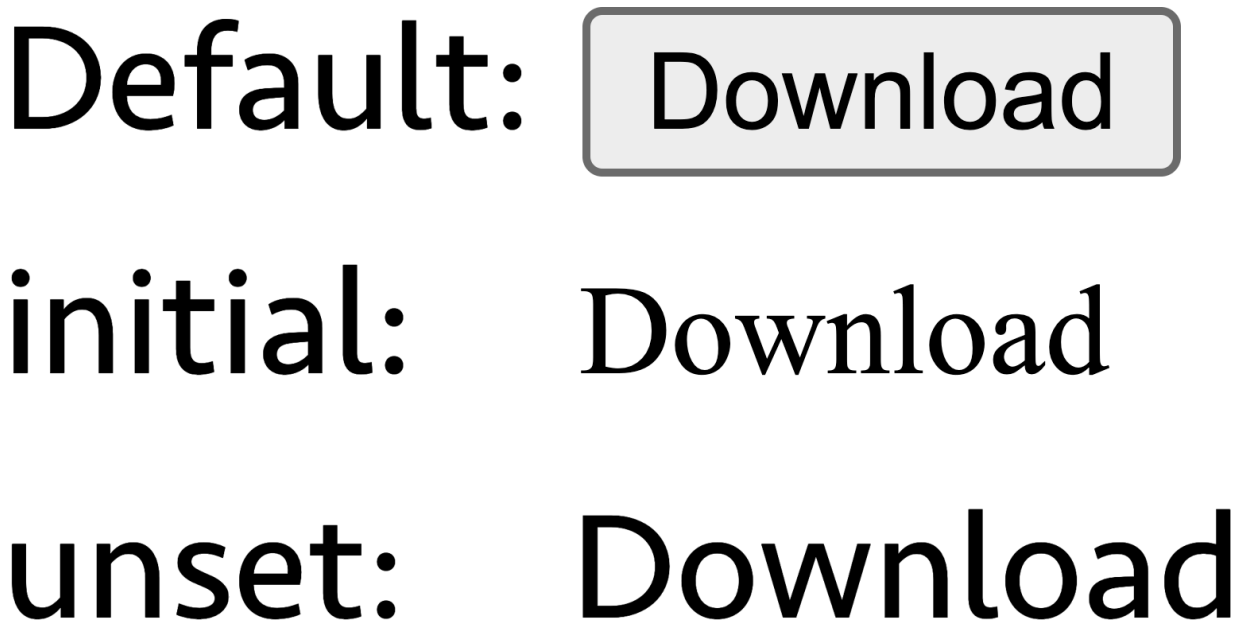


Figure 4-1. A button with default styling, a button with all properties set to `initial`, and a button with all properties set to `unset`

The `initial` keyword sets all property values to their initial value. Each property has an initial value, defined in the property's definition table. For example, if you look at the color property in the specification, the defined initial value in the definition table is `CanvasText`. Note that the initial value is not the default property value defined in the user agent.

The `unset` keyword resets a property to its inherited value, if the property naturally inherits from its parent and to its initial value if not. That can be useful if you want to keep specific properties like `font-family` or `color`.

Either way, you may have to bring back `focus` and `hover` styles, because `all` resets those, too. The gist is that even when a button doesn't look like a button, it must behave like one.

4.4 Add States and Properties

Problem

When screen reader users use buttons to control other elements on the page or settings for the site, those buttons must provide as much information as possible. That may include the type of button, the type of associated element, or the state of a button or controlled element. If that information is missing, it's much more complicated—or sometimes even impossible—to tell if the user has activated the button successfully and what happens when they do.

Solution

A button that toggles the visibility of another element needs to communicate whether the element is expanded, as shown in Example 4-16.

Example 4-16. The `aria-expanded` attribute on the button communicating whether the navigation is visible

```
<nav>
  <button aria-expanded="false" aria-controls="ma
    Navigation
  </button>

  <ul id="main_nav">…</ul>
</nav>

<style>
  [aria-expanded="false"] + ul { ❷
    display: none;
  }
</style>

<script>
const button = document.querySelector("button");

button.addEventListener("click", (e) => {  ❸
  const isExpanded = button.getAttribute("aria-ex
  button.setAttribute("aria-expanded", !isExpand
});
</script>
```

❶ `aria-expanded` must be on the button and not the expandable element.

❷ The list is hidden, depending on the value of the attribute.

❸ Click event on the button toggles the value of the `aria-expanded` attribute.

A button that turns a setting on or off must communicate whether it's active, as shown in Example 4-17.

Example 4-17. A button communicating its pressed state

```
<button type="button" aria-pressed="true"> ❶
  Add to favourites
</button>

<script>
const button = document.querySelector("button");

button.addEventListener("click", (e) => {
  const isPressed = button.getAttribute("aria-pr
  button.setAttribute("aria-pressed", !isPressed
});
</script>
```

❶ The button indicates that something was added as a favorite.

A button that toggles a setting must communicate whether it's active. Figure 4-2 and Example 4-18 show a switch button

communicating its state using the `aria-checked` attribute.

Figure 4-2. A toggle switch

Example 4-18. A switch button

```
<button role="switch" aria-checked="false">Funct

<script>
const button = document.querySelector("button");

button.addEventListener("click", (e) => {
  const isChecked = button.getAttribute("aria-ch
  button.setAttribute("aria-checked", !isChecked
});
</script>

<style>
  button {
    --toggle-offset: 0.125em;
    --toggle-height: 1.6em;
    --toggle-background: oklab(0.82 0 0);
```

```
    all: unset; ❶
    align-items: center;
    display: flex;
    gap: 0.5em;
    position: relative;
  }

  button::before { ❷
    background: var(--toggle-background);
    border-radius: 4em;
    content: "";
    display: inline-block;
    height: var(--toggle-height);
    transition: background 0.3s, box-shadow 0.3s
    width: 3em;
  }

  button::after { ❸
    --_size: calc(var(--toggle-height) - (var(--

    background: #FFF;
    border-radius: 50%;
    content: "";
    height: var(--_size);
    left: var(--toggle-offset);
    position: absolute;

    transition: translate 0.3s;
    top: var(--toggle-offset);
```

```
    width: var(--_size);
  }

  button:focus-visible::before { ❹
    outline: 2px solid;
    outline-offset: 2px;
  }

  button:is(:focus-visible, :hover)::before {  ❺
    box-shadow: 0px 0px 3px 1px rgb(0 0 0 / 0.3)
  }

  [aria-checked="true"] { ❻
    --toggle-background: oklab(0.7 -0.18 0.17);
  }

  button[aria-checked="true"]::after {  ❼
    translate: 100% 0;
  }
</style>
```

❶ Resets the default button styles.

❷ Pseudoelement for the background of the switch.

❸ Pseudoelement for the movable indicator of the switch.

❹ Adds custom focus styles.

❺ Shows a box shadow on `:hover` and `:focus-visible`.

❻ Turns the background color of the switch to green when active.

❼ Moves the indicator to the end of the switch when active.

A button can communicate its state and what kind of element it controls, as shown in Example 4-19.

Example 4-19. A button controlling a menu

```
<button
  type="button"
  aria-haspopup="menu" ❶
  aria-expanded="false" ❷
  id="button_settings"
>
  Settings
</button>

<ul role="menu" aria-labelledby="button_settings
  <li role="none">
    <button role="menuitem">Print</button>

  </li>
  <li role="none">
```

```
    <button role="menuitem">Save</button>
  </li>
</ul>

<style>
  [aria-expanded="true"] + ul { ❹
    display: block;
  }
</style>

<script>
const button = document.querySelector("button");

button.addEventListener("click", (e) => { ❺
  const isExpanded = button.getAttribute("aria-e
  button.setAttribute("aria-expanded", !isExpand
});
</script>
```

❶ Indicates that the button controls a menu.

❷ Indicates that the controlled menu is collapsed.

❸ The menu is hidden by default.

❹ The list is hidden, depending on the value of the attribute.

6. Click event on the button toggles the value of the `aria-expanded` attribute.

Discussion

Many attributes in the ARIA specification provide elements with states or properties. When you build custom JavaScript widgets, you'll use some of them often. The `arial-label` property, for example, gives an element an accessible name, while the `aria-hidden` state removes an element from the accessibility tree.

This recipe focuses on four attributes commonly used with buttons.

The expanded state

You use the `aria-expanded` attribute on a button element to indicate whether a grouping element it controls is expanded or collapsed.

NOTE

The controlled element can be pretty much any element, but it's usually a grouping element like `<div>`, `<p>`, or `<ul>`.

In Example 4-16, you can see that the button has the attribute and communicates that the associated element is expanded. Users should understand from the button text which element the button controls. That's why you should avoid generic text like "show/hide." The attribute can be useful for fly-in navigations (see Recipe 7.5), for submenus in nested navigations (see Recipe 7.7), and for disclosure widgets (see Recipe 8.3).

The button's job is to communicate whether the controlled element is expanded. You must follow three essential rules when you apply it:

- You set the attribute on the element that does the controlling (the button), not the controlled element (the group).
- The sheer presence of the attribute is not enough; you must set it to "true" or "false."
- The attribute must be present and set before the user interacts with the button. If you set the value to "false," it means that the controlled element is collapsed. If you don't set the attribute, the button doesn't control anything.

As you can see in Example 4-16, the button has another ARIA attribute: `aria-controls`.

The controls property

The `aria-controls` attribute identifies the element the button controls. The *value* is a list of one or more `id` references. With the attribute present, a screen reader can identify a relationship between a button and another element. JAWS doesn't automatically announce this relationship, but you can use the `JAWSKEY` + `ALT` + `M` shortcut to jump directly to the controlled element.

For disclosure widgets, this attribute isn't well supported (JAWS is the only screen reader that uses it), and it's unclear how much importance screen reader vendors plan to attach to it. However, it does no harm. For NVDA, this has been an open issue since 2018; an open discussion in the ARIA GitHub repository dates to 2019. Whether you use it is up to you. Until there's an official recommendation for or against it, I'm using it in the following chapters.

Pressed state

The `aria-pressed` attribute indicates the current "pressed" state of toggle buttons.

A *toggle button* is similar to a checkbox but not quite the same. Aside from the styling, the most significant difference is that a

checkbox conveys only a state (checked/unchecked/mixed), whereas a button performs an action. When users click a toggle button, they expect something to happen. Pressing the button in Example 4-17 toggles the value of the `aria-pressed` attribute and changes the state of the *Add to favorites* button. Client-side changes like that require you to work in an environment that relies on JavaScript because the pressed state must change on click. If that's not the case and you want to enhance the control progressively, use a checkbox instead. Adrian Roselli describes the differences between checkboxes and toggle buttons in depth in "Under-Engineered Toggles" and "Under-Engineered Toggles Too".

Checked state

The `aria-checked` attribute indicates the current "checked" state of checkboxes, radio buttons, and other widgets.

You're not supposed to use `aria-checked` on a button with the role `button`, but you can use it on a switch (see Example 4-18). A *switch* is a type of checkbox that represents on/off values instead of checked/unchecked/mixed values. It provides approximately the same functionality as a checkbox or toggle button, but you can distinguish between them for screen readers in a fashion consistent with its on-screen appearance.

Switches are a problematic pattern in terms of user experience. I talk a bit about why in Recipe 9.1. If you decide to use switches, test them thoroughly with users, including those who use screen readers.

haspopup property

The `aria-haspopup` attribute indicates that a button controls another interactive pop-up element. Most screen readers also announce the type of pop-up element. The attribute supports seven values: *true*, *false*, *menu*, *dialog*, *grid*, *listbox*, and *tree*, indicating that the referenced element has the respective role. *true* is the same as *menu*.

Depending on the screen reader software you're using, if you focus the button in Example 4-19, it will announce something like "Settings, button, menu" (JAWS) or "Settings, pop-up button, menu pop-up" (VoiceOver). The value you use must keep its promise: if the value is *menu*, the role of the controlled element should also be *menu* and function accordingly. JAWS, for example, will also announce appropriate instructions when an attribute with a specific value is present. For *true* and *menu*, it announces, "Press Space to activate the menu. Then navigate with arrow keys." For *listbox*, *tree*, and *grid*: "To activate, press Enter."

The values *true* and *menu* are well supported in all screen readers. However, TalkBack and Narrator don't support *grid*, *dialog*, *listbox*, or *tree*.

4.5 Don't Disable Buttons

Problem

Disabling buttons can cause more problems for users than benefits.

Missing feedback

When you click a disabled button, nothing happens. The button doesn't explain what's wrong or help you fix the problem. It provides no helpful feedback. If the user thinks their answers are correct, not providing feedback can make the UI feel broken.

Missing focus

Disabled buttons are not focusable, so screen reader users who use the `Tab` key to navigate might not know that there even is a button. If the button's styling isn't obvious, it may confuse keyboard users trying to focus the button.

Low contrast

WCAG's minimum contrast rule doesn't apply to disabled controls, but they're often hard to read, especially for people with low vision.

Deception

It's not always apparent that buttons are disabled. Some users will try to click them; if nothing happens, they can feel irritated, confused, or disappointed. That may occur because the design isn't distinct or because disabled buttons usually contain call-to-action words like "submit," "send," or "order," which lure users into clicking them.

Solution

Don't disable buttons. Users should always be able to interact with them and get feedback.

Discussion

The point of disabling buttons is to avoid premature clicks and to make it difficult for users to make mistakes when filling out forms. Developers use this technique to indicate that something

important is wrong or missing, and must be fixed before the user can continue to the next step. That sounds good, but a disabled button is not the best solution. It's a flawed pattern. A button can be disabled for many reasons, but using it forces the user to figure out what went wrong.

Instead of disabling buttons, there are several measures you can take to avoid errors up front:

- Use clear labels for your input fields.
- Add hints and descriptions when the label alone isn't clear enough.
- Split complex forms into multiple steps or pages to reduce cognitive load.
- Always enable buttons and validate input on submit.
- Give clear error messages.

When the user presses the button, provide them with a list of errors that point to the respective field, or move focus to the erroneous field if there's only one (see Recipe 9.4 for details).

See Also

- "Buttons and the Baader–Meinhof phenomenon" by Manuel Matuzović

- “hasPopup hasPoop” by Steve Faulkner
- “aria-hasPopUp less is more” by Steve Faulkner
- “aria-haspopup and screen readers” by Manuel Matuzović
- “The problem with disabled buttons and what to do instead” by Adam Silver
- “Usability Pitfalls of Disabled Buttons, and How To Avoid Them” by Vitaly Friedman
- “Making Disabled Buttons More Inclusive” by Sandrina Pereira
- “Disabled buttons suck” by Hampus Sethfors
- “Perceived affordances and the functionality mismatch” by Léonie Watson
- “Toggles suck!” by Joel Holmberg

Chapter 5. Styling Content

Respecting users and user preferences is one of the most critical aspects of designing and building inclusive UIs. Operating systems and browsers provide users with different options for tweaking UIs and behavior according to their needs. For example, they can change the default font size or reduce motion in UIs. You can query those settings and adapt your CSS accordingly to provide them with an appropriate experience on your website.

Another aspect of inclusivity in CSS is understanding how it influences semantics and operability. CSS is a stylesheet language primarily intended for changing the presentation of HTML elements. Still, some of its properties also affect the semantic meaning of HTML and how it functions.

5.1 Work with Color

Problem

Color is an integral part of web design and user experience. You can use it for styling and decorating or for conveying emphasis, state, and importance. However, not everyone perceives color

the same way. If you rely on color alone to communicate, or use colors with low contrast, you may exclude people, especially those with low vision.

Solution

First, use tools to test color contrast. Ensure that text has a high contrast ratio against its background and the background colors of components against adjacent colors (see Figure 5-1).

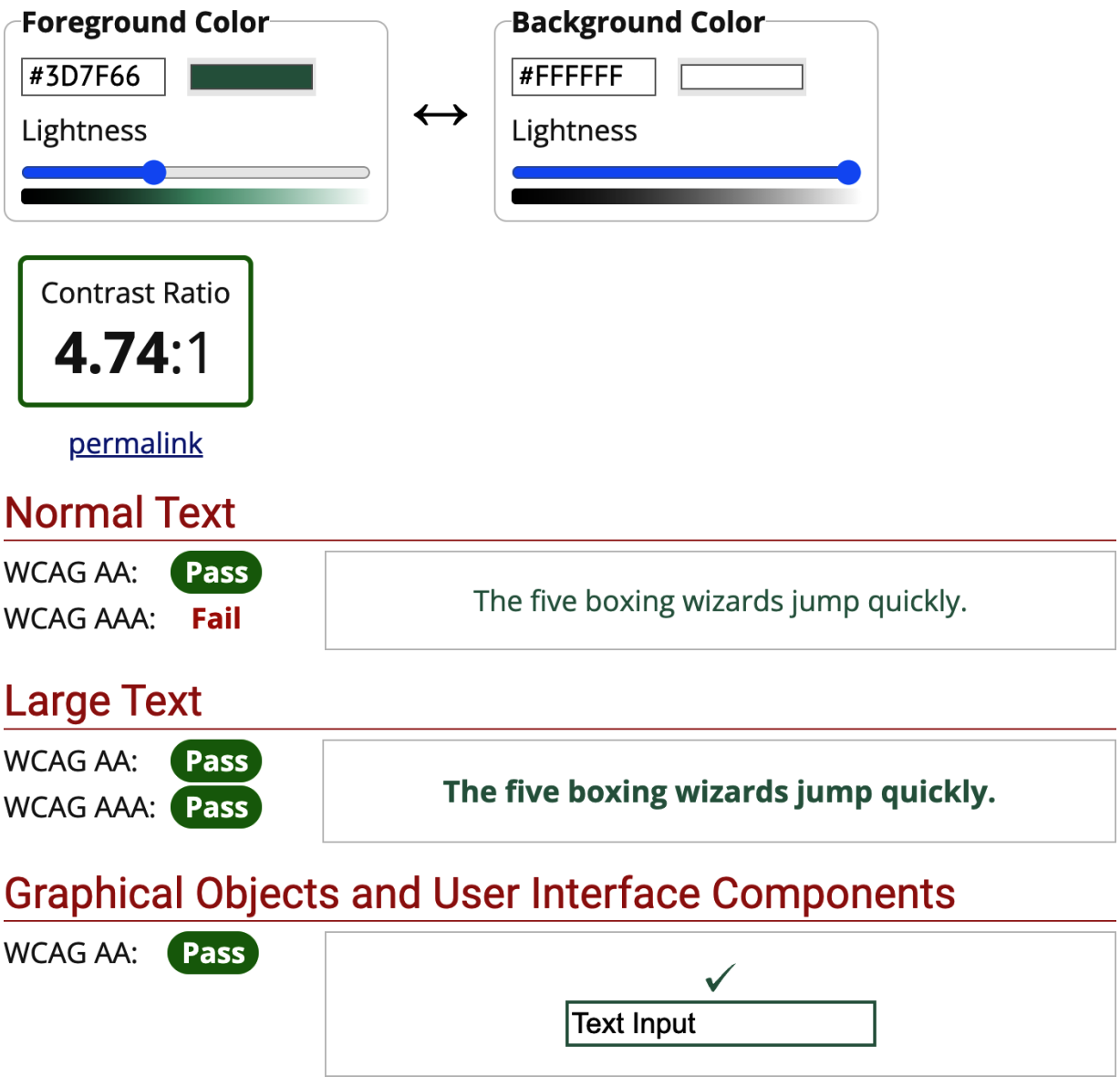


Figure 5-1. The WebAIM color contrast checker showing a ratio of 4.74:1 for the given color combination

Second, don't use color alone to indicate an action, prompt a response, visualize a change of state, or distinguish a visual element (see Example 5-1).

Example 5-1. An error message, an *x* icon, the color red, and the `aria-invalid` attribute, all conveying that something

has gone wrong

```
<label for="email">Your email address</label>
<input type="email" id="email" required aria-inv
       aria-describedby="error">
<div id="error" class="error" style="color: #D52
  <svg viewBox="0 0 640 640" width="16" fill="cu
    <path d="M640 128 512 0 320 192 128 0 0 128l
    512l128 128 192-192 192 192 128-128-192-192
  </svg>

  Please enter a valid e-mail address
</div>
```

Discussion

Many of the solutions in this book target optimizing your frontend components for usage with a screen reader. While most screen reader users are blind, not all are; and not everyone with a vision impairment uses or needs a screen reader. According to the World Health Organization (WHO), 39 million people worldwide are blind and 246 million have low vision.

Definitions of *low vision* usually include only impairments that aren't correctable with regular glasses, contact lenses, medicine, or surgery. That includes problems with visual acuity, light sensitivity, contrast sensitivity, a smaller or obscured field of vision, or color deficiencies. Eye diseases and health conditions such as cataracts, glaucoma, and diabetes are often the cause of low vision. Sometimes, it's congenital or caused by an injury. In addition, other factors, like old or limited hardware or bright sunshine, can affect vision and perception.

Low vision is manifold, but contrast sensitivity and color vision deficiency are two types of low vision that impact web use, so the next few sections cover those use cases.

Color contrast

You can make text more readable for people with moderately low vision by providing a minimum luminance contrast ratio between it and its background. To calculate this ratio, the W3C created a formula. Contrast ratios can range from 1 to 21 (commonly written 1:1 [lowest] to 21:1 [highest]. You get 1:1 when you use the same color for the background and text. You get 21:1 when you use black text on a white background and vice versa.

The WCAG's minimum requirements for color contrast are:

- Regular text must have a ratio of 4.5:1.
- Regular text at 24px and larger must have a ratio of 3:1.
- Bold text at 19px and larger must have a ratio of 3:1.
- Logos, brand names, and purely decorative elements don't underlie any contrast requirements.

There are many tools available for testing color contrast. Figure 5-1 shows the Contrast Checker by WebAIM. There are also tools built into browsers. DevTools in Google Chrome shows the contrast ratio in a tooltip when you click the little square next to the value of a color declaration in the styles pane, as shown in Figure 5-2.

Please note that those are only the *minimum* requirements; even if you meet them, they don't guarantee high contrast. The color-contrast ratio formula receives a lot of criticism because it doesn't reflect modern technological requirements and scientific research. It was created in the mid-2000s when the hardware and software landscape was fundamentally different.

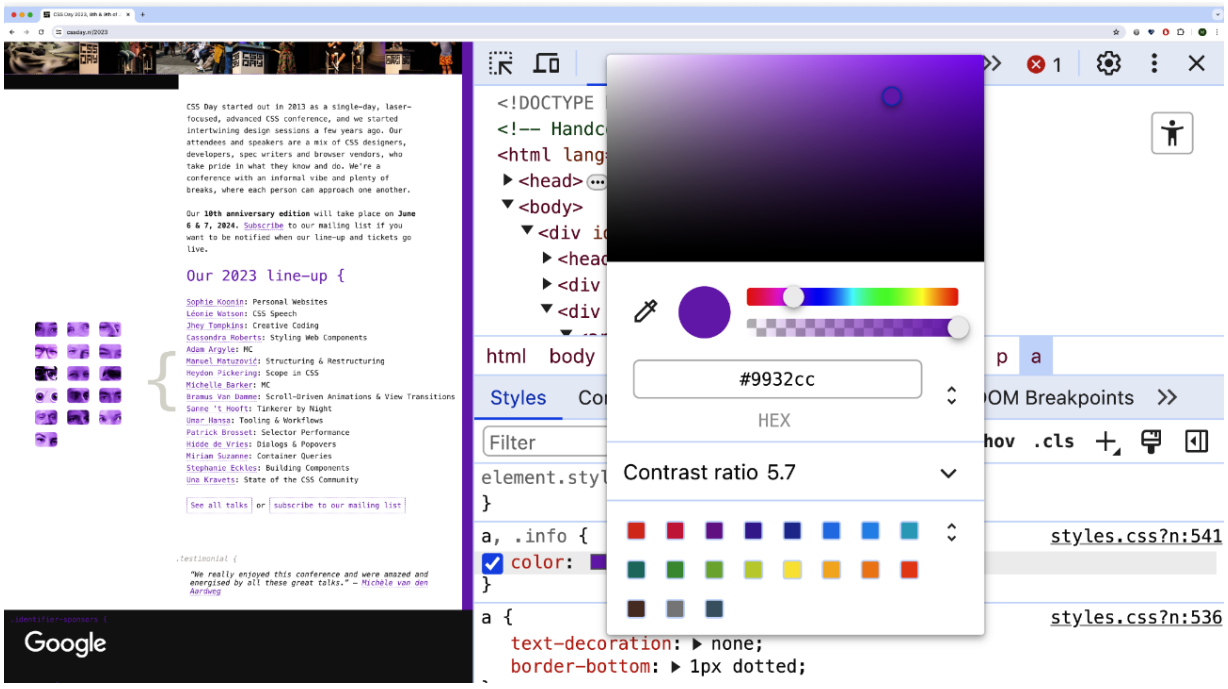


Figure 5-2. DevTools shows a ratio of 5.7:1 for the selected color combination

The biggest points of criticism of the formula are:

- It doesn't take human perception into account.
- It doesn't consider the font's characteristics, like size and weight.
- It treats front and background colors the same. Inverting the color doesn't change the calculation, but it should, because the surrounding color affects perception.

The draft of WCAG 3.0 mentions a different algorithm, APCA (Accessible Perceptual Contrast Algorithm), which may become the new way of testing contrast ratios. It's based on modern research and premises. However, WCAG 3.0 is not ready yet,

and according to accessibility expert Eric Eggert, there are also some open questions regarding APCA. It's probably not coming any time soon, but if you want to test it today, activate it in Chrome/Edge DevTools.

For WCAG compliance, you still have to follow the current formula. Beware that it has flaws, so don't rely on the ratio alone; use your own judgment and test with users.

Color vision

Approximately 1 in 12 men (8%) and 1 in 200 women (0.5%) have color vision deficiencies. They don't see certain colors well or, in rare cases, don't see any color at all. Since many people perceive color differently, it's critical not to use color alone to convey information.

Recipe 3.2 discusses link styling and explains why underlines for links in running text are crucial. In Figure 5-3, you can see that it's hard to distinguish links from regular text on wikipedia.org because they rely on color alone. If the links had additional underlines, people who can't perceive or distinguish their color could still recognize them.

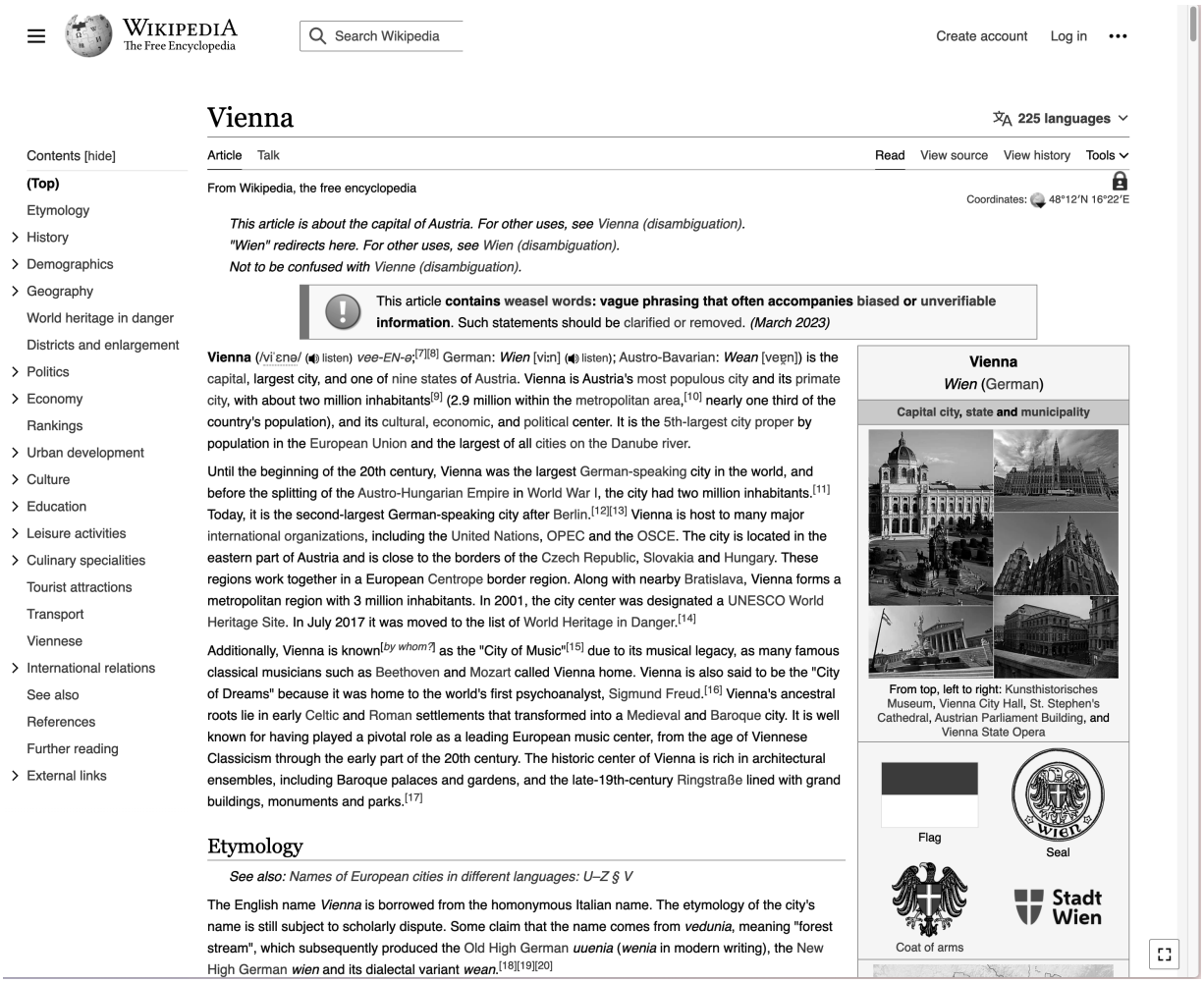


Figure 5-3. On a Wikipedia page presented without colors, links are hard to distinguish from regular text

In Figure 5-4, you can see how *caniuse.com* combines color with different background patterns to indicate support for features in different browser versions.

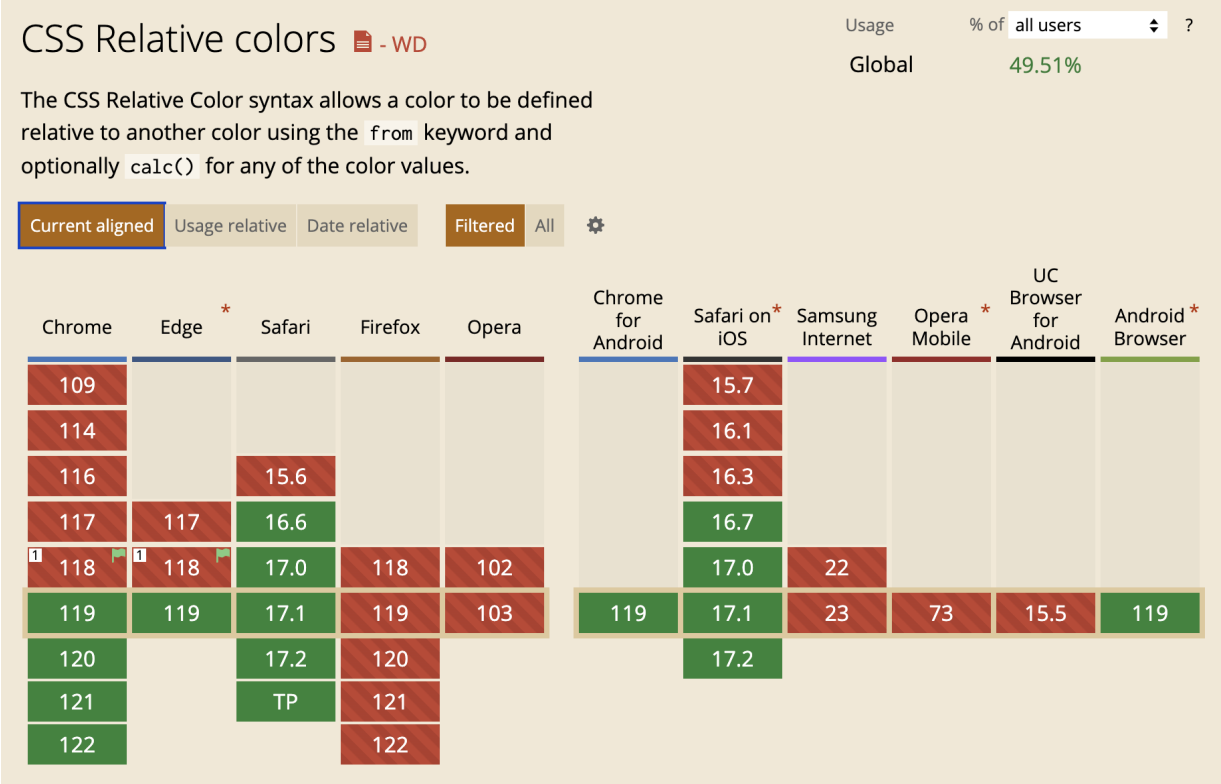


Figure 5-4. A browser-support chart on caniuse.com. Green (solid background color) indicates support; red (striped background) indicates lack of support

Combining color, text, and icons can also work well, as shown in Example 5-1 and Figure 5-5.

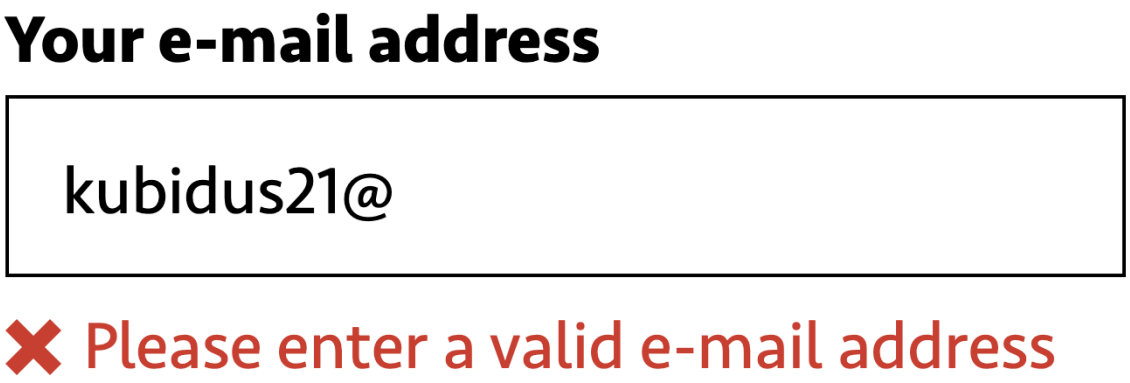


Figure 5-5. The *x* icon supports the error message and the color in indicating an error

Color on the web is a complex topic. It's impossible to make it right for everyone, but you should at least fulfill these two basic requirements: use colors with good contrast and don't use color alone to communicate information.

5.2 Respect User Preferences

Problem

Users can configure accessibility-related settings in their operating systems and browsers. When you don't respond to those settings in your code, you're disrespecting their choices. The effects on users range from limiting access to your content making it more challenging to simply making it impossible for them to use the website.

Solution

Use media queries to retrieve browser and operating system settings. There are several preferences you can detect.

- Dark mode (Example 5-2)
- Increased contrast (Example 5-3)
- Forced colors (high-contrast themes) (Example 5-4)

- Inverted colors (Example 5-5)
- JavaScript support (Example 5-6)
- Reduced transparency (Example 5-7)
- Reduced motion (Recipe 5.5)

Example 5-2. Swapping the background and text color when dark mode is active

```
html {
  --dark: oklch(37.34% 0.081 236.96);
  --light: oklch(98.89% 0.005 17.25);
  --background: var(--light); ❶
  --text: var(--dark);
}

@media(prefers-color-scheme: dark) {  ❷
  html {
    --background: var(--dark);
    --text: var(--light);
  }
}

body {
  background-color: var(--background);
  color: var(--text);
}

svg {
```

```
    fill: currentColor;  ❸
  }
```

❶ By default, the background color is light and the text dark.

❷ If users prefer a dark color scheme, colors swap.

❸ The `currentColor` is always relative to its own or its parent's text color. It's automatically either light or dark.

Example 5-3. Detecting whether users prefer increased contrast, and increasing contrast accordingly

```
html {
  --blue: oklch(0.56 0.13 237.77);
  --text: var(--blue);
}

@media (prefers-contrast: more) {
  html {
    --text: oklch(from var(--blue) calc(l - 16)
  }
}

body {
  color: var(--text);
}
```

Example 5-4. Detecting forced-colors mode

```
@media (forced-colors: active) {
  /* custom styles */
}
```

Example 5-5. Detecting whether colors are inverted

```
@media (inverted-colors: inverted) {
  :is(img, video) {
    filter: invert(100%);
  }
}
```

Example 5-6. Only hiding content in CSS, when JavaScript is enabled

```
@media (scripting: enabled) {
  .disclosure-content {
    display: none;
  }
}
```

Example 5-7. Removing transparency when users prefer reduced transparency

```
dialog {
  --transparency: 0.6;

  background-color: rgb(4 227 215 / var(--transpa
  backdrop-filter: blur(5px);
}

@media(prefers-reduced-transparency: reduce) {
  dialog {
    --transparency: 1;
  }
}
```

Discussion

The progressive enhancement principle in web design describes a strategy that focuses on content. It's at the very core of the user experience. It should be accessible regardless of the user's hardware or software, security restrictions, or other system or browser settings. Things like styling or nonessential interactivity are optional layers stacked on top of the content. When one of those layers isn't accessible to some users, the site doesn't break entirely; it falls back to the previous layer. This paradigm enables you to build highly accessible and flexible web experiences.

In “Understanding Progressive Enhancement”, Aaron Gustafson explains that content is the reason we create websites to begin with. It’s what’s most important to users. That’s why you should make access to the core of your websites as frictionless as possible. That means shipping text and HTML with as few dependencies as possible. On top of your HTML you have multiple layers of CSS, and on top of that, multiple layers of JavaScript. Users get more or fewer layers depending on their operating system, browser, and settings for each. That works well because HTML and CSS are designed to be error-tolerant.

When you hit the wall with that tolerance, feature detection in CSS and JavaScript is another excellent way to control your layers (see Examples 5-8 and 5-9 for examples).

Example 5-8. Feature detection in CSS: The rule applies only if the browser supports the subgrid feature

```
@supports (grid-template-columns: subgrid) {
  .grid {
    display: grid;
    grid-template-columns: subgrid;
  }
}
```

Example 5-9. Feature detection in JavaScript: The function call runs only when the browser supports geolocation

```
if ("geolocation" in navigator) {
  navigator.geolocation.getCurrentPosition(funct
    // show the location on a map
  });
}
```

Feature detection lets you query which features the user's browser supports and adapt the interface to the technical requirements of your user's software and hardware. To build interfaces that also respect personal needs and preferences, you can use media features in CSS and JavaScript.

Color schemes, also known as dark mode

In Recipe 5.1, you learned that there are several types of low vision. Many people with low vision have *photophobia*, an extreme sensitivity to light. Bright light from a screen makes it difficult or impossible to read and can even cause pain for some. To counteract this, users dim their screen, use a screen overlay, or switch the theme of their operating system to one that uses dark colors. The `prefers-color-scheme` media feature enables you to react to those settings and serve your

website in darker colors, as shown in Example 5-2. In Figure 5-6, you can see how developer Max Böck respects user preferences on his website.

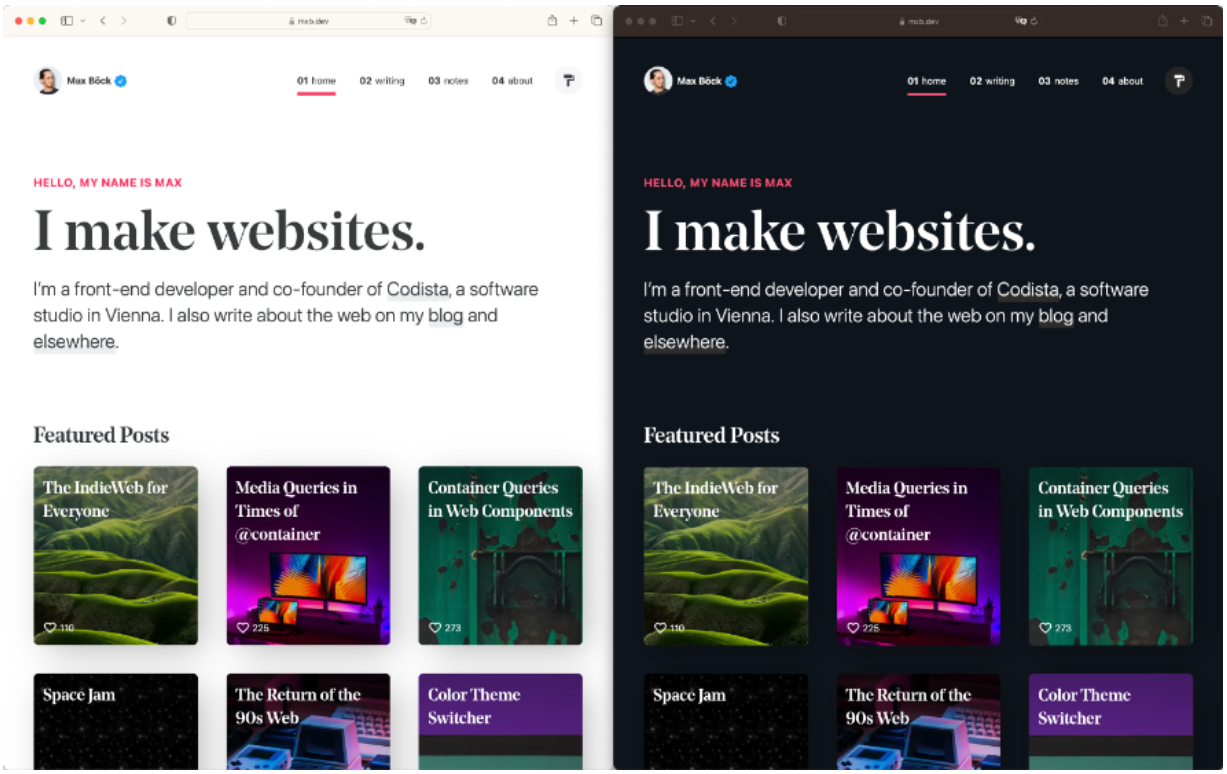


Figure 5-6. Comparison of mxb.dev in light and dark mode

In Recipe 13.5, you can learn how to emulate color-scheme settings in DevTools.

More contrast

Users who prefer more contrast can enable high-contrast colors in their operating systems. As shown in Example 5-3, you can

query that setting using the `prefers-contrast` media feature and increase the contrast of your colors.

Forced colors

Windows users can switch to light or dark themes for the entire operating system (forced-colors mode), as shown in Figure 5-7. They can even manually pick colors for backgrounds, text, links, buttons, and selected or inactive text. You can use the `forced-colors` feature query to detect whether forced-colors mode is active, as shown in Example 5-4. You don't have access to the colors the user has picked, which means that it can be a very dark or a very light theme or something in between. The colors the user will see are unpredictable for you. That's why you shouldn't use this media feature to tweak colors. Instead, use it for things like assigning the appropriate color keyword to a control built without using native HTML controls. Sarah Higley shares useful optimization tips in her blog post "Quick Tips for High Contrast Mode", and Adrian Roselli goes into detail in "WHCM and System Colors".

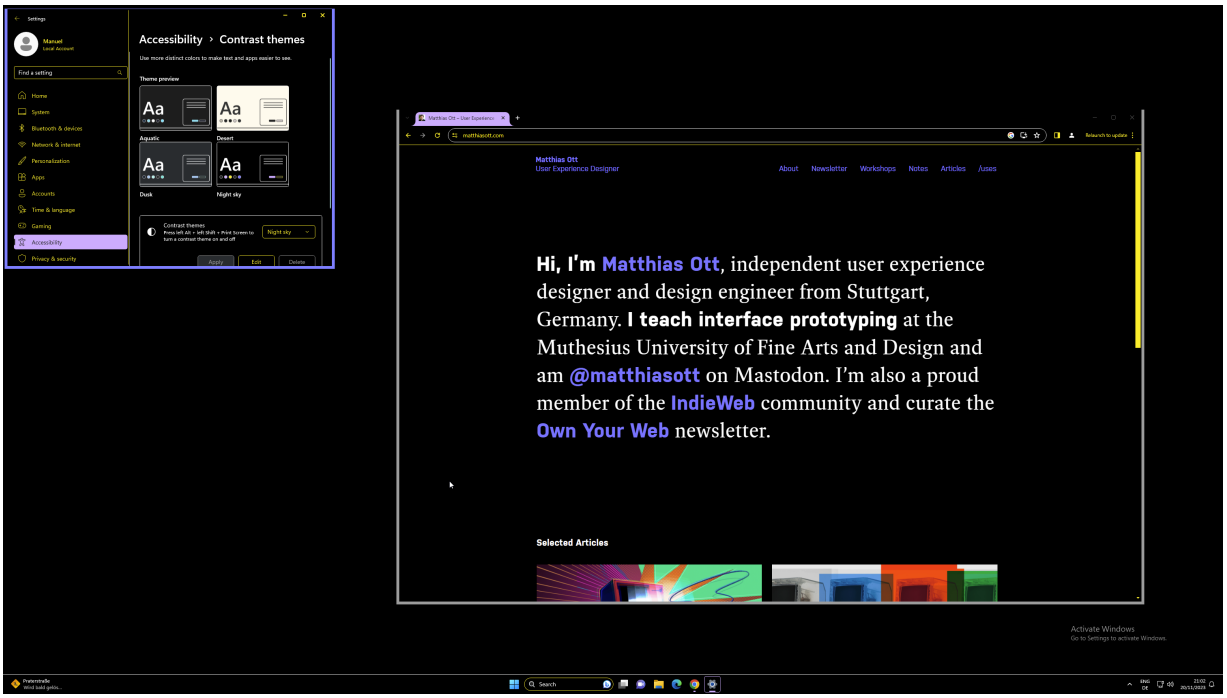


Figure 5-7. Windows in a high-contrast theme

Test your websites in forced-colors mode (currently only possible on Windows for the entire operating system, or browser settings or developer tools [see Recipe 13.5]), and tweak the styling of any element that becomes unrecognizable. Ideally, you should have little to do, because most parts of the UI usually work well by default.

Please keep in mind that optimizing for forced-colors mode is about visibility, not beauty. Don't try to *fix* something that doesn't need fixing just because it doesn't look pretty.

Inverted colors

Inverting operating system colors is another option for people with photosensitive conditions. Inverted colors mode is a good option for websites that don't provide a dark mode because, when activated, it reverses the color of every pixel on the screen. You can query that setting in Safari on macOS and iOS using the `inverted-colors` media feature.

In Figure 5-8, you can see how inverted colors mode doesn't just affect background and text colors, but images as well. As with forced colors, you probably don't want to make color adjustments in this query, only improve the display of specific elements like images, as shown in Example 5-5.

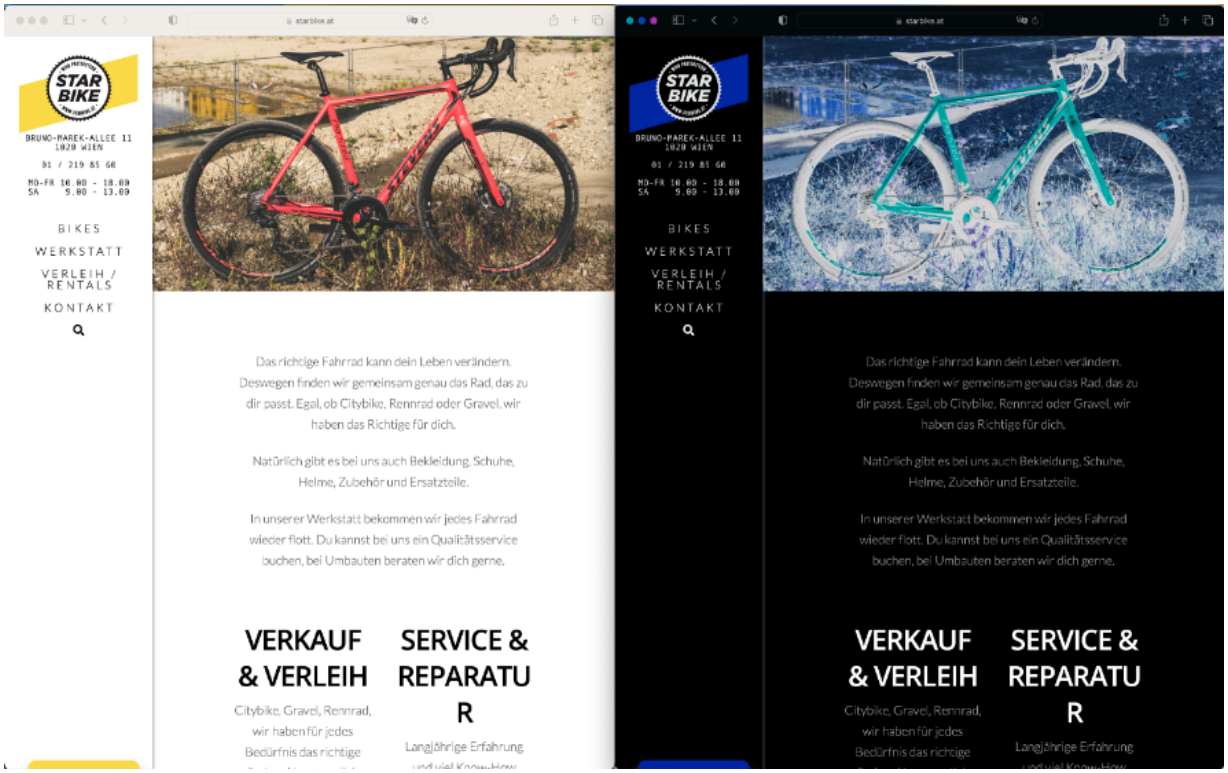


Figure 5-8. Comparison of starbike.at in regular and inverted-colors mode

JavaScript

Some people disable JavaScript for performance or security reasons. You can detect whether scripting languages are supported, as shown in Example 5-6, and adapt your interfaces accordingly.

Transparency

Design trends like Glassmorphism use translucent backgrounds to create a specific visual effect, resulting in underlying background colors or elements that shimmer through the background of the overlying element. That may be visually appealing, but it can distract some people and impair legibility.

Operating systems like macOS and Windows offer options to reduce transparency in the operating system. In CSS, you can query that setting using the `prefers-reduced-transparency` media feature, as shown in Example 5-7 and Figure 5-9.

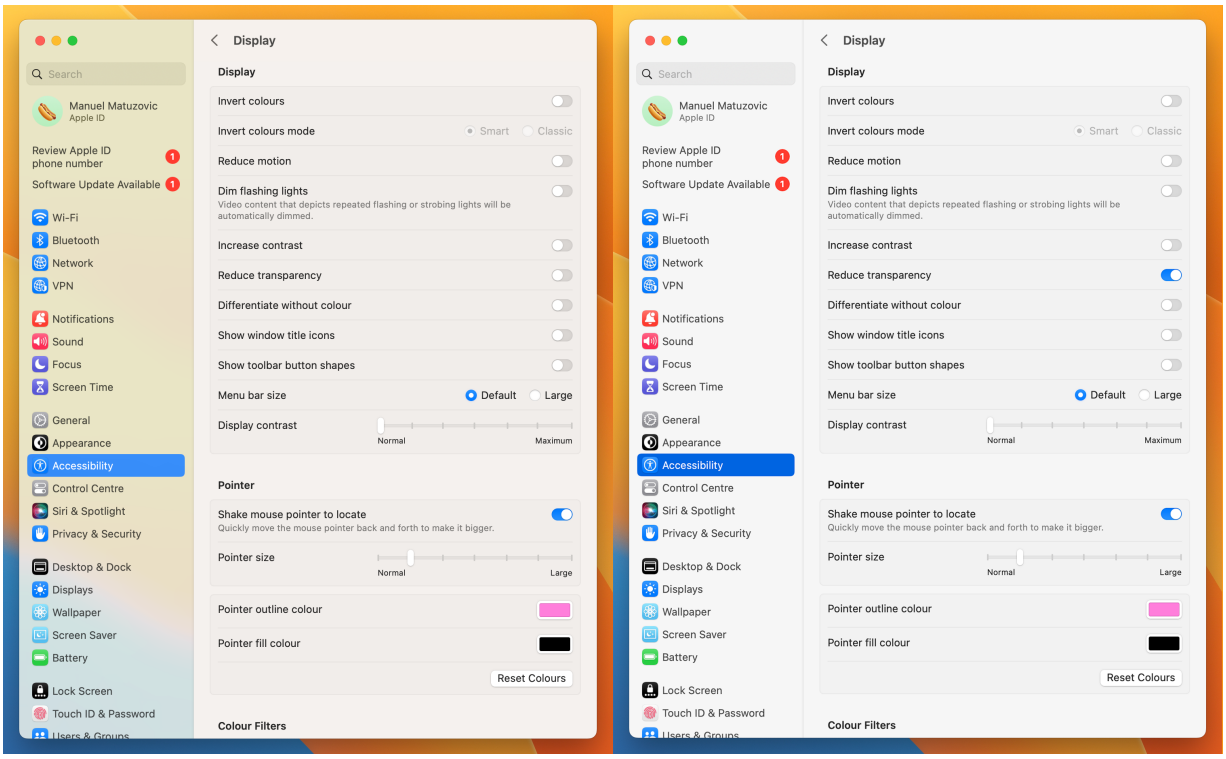


Figure 5-9. The settings panel in macOS without (left) and with (right) reduced transparency active

5.3 Work with Units and Sizes

Problem

Users can adjust the base and minimum font sizes in their browsers according to their needs. When you use absolute units for font sizing and general sizing and spacing in CSS, you risk overriding their preference, potentially forcing a smaller font size and interface than they feel comfortable reading and interacting with.

Solution

Use relative units for most of your sizing. Examples 5-10 through 5-14 illustrate using relative units in CSS.

Example 5-10. A component using `rem`, a unit relative to the font size of the root element

```
.a-component {
  font-size: 1.125rem;
  line-height: 1.5;
  max-width: 28.125rem;
}
```

Example 5-11. The maximum width of the paragraph is relative to the width of the glyph 0; in this font and with this font size, approximately 65 characters fit into one line

```
p {
  max-width: 65ch;
}
```

Example 5-12. The column takes up 50% of its parent's width. The dialog fits the dynamic height of the viewport.

```
.grid {
  max-width: 60rem;
}

.column {
  width: 50%;
}

dialog {
  height: 100dvh;
}
```

Example 5-13. The padding is relative to the font size of the button and increases and decreases with the font size

```
button {
  --btn-font-size: 1.2rem;

  border: 1px solid; ❶
  border-radius: 4px;
  background: #123456;
  color: #ffffff;
  font-family: inherit;
  font-size: var(--btn-font-size);
  padding: 0.4em 0.8em; ❷
}
```

```
.btn--small {
  --btn-font-size: 1rem;
}

.btn--large {
  --btn-font-size: 1.6rem;
}
```

- ❶ Fixed value for the border and border-radius. Use a relative unit if you want the border to grow with the font size.

- ❷ The padding is relative to the button's font size.

Example 5-14. The media query is relative to the base font size in the browser

```
@media (min-width: 60em) {
  .wrapper {
    display: flex;
  }
}
```

Discussion

One of the most important aspects of creating UIs that respect user preference is allowing users to adjust font sizes according to their needs—and building layouts that respond to those settings.

The base font size in most browsers for regular text is `medium`, which usually equals `16px`. You can change that value by selecting the `<html>` element and defining another pixel value, as shown in Example 5-15.

Example 5-15. Bad practice: Overriding the default font size

```
html {
  font-size: 14px;
}
```

Now, the base font size in the browser is still `16px`, but you've decided to ignore it and use another value instead. That's problematic, because the browser's base font size is not a static value. In some browsers, users can change it in the settings. They usually do that to increase it when the text is too small to read. That's why using an absolute unit like `px` (pixels) for text-related properties and properties that base their dimensions on font size is a bad practice.

In Figure 5-10, you can see how the custom font-size setting "Very large" affects the font size in the settings itself, but the demo text remains at `14px`.

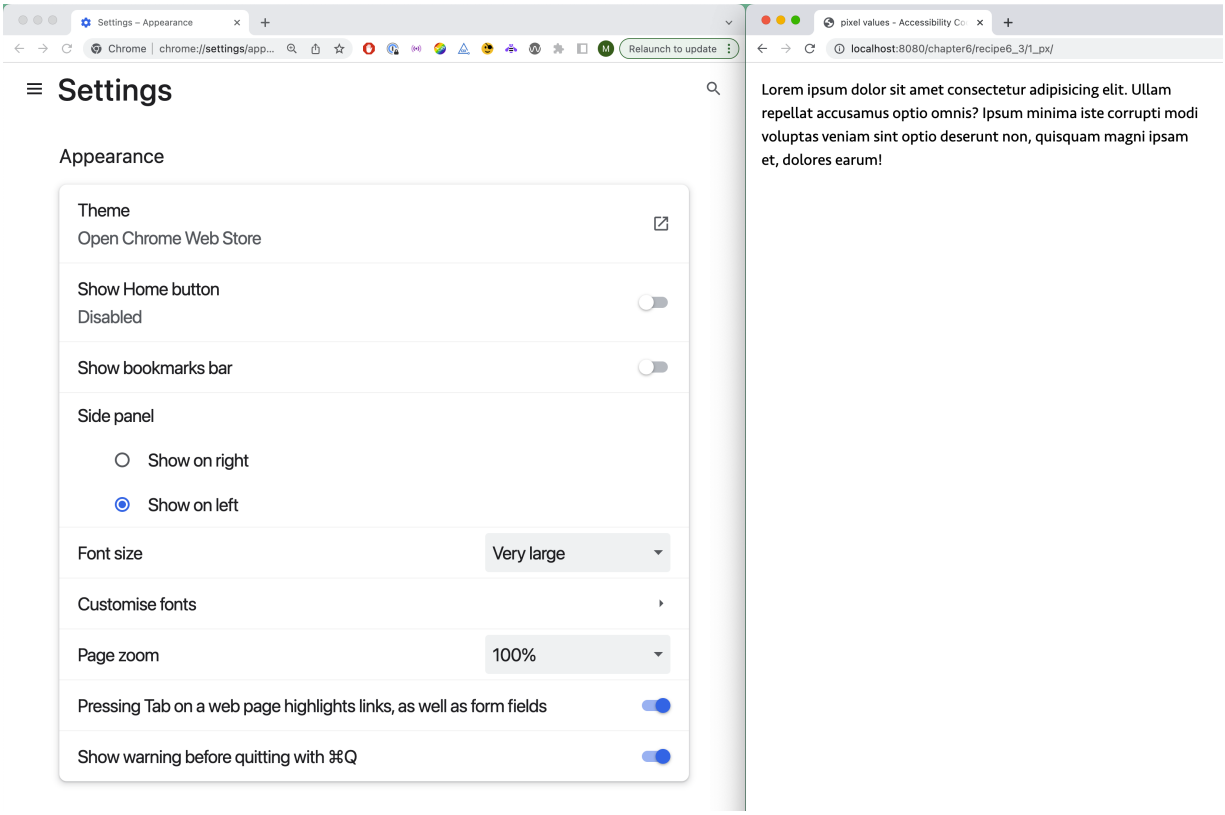


Figure 5-10. Font settings in Google Chrome on the left; a simple paragraph on the right

If you want to consider font-size preferences, use relative units for most of your sizing.

Relative units

CSS provides several relative units: `em`, `ex`, `rem`, `ch`, viewport units, numbers, and percentages.

If you need a different font size in some components, don't use fixed pixel values, as shown in Example 5-16, but make your declarations relative to the base font size defined by the user. You can do that by using the `rem` unit. `1rem` is relative to the font size of the root element; as already mentioned, `16px` in most browsers. By default, `1rem` equals `16px`. If the user changes their base font size to `24px`, `1rem` will equal `24px`.

Example 5-16. Bad practice: Setting font size, line height, and max-width using pixel values

```
.a-component {
  font-size: 18px;
  line-height: 27px;
  max-width: 450px;
}
```

If you want to convert `px` to `rem`, you take the target size and divide it by the conversion factor, which is the base font size of `16px`. For the `18px` in Example 5-16, that means dividing 18 by 16, which equals 1.125, as shown in Example 5-17 and Figure 5-11.

NOTE

The equation for converting pixels to rem is:

```
px target size / px base font size = rem size
```

Example 5-17. Setting the font size in rem, and line height and max-width using pixel values

```
.a-component {
  font-size: 1.125rem;
  line-height: 27px;
  max-width: 450px;
}
```

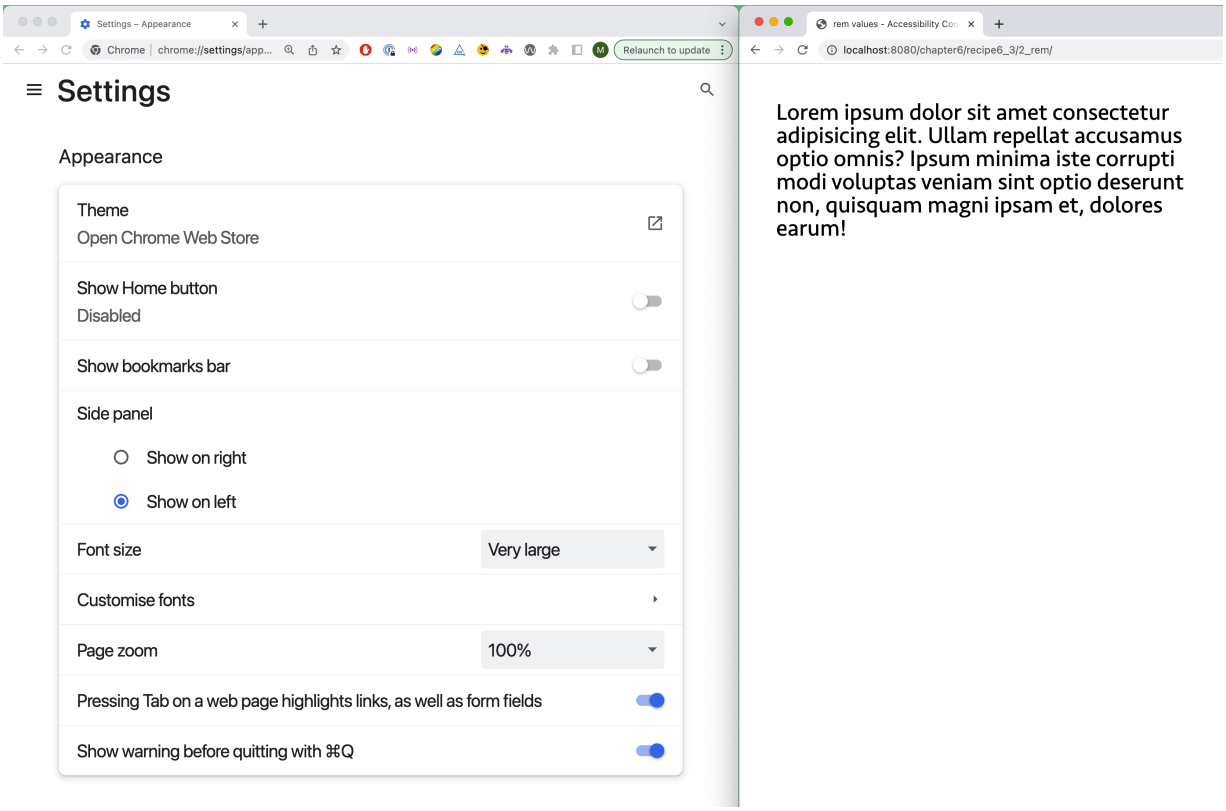


Figure 5-11. Text is larger because it's relative to the base font size; `line-height` still uses an absolute value

That's much better because now the font size responds to browser settings. The fixed px value of the line height worked well with the 18px, but it doesn't scale. You could use `rem` here, but `line-height` also supports number values (see Example 5-18.)

Example 5-18. Setting the font size in rem, line height using a multiplier, and max-width using pixel values

```
.a-component {
  font-size: 1.125rem;
```

```
    line-height: 1.5;
    max-width: 450px;
  }
```

The text feels a bit wedged because the maximum width of the paragraph didn't respond to the font size. However, you should also use `rem` for values of properties that should correspond visually with the root text size. To do that, you take our equation and divide 450 by 16, which equals 28.125 rem, as shown in Example 5-10.

The result is a UI that scales nicely with the user-preferred base font size, as Figure 5-12 shows.

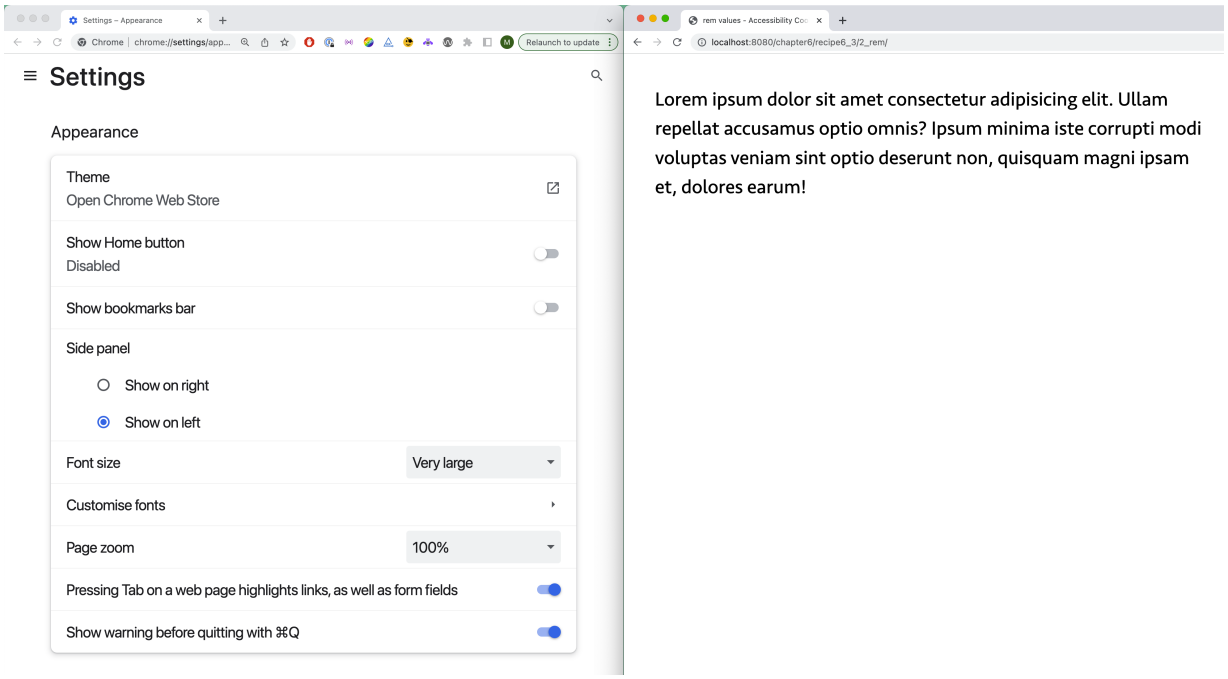


Figure 5-12. The readability of the paragraph is much better because of its improved line-height and width

Instead of using `rem` for paragraph widths, you can also use the `ch` unit, as shown in Example 5-11. One `ch` equals the width of the glyph 0 in the respective font, which allows you to define the maximum number of characters that have the same width as "0" that should fit in a line instead of using a length value. The ideal number of characters per line depends on several factors like the typeface, font size, and language, but a good rule of thumb is that it should be at most 60 to 80 characters.

The `rem` unit serves as the foundation of most of your sizing, but I don't recommend basing everything off the root font size. If you want to make an element's width relative to its parent's width, you can use percent, or if you want to make an element's height relative to the viewport, you can use viewport units, as shown in Example 5-12.

Another practical unit is `em`, which is relative to an element's font size. In Figure 5-13, you can see how the padding scales nicely with the font size of the button, thanks to the use of `em` in Example 5-13.

Figure 5-13. The padding in each button increases with the font size

Absolute units

`em` and `rem` don't replace the `px` unit; it still has its justification. In "Why You Should Use px Units for margin, padding, & Other Spacing Techniques", Ashlee M. Boyer explains that using px values for margin, paddings, or other properties related to spacing can be better for the user experience because when spacing grows, it can eat up vital real estate.

There is no yes or no to the question: should I use pixel values? You have to test it with your layout, but for properties like `margin`, `padding`, `border`, `box-shadow`, `text-shadow`, or `border-radius` it can be better to use `px`.

For the rest, it's worth preferring the `rem` unit over pixels because it respects users' preferences—on desktop, at least. The

situation is very different on mobile. According to my research, none of the tested mobile browsers treated `rem` differently than pixels. They all apply their own logic for scaling and zooming. Ultimately, it doesn't matter how browsers zoom text; what counts is that users can use that feature and that you build interfaces that respect user preferences. Part of that is working with relative units like `rem`.

Media Queries

I recommend using `em` or `rem` over `px` in your media queries, as shown in Example 5-14. Before I explain why, let me offer a quick overview of users' zooming options.

- In desktop browsers, they can change the base font size in their browser settings or press `Cmd` / `Ctrl` `+` / `-` to zoom the page.
- In mobile browsers, they can change the text size or zoom the entire page, depending on the operating system and browser.
- Users can pinch-zoom using a screen or trackpad in all browsers and operating systems that support it.

Example 5-19 shows three media queries. Each queries the minimum width of the viewport at `960px`. The first query uses

pixel, the second uses `em`, and the third uses `rem`. `60em` and `60rem` each equal `960px` by default (60 * 16 = 960).

Example 5-19. Comparison of the same media query but with different units

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

@media(min-width: 60em) {
  .wrapper-em {
    display: flex;
  }
}

@media(min-width: 60rem) {
  .wrapper-rem {
    display: flex;
  }
}
```

At a viewport width of 1280px and 100% zoom, all media queries apply and show a horizontal instead of a vertical layout, as shown in Figure 5-14.

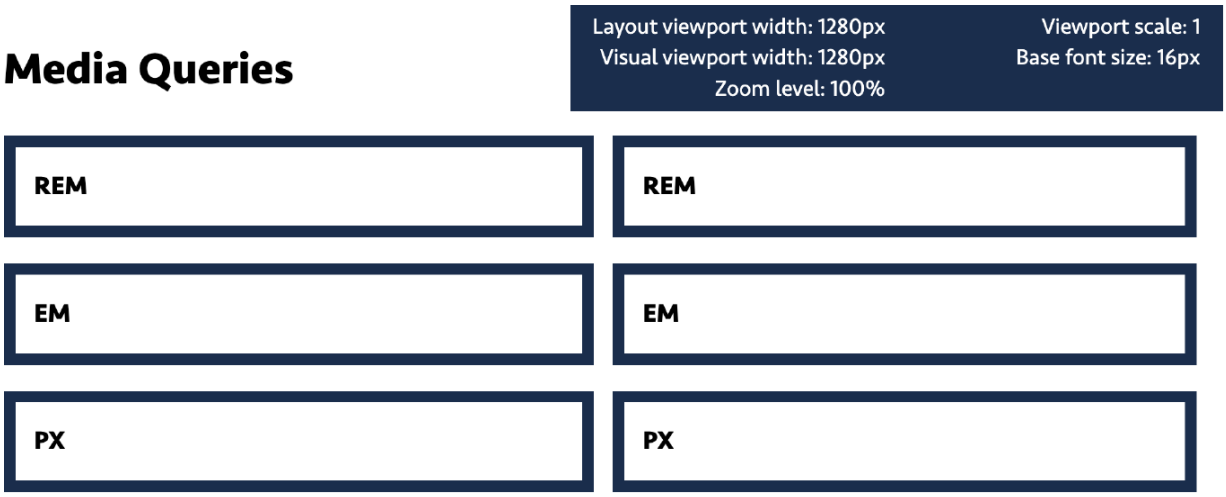


Figure 5-14. All media queries behave the same

Page zoom

`Cmd` / `Ctrl` `+` / `-` resizes the viewport in desktop browsers. If the viewport's width is 1280px by default and you zoom the page up to 150%, its new width is 853px (1280 / 1.5), causing the page's content to reflow. `em` and `rem` in media queries are relative to the base font size in the browser. Since pressing Cmd/Ctrl +/- doesn't change the font size but the viewport's width, all three media queries behave equally. In that case, the media queries aren't effective at a 150% zoom level, because the viewport's width is less than 960px, as shown in Figure 5-15. That's usually the expected and desired behavior.

Figure 5-15. All media queries behave the same at a zoom level of 150%

Text size

The `em` value in a media query is relative to the initial value of the font-size property `medium`. The `medium` keyword represents the scaling factor 1, which refers to the base font size in the browser. That means changing your browser's base font size to 22px affects the em- and rem-based media queries, as illustrated in Figure 5-16. After changing the setting, they fire at a viewport width of at least 1320px (60 * 22 = 1320), while the px-based media query still fires at 960px. The fact that the media queries fire later is good because the user gets a layout optimized for smaller screens. Indeed, the screen size didn't change,

but elements have less space due to the increased font size.

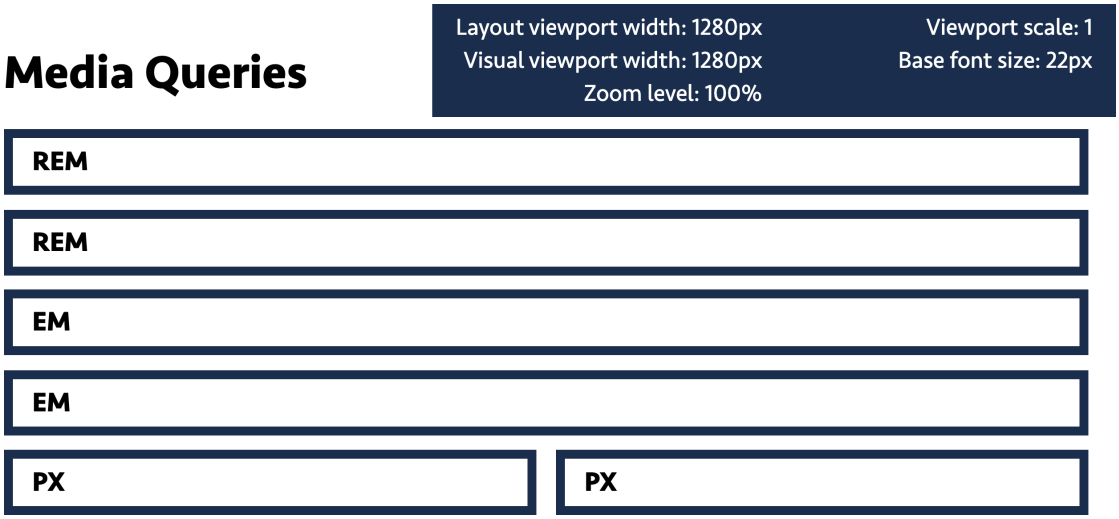


Figure 5-16. With a base font size of 22px, the rem- and em-based layouts show a single-column layout at a layout viewport width of 1280px

Pinch-zoom

When you pinch-zoom, you change the size of the visual viewport. Zooming in increases the size of the CSS reference pixel but scales the layout viewport proportionally (see Figure 5-17). It seems like you're increasing the page's size, but you're actually just decreasing the size of the visual area. That means pinch-zooming doesn't change the layout viewport's size and thus doesn't cause content to reflow. To help you better understand how that works, take a look at this interactive demo.

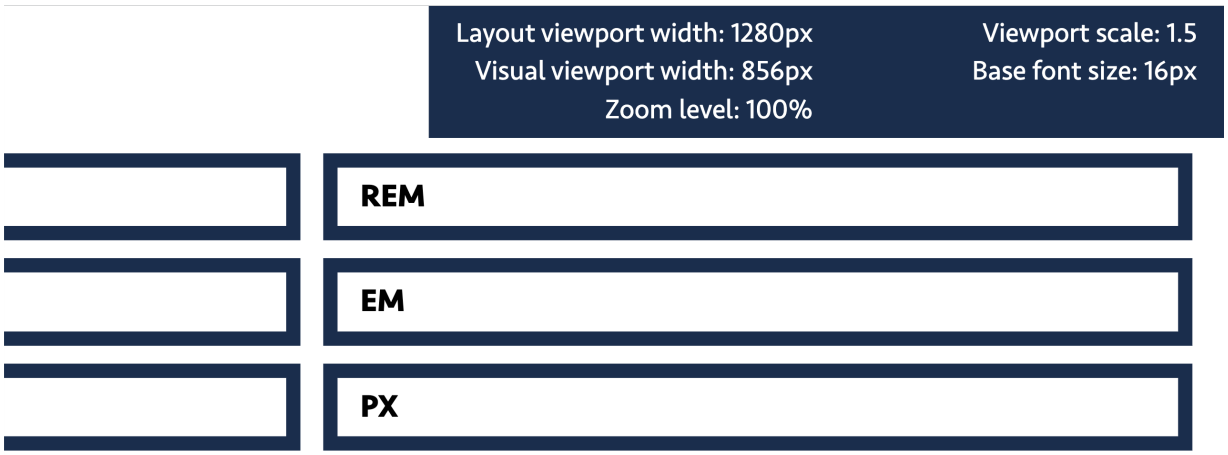


Figure 5-17. At a 150% page zoom, the entire UI scales

There used to be a time when using `em` or `rem` over `px` in media queries had a much more significant impact on the user experience. However, it's still worth preferring `em` over `px`, because users who have changed the base font size in their browsers benefit from it.

The Ideal Font Sizes and Line Height

The ideal font size depends on different factors, like the typeface and device. A good starting point is not setting a default font size, so it adapts to the default 16px in most browsers or whatever the user chose. You can increase or decrease the font size from there, although generally, I'd recommend against going lower than 16px for running text. UI designer and typographer Oliver Schöndorfer also recommends increasing the font size with the screen size, because the larger

the screen, the larger the distance between the eyes and the device. If you consider using fluid type, e.g., font sizes that scale with the viewport width, use a tool like Fluid Style to get the correct values and test your solution at different zoom levels.

Another factor is the line height. Larger line heights are usually more visually appealing and help people with cognitive disabilities track lines. A line height of 1.5 to 2 allows them to start a new line more easily after finishing the previous one. This rule of thumb mainly applies to running text. Line heights should decrease as the font size increases.

5.4 Preserve Semantic Information and Operability

Problem

Some CSS properties don't just affect an element's visual styling but also its semantic information. That can make it hard for users to identify an element using a screen reader or completely prevent access to the element or its data.

Solution

1. Be careful when using `display: contents;`. Especially avoid using it on interactive elements.
2. Be careful when overriding the display properties of table elements.
3. Apply `content-visibility` only to generic elements, as shown in Example 5-20.

   Example 5-20. The `content-visibility` property set on a wrapping `div`

   ```
   <div style="content-visibility: hidden">
     <section>
       <h2>…</h2>
       …
     </section>
   </div>
   ```

4. Don't use `display: none` or `visibility: hidden` on elements that name other elements.
5. Don't use `display: none` or `visibility: hidden` to visually hide elements that you still want in the accessibility tree.

Discussion

The following list is not comprehensive and is subject to change, because browsers constantly ship bug fixes. Nevertheless, most of the bugs or features I mention here have existed for many years and aren't likely to go away soon. If you're unsure whether a bug still exists, please do your own testing.

The code samples in this recipe are all bad practices that you should avoid. I've tested them using the operating systems, browsers, and screen readers listed in the preface of this book.

Buttons and links

Buttons or links with `display: contents` don't accept keyboard focus (see Example 5-21).

Example 5-21. Antipattern: The `display: contents` property and value set on a button

```
<button style="display: contents"></button>
```

This has already been reported. You can track it here: WebKit Bug 255149, Chromium Issue 1366037, and Mozilla Bug 1791648.

Links with zero dimensions (0 width, 0 height, no padding, and no border) don't accept keyboard focus in Safari (see Example 5-22).

Example 5-22. Antipattern: A link with 0 dimensions

```
<a href="#" style="display: block; height: 0; wi
```

Tables

Applying `display: contents` to tables, table rows, table heads, or table cells removes all semantic information for the elements in versions of Safari up to and including 16. That issue has been addressed starting with Safari 17.

Applying `display: flex|grid|block` to tables, table rows, table heads, or table cells removes all semantic information for the elements in versions of Safari up to and including 16. That issue has been addressed starting with Safari 17.

Using `display: none` on the `<caption>` element may remove the accessible names of tables in some browsers and screen readers.

Form elements

If you want to hide native form elements visually and replace them with custom solutions, don't use `display: none` or `visibility: hidden`, because these properties remove the form element from the accessibility tree, as shown in Example 5-23. Use the class `visually-hidden` instead (see Recipe 8.1).

Example 5-23. Antipattern: `display: none` removes the element from the accessibility tree

```
<div>
  <input type="checkbox" style="display: none" i
  <label for="toc">I accept the terms of service
</div>

<style>
[type="checkbox"] + label {
  /* custom checkbox styling */
}
</style>
```

The same applies if you want to hide the label of a form element visually, as Example 5-24 shows.

Example 5-24. Antipattern: `display: none` removes the label from the accessibility tree and with it the accessibile

name of the input field

```
<div>
  <label for="search" style="display: none">Sear
  <input type="text" id="search">
</div>
```

Using `appearance: none` on checkboxes removes the semantic meaning of the element using NVDA with Firefox.

Using `display: none` on the `<legend>` element may remove its parent `fieldset` from the accessibility tree.

Lists

Using `list-style: none` removes the semantic information of lists in VoiceOver on Safari. Instead of “list, 2 items” (see Example 5-25), the software may not announce the list as a list. This is by design. The WebKit team decided to remove list semantics when a list doesn’t look like a list. Their reasoning is that if a sighted user doesn’t need to know it’s a list, a screen reader user doesn’t need or want to know, either.

Example 5-25. An unordered list without list styling

```
<ul style="list-style: none">
  <li>Hello</li>
  <li>World</li>
</ul>
```

I understand their decision, but whether a user agent should intervene like that is debatable. However, there's a workaround if you need to maintain the semantic information. I have discovered that you can remove list styles without affecting semantics by setting the `list-style-type` property's value to an empty string instead of removing it, as shown in Example 5-26.

Example 5-26. Removing list styles without affecting semantics

```
<ul style="list-style-type: ''">
  <li>Hello</li>
  <li>World</li>
</ul>
```

All interactive elements

If you put `pointer-events: none` on an interactive element, the user can't activate it by using the mouse or pressing Enter.

They can still focus it using the keyboard. This means screen reader users will find it but won't know it's inactive, because it doesn't convey this state semantically.

All elements

Using `visibility: hidden` and `display: none` will remove an element from the accessibility tree. You can learn more about hiding content in CSS in Recipe 8.1.

Applying the `content-visibility` property on semantic elements may yield undesired results. If you put `content-visibility: hidden` on a heading, it's not rendered on screen, but it still might be in the accessibility tree. If you put it on a named region, you won't be able to access its contents, but you may still find an unlabelled, empty group.

The essential takeaway from this recipe is that you have to be careful with specific CSS properties. Ensure that your components still work as expected by testing, even if you make only stylistic changes.

5.5 Add Motion and Animation

Problem

Animations on the web can physically harm users. At their mildest, they can be annoying and distracting, but they can also cause nausea, dizziness, and headaches in some users. For people with vestibular disorders, it may even cause pain and make them feel so bad that they have to stop using the computer and take time to recover.

Solution

Avoid large-scale animations, or ship them only to users who have no preference for reduced motion.

You can react to user preferences in CSS, HTML, and JavaScript, as shown in Examples 5-27, 5-28, and 5-29.

Example 5-27. CSS: Picking the type of animation based on user preference

```
div {
  overflow: hidden;
}

.banner {
  --animation: fade; ❶

  translate: var(--position) 0;
  animation: var(--animation) 3s cubic-bezier(0.
```

```
  opacity: var(--opacity);
}

@media(prefers-reduced-motion: no-preference) {
  .banner {
    --animation: move;
  }
}

@property --position { ❸
  syntax: "<length>";
  inherits: false;
  initial-value: 0;
}

@property --opacity {
  syntax: "<number>";
  inherits: false;
  initial-value: 1;
}

@keyframes move {
  from { --position: 100vw; }
  to   { --position: 0; }
}


@keyframes fade {
  from   { --opacity: 0; }
```

```
    to   { --opacity: 1; }
  }
```

1. The default animation is “fade.”
2. If the user has no preference for reduced motion, change the animation to “move.”
3. You must register the custom properties to make them animatable.

Example 5-28. HTML: Show an animated gif or a jpg, depending on user preference

```
<picture>
  <source srcset="/assets/images/boy.jpg"
  media="(prefers-reduced-motion: reduce)" />
  <img src="/assets/images/boy.gif"
  alt="A boy looking at a computer and showing t
</picture>
```

Example 5-29. JavaScript: Picking scroll behavior based on user preference

```
const button = document.querySelector("button");
const motionQuery = matchMedia("(prefers-reduced
```

```
let behavior;

const handleReducedMotion = () => { ❷
  if (motionQuery.matches) {
    behavior = "smooth";
  } else {
    behavior = "instant";
  }
};

motionQuery.addListener(handleReducedMotion); ❸
handleReducedMotion(); ❹

button.addEventListener("click", (e) => {
  document.querySelector(".numbers > :last-child
    behavior: behavior,
  });
});
```

❶ Query the `prefers-reduced-motion` media feature.

❷ Use smooth or instant scrolling depending on user preference.

❸ Listen to changes to the OS setting.

❹ Run the function once when the page loads.

Discussion

Animation on the web is a powerful and essential tool for designers and developers. You can use it to create engaging and fun experiences and help users better understand interfaces and follow user flows by improving microinteractions. According to Val Head, a web animation expert, the most outstanding characteristics of animations are:

- They make an element's path visible on the screen, reducing cognitive load. Users don't have to keep track of its movement in their heads because it's loaded off to the animation on screen.
- They can help improve decision making.
- They help users build mental maps of spatial information.
- They can help prevent change blindness.
- They help establish connections and relationships between objects.

Val Head summarizes more useful applications of animation in "UI Animation and UX: A Not-So-Secret Friendship", as does Page Laubheimer in "The Role of Animation and Motion in UX".

When animation goes bad

There's no doubt that animation on the web can improve usability, user experience, and accessibility, but it can also have physical consequences for people with motion sensitivities. The root causes for motion sensitivity can be manifold, and so can its manifestation. Animation may cause nausea, dizziness, and headaches in some users. For people with vestibular disorders, it may even cause pain and make them feel so bad that they have to stop using the computer and take time to recover.

That doesn't apply to all motion on the screen. Fading transitions or slight movements aren't usually troublesome. More common triggers of motion sensitivity include:

- Animation that moves an object across a large amount of space
- Mismatched directions and speed, as you often see in parallax scrolling
- Animations that cover a large perceived spatial distance, like scaling and zooming
- Fixed background images (`background-attachment: fixed`)

> *Really, there are no words to describe just how bad a simple parallax effect, scrolljacking, or even background-attachment: fixed would make me feel. I would rather jump on one of those 20-G centrifuges astronauts use than look at a website with parallax scrolling.*
>
> —Facundo Corradini

Facundo Corradini, a developer who suddenly and unexpectedly suffered from a bad case of vertigo caused by labyrinthitis, describes how parallax scrolling triggered his symptoms in "Accessibility for Vestibular Disorders: How My Temporary Disability Changed My Perspective". He also explains that regular animations didn't trigger severe reactions but that anything moving on the screen would instantly break his focus. It took him a lot of conscious, focused effort to read if there was any movement on the page.

Reduced motion

When Apple released iOS7 for the iPhone, it came with many changes to the design and UI, some of which negatively affected users. The new OS made frequent use of zoom and slide animations and parallax scrolling. This reportedly made many people sick. Since then, iOS and most other operating systems

have added options to reduce motion in their system settings. These include:

- In Windows 11: Settings > Accessibility > Visual Effects > Animation Effects
- In macOS: System Preferences > Accessibility > Display > Reduce motion
- In iOS: Settings > Accessibility > Motion
- In Android 13: Settings > Accessibility > Color and Motion > Remove animations

The great news for users of your websites is that you can react to those preferences in HTML, CSS, and JavaScript and apply or reduce motion accordingly.

In Example 5-27, you can see that the element has a fade-in animation by default. Only if the user hasn't expressed a preference for reduced motion does it replace the fading with a moving animation. You can also use the media feature to replace animated images with static images, as illustrated in Example 5-28. In this example, you're actively querying the presence of the setting, not its absence, but you can also do it the other way around. Just like with any other media feature, you can also query `prefers-reduced-motion` in JavaScript, as shown in Example 5-29.

Examples of websites that have prominent animations but respect user preference are the web framework enhance, the Airpods Pro product page, and the website for the game Animal Crossing.

The media feature is named `prefers-reduced-motion` and not *prefers-no-motion* for a reason. The whole point is not to eliminate motion entirely, but to reduce nonessential movement to a minimum and ensure that critical elements still display.

Respecting user preferences also means giving users control. Avoid auto playing animations or videos, or at least allow users to pause and stop them, as shown in Figure 5-18.

Figure 5-18. An option to reduce motion on animalcrossing.nintendo.com

Animations and transitions are a powerful tool for designers and developers to improve the UX. When you add them to your websites, ensure that you do it purposefully, respect users' preferences, and give them control.

See Also

- "Diverse Abilities and Barriers (visual)" by WAI
- "Learn Accessibility: Color and contrast" by web.dev
- "Writing even more CSS with Accessibility in Mind, Part 2: Respecting user preferences" by Manuel Matuzović

- “Designing Safer Web Animation For Motion Sensitivity” by Val Head
- “Designing With Reduced Motion For Motion Sensitivities” by Val Head
- “Responsive Design for Motion” by James Craig
- “The ideal line length & line height in web design” by Oliver Schöndorfer
- “Display: Contents Is Not a CSS Reset” by Adrian Roselli
- “It’s Mid-2022 and Browsers (Mostly Safari) Still Break Accessibility via Display Properties” by Adrian Roselli
- “CSUN 2020: CSS Display Properties versus HTML Semantics” by Adrian Roselli

Chapter 6. Managing Focus

Web pages are keyboard accessible by default because native interactive elements come with the styling and functionality you need to use them out of the box. They are focusable, and they indicate their focus state visually. When you add CSS or especially JavaScript, you must ensure that you maintain that base accessibility or even improve it, and that your custom solutions are accessible as well.

6.1 Provide Focus Styles

Problem

When people access a website using a keyboard, switch device, screen reader, or similar assistive technology, they can use the `Tab` key (or a control on another physical device that maps to the key) to jump from one interactive element to another.

If you, as the developer, don't highlight the currently active item visually using CSS, users can't orient and navigate.

Solution

Use pseudoclasses to style interactive elements in their focus or focus-visible state, as shown in Examples 6-1, 6-2, 6-3, and 6-4.

Example 6-1. Styling all elements in their focus-visible state

```
:focus-visible {
  outline: 0.25em solid;
  outline-offset: 0.25em;
}
```

Example 6-2. Styling all elements in their focus state

```
:focus {
  outline: 0.25em solid;
  outline-offset: 0.25em;
}
```

Example 6-3. Showing a shadow on video and audio elements if a contained item is currently focused

```
:is(video, audio):focus-within {
  box-shadow: 0 0 10px 3px rgb(0 0 0 / 0.2);
}
```

Example 6-4. Providing custom focus styles for nonkeyboard users

```
button:focus-visible { ❶
  outline: 0.25em dashed black;
}

button:focus:not(:focus-visible) { ❷
  outline: none;
  box-shadow: 1px 1px 5px rgba(1, 1, 0, .7);
}
```

❶ Styles for keyboard users

❷ Styles for users of pointing devices

Discussion

Styling the currently focused element is one of the most important things you can do for keyboard accessibility. If you remove the default focus styles that all browsers come with, you're making many people's lives harder. Removing focus indicators for keyboard users is like hiding the cursor for mouse users. Unfortunately, you'll find the code in Example 6-5 on many websites. That's because, in the past, adding focus styles had undesired side effects for mouse users, such as showing an outline that stakeholders often considered aesthetically unpleasing or disturbing (it's rarely users who

complain). Fortunately, that's a problem of the past. Now, you have fine-grained control over who sees focus styles and when, and browsers use different heuristics to determine when to show them.

Example 6-5. Bad practice: No default outline on interactive elements

```
:focus {
  outline: none !important;
}
```

The :focus pseudoclass

The `:focus` pseudoclass applies when the user focuses an element using the keyboard, a mouse, or any other input form. Using the code in Example 6-2, the custom 0.25em outline shows when the user finds the button with a keyboard and when they click or tap it. Formerly `:focus` was in the user agent stylesheet of all browsers until they switched to `:focus-visible` as the default.

The :focus-visible pseudoclass

Starting with Chrome 90, Firefox 87, and Safari 15.4, the `:focus` pseudoclass still matches focusable elements, but user

agents only sometimes visibly indicate focus. Instead, they default to `:focus-visible`, which uses a variety of heuristics to indicate focus only when it's most useful to users. That's a crucial change, because now you can style focus indicators without worrying too much about when they show. If you want to change the default styles, you can use the `:focus-visible` pseudoclass, as shown in Example 6-1. In Figure 6-1, you can see a comparison of default and custom focus styles Google Chrome.

Figure 6-1. Default focus styles in Chrome on the left, custom focus styles on the right

There are minor differences between browser engines, but focus styles should show under only certain circumstances, such as:

- If the user interacts with the page using a keyboard or some other nonpointing device.
- If the element supports keyboard input, such as an input element or a text area.

- If you move focus using JavaScript and the previously focused element indicated focus, the newly focused element will show it, too.
- If the user has expressed a preference in their browser settings (for example, in Chrome, under Preferences, Accessibility, “Show a quick highlight on the focused object”).

Although there are no default styles for the `:focus` pseudoclass, you can still use it if you want to show the indicator regardless of the input method. You can even combine the pseudoclasses, as shown in Example 6-4. The first rule matches when the user interacts with the button using a nonpointing device, and the second only matches when using a pointing device.

The :focus-within pseudoclass

The `:focus-within` pseudoclass applies to any element that matches the `:focus` pseudoclass. Most importantly, it applies to an element whose descendants match the conditions for matching `:focus`. That means that you can use it to select an element that has children that are currently focused, as illustrated in Example 6-3.

Default focus styles

Browsers come with default styles for focusable elements. These are often called *focus-ring* because they all use the `outline` property, which adds an outline around the element's rectangle. Default styles are better than nothing, but they're usually not noticeable enough, lack color contrast, and their styling is inconsistent across browsers, as illustrated in Figure 6-2. That's why I recommend you provide your own rules.

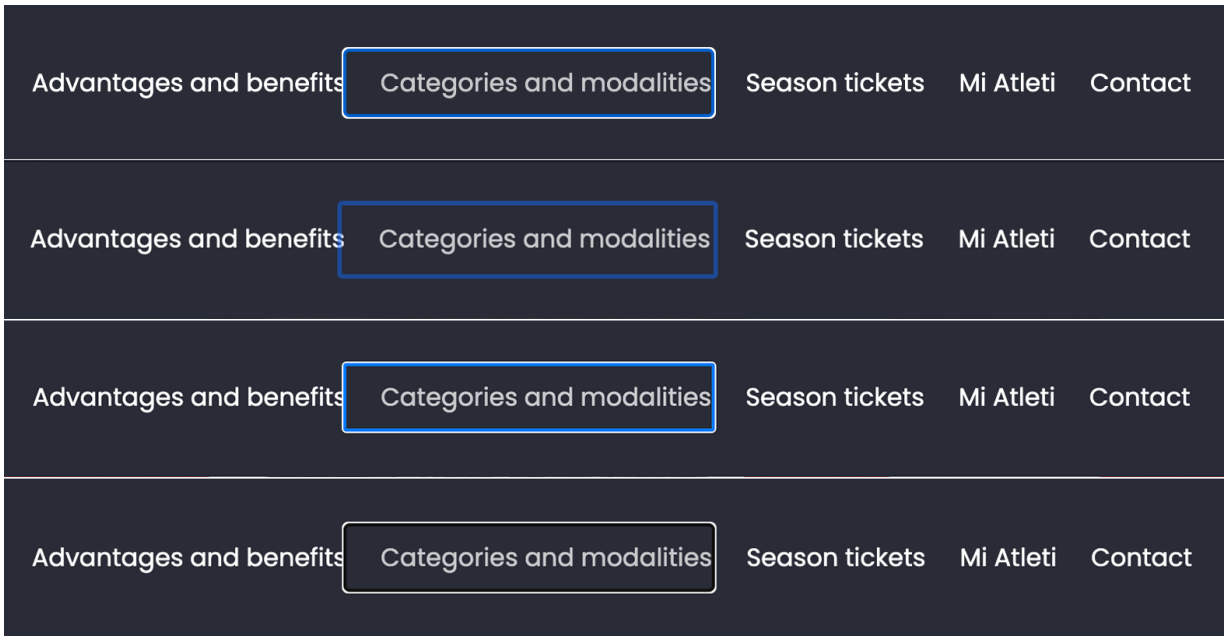


Figure 6-2. Default outline styles on atleticodemadrid.com: Chrome, Safari, Firefox, and Edge

The rule in Example 6-1 uses a thicker `outline-width` and an additional `outline-offset` to add space between the content and the outline. It also ensures that it meets the WCAG requirements for color contrast, which states that the visual

focus indicator must have sufficient contrast against adjacent colors.

In addition, `outline` properties work well for focus styling because they're well supported. They're clearly visible, and they don't interfere with the page layout because unlike `border`, they don't adjust the box size. You're not limited to these properties, though. You can also use properties like `background-color` or `box-shadow`, as illustrated in Figure 6-3.

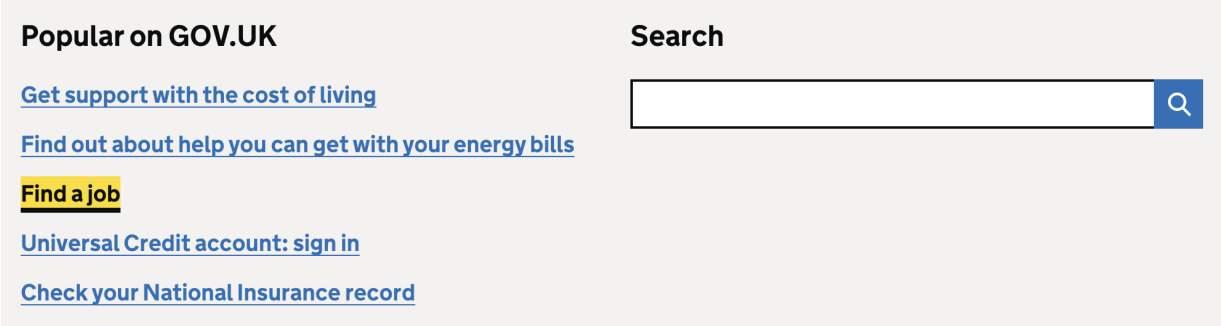


Figure 6-3. Custom focus style on gov.uk

However, it's important to know that these properties might not be visible in forced-color mode, as illustrated in Figure 6-4.

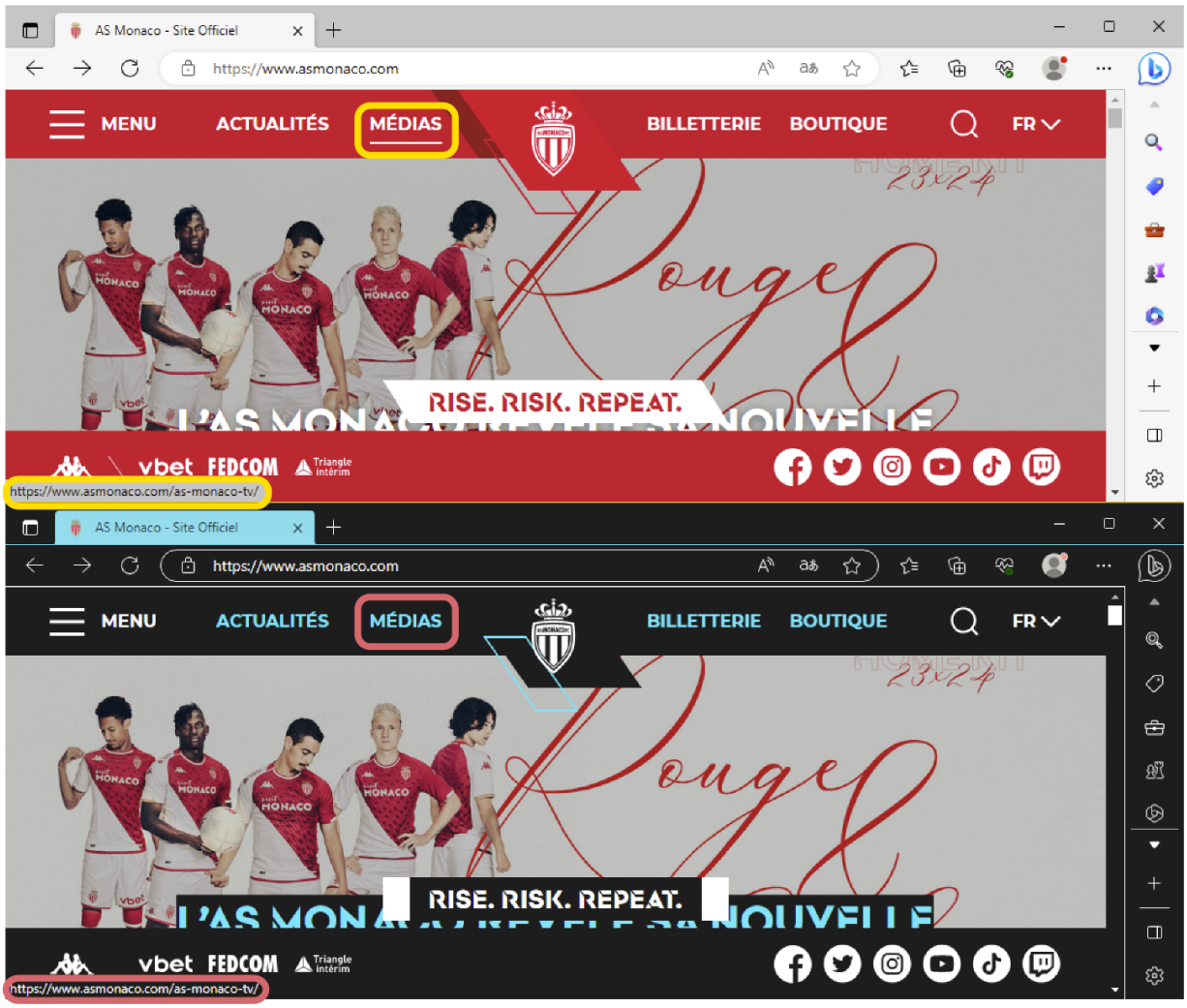


Figure 6-4. The underline below the “Medias” link is not visible in high contrast mode on Windows

A great workaround is to combine other properties with a transparent outline, as shown in Example 6-6. Forced-color modes might not render background colors and box shadows, but they may interpret and display transparent outlines.

Example 6-6. An invisible outline for all focused elements

```
:focus-visible {
```

```
    --black: #0b0c0c;
    --yellow: #fd0;

    color: var(--black);
    background-color: var(--yellow);
    box-shadow: 0 -0.125em var(--yellow), 0 0.25em
    outline: 0.25em solid transparent; ❷
  }
```

❶ Not visible in forced-colors mode

❷ Visible in forced-colors mode

Figure 6-5 shows how the gov.uk example (Figure 6-3) looks in forced-colors mode.

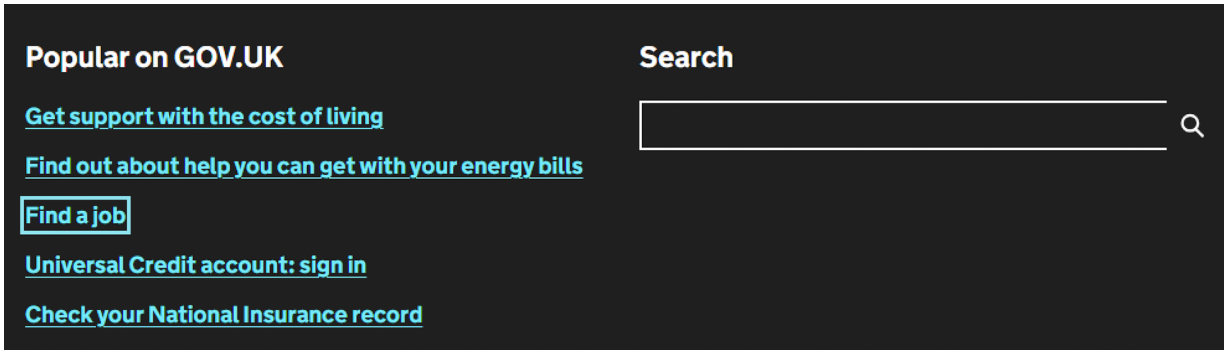


Figure 6-5. No background color and shadow in forced-colors mode, but the transparent outline is visible

When you design the focus styles of your site's interactive elements, focus on the user experience. They don't have to look

nice, but they have to do their job well.

6.2 Make Elements Focusable

Problem

People who rely on keyboard accessibility navigate using the `Tab` key, or a physical device that maps to the key, to jump from one interactive element to another. When they use a website, they must be able to perform the same actions mouse users can. If you don't take specific measures to ensure that, they might not be able to reach certain parts of the interface.

Solution

You can make nonfocusable elements focusable if necessary, as shown in Example 6-7.

Example 6-7. Making a tab panel focusable

```
<div class="tabs">
  <h3 id="tablist-1">Election results</h3>
  <div role="tablist" aria-labelledby="tablist-1

    <button id="tab-1" type="button" role="tab" a
            aria-controls="tabpanel-1">
```

```
        Graph
      </button>
      <button id="tab-2" type="button" role="tab" a
              aria-controls="tabpanel-2" tabindex=
        Table
      </button>
    </div>

    <div id="tabpanel-1" role="tabpanel" tabindex=
      <!-- graph -->
    </div>
    <div id="tabpanel-2" role="tabpanel" tabindex=
         class="is-hidden">
      <!-- table -->
    </div>
  </div>
```

Discussion

You can avoid many keyboard-accessibility issues before they arise by using semantic HTML. Several elements are intended for interactive content and are accessible by keyboard by default (see Table 6-1).

Table 6-1. Interactive elements

Element	Condition
`<a>`	If the `href` attribute is present
`<audio>`	If the `controls` attribute is present
`<button>`	
`<details>`	
`<embed>`	
`<iframe>`	
`<img>`	If the `usemap` attribute is present
`<input>`	If the `type` attribute is not set to `hidden`
`<select>`	
`<textarea>`	
`<video>`	If the `controls` attribute is present

The two solutions in Example 6-8 are semantically the same and can look identical with the right CSS. The big difference is that only the native button is accessible via the keyboard. Even if the

custom button was focusable, you wouldn't be able to perform any action on it because it has no key events (see Chapter 4).

Example 6-8. A native and a custom button

```
<button> ❶
  Open
</button>

<div role="button"> ❷
  Open
</div>

<style>
  button,
  [role="button"] {
    background: #123456;
    color: #FFFFFF;
    border: none;
    display: inline-block;
    padding: 0.5em 1rem;
    font-family: inherit;
    font-size: 1.1rem;
    line-height: 1;
  }

  :focus-visible {
    outline: 4px solid #123456;
```

```
    outline-offset: 4px;
  }
</style>
```

1. Focusable and keyboard-accessible by default
2. Not focusable and not keyboard-accessible

Using native HTML is the best approach. You usually get great baseline accessibility out of the box, but sometimes you need to make elements focusable that aren't by default. For example:

- When you have to move focus (see Recipe 6.3), but the target element is not focusable or has no focusable descendants
- When you want to make scrollable areas keyboard-accessible
- When you build a custom interactive element and have very good reason not to use a semantic element

The tabindex attribute

The code in Example 6-7 shows a custom JavaScript solution for a tabs component. Try out the demo on the ARIA Authoring Practices Guide (APG) website by the W3C, where this code is from. You'll learn that you can interact with the buttons in the tab list by using the arrow left and right keys. To access the content for a tab, you can use the `Tab` key, and the

corresponding tab panel receives focus. The tab panel is a `div` element with a custom `tabpanel` role. It's focusable only because the page uses the `tabindex` attribute with a value 0. That attribute and value make an element focusable and adds it to the natural tab order.

You'll also notice that the buttons have a `tabindex` attribute with `-1`, which can be useful for two things: when you want to make a generic element focusable using JavaScript's `focus()` method without making it keyboard-accessible (see Recipe 6.3 for an example), or when you want to remove an element from the tab order but still be able to focus it via JavaScript. Buttons are focusable by default, but in this example, pressing Tab must skip the buttons and focus the active tabpanel instead. Setting `tabindex=-1` makes that possible while allowing you to put focus on the buttons when the user presses the arrow keys.

Using `tabindex` with the values 0 or -1 is a common practice and unproblematic if you use it on a semantic and labeled element. Problems arise when you use a positive number larger than 0, as shown in Example 6-9. A keyboard user navigates a website by jumping from one interactive element to another in sequential order. The Tab key always follows the order of elements in the document, *unless* you use `tabindex` with a positive value.

Example 6-9. Bad practice: Buttons with a custom tabbing order

```
<button tabindex="2">Button 1</button>
<button tabindex="1">Button 2</button>
<button tabindex="3">Button 3</button>
```

The buttons in Example 6-9 receive focus in the following order: **Button 2**, **Button 1**, **Button 3**. The custom order defined by the `tabindex` attributes overrules DOM order.

If only some elements have the attribute, focus will find these elements first and traverse them based on their value, then traverse the elements without the attribute in sequential order. In Example 6-10, that would give you **Button 2**, **Button 4**, **Button 1**, **Button 3**, **Button 5**. Maintaining a meaningful order becomes complicated once you add `tabindex` attributes with values larger than zero.

Example 6-10. Bad practice: Only two buttons have a custom positive `tabindex` attribute

```
<button>Button 1</button>
<button tabindex="1">Button 2</button>
<button>Button 3</button>
```

```
  <button tabindex="2">Button 4</button>
  <button>Button 5</button>
```

The `tabindex` attribute is powerful, versatile, and useful for improving keyboard accessibility, but it's also dangerous. Using positive values will get out of hand quickly. Follow these rules:

- Use `tabindex=-1` to make nonfocusable elements focusable via JavaScript only or to make focusable elements nonfocusable.
- Use `tabindex=0` to make nonfocusable elements keyboard-focusable.
- Avoid using positive values.

Focusable elements

Knowing which elements are focusable can be useful, but listing them all is more challenging than it sounds. In addition to the list of interactive elements at the beginning of this recipe, we have elements with a positive tabindex. Then there's also the `contenteditable` attribute, which makes elements editable. All that is true only if the elements are not hidden, disabled, or inert, and if they don't have a tabindex with a negative value.

6.3 Move Focus

Problem

When you move the users' attention to another place or layer within the page (for example, by showing a modal dialog or some other overlay), you must ensure they can also immediately interact with that content without additional effort. Consequently, you must move focus programmatically to that part of the UI. If you don't, users might still be on an interactive element on the page in the background. Pressing `Tab` would then result in focusing elements that are obscured and irrelevant. That can be confusing and prevents people from accessing important information and functionality.

Solution

You have to move focus to where it's currently needed. That requires you to shift focus, remember where it was before, and return to that spot once the action is completed. That process is called *focus management*. Examples 6-11 and 6-12 illustrate how to create a custom modal window and move focus from the button to the modal and back again, as needed.

Example 6-11. A button and a custom modal dialog

```
<button class="open" aria-haspopup="dialog">Logi

<div role="dialog" aria-modal="true" hidden aria
  <div>
    <button class="close">Close</button>

    <h1 id="heading" tabindex="-1">Login</h1>
  </div>
</div>

<style>
  [role="dialog"] { ❶
    align-items: center;
    background: rgb( 0 0 0 / 0.2);
    inset: 0;
    justify-content: center;
    margin: auto;
    position: fixed;
  }

  [role="dialog"]:not([hidden]) { ❷
    display: flex;
  }

  [role="dialog"] > div { ❸
    background: rgb(255 255 255);
```

```
      box-sizing: border-box;
      max-width: 40rem;
      padding: 2rem;
      width: 90vw;
    }

    :focus-visible {  ❹
      outline: 4px solid #123456;
      outline-offset: 0.25em;
    }

    [role="dialog"]:focus-visible {  ❺
      outline-offset: -0.5em;
    }
  </style>
```

❶ Base styling for the dialog

❷ Changes the display of the dialog when it's visible

❸ Base styling for the dialog's content

❹ Shows focus style for all elements

❺ Moves the outline on the dialog inside so that it's visible

Example 6-12. Simple focus management example

```
const dialogOpen = document.querySelector(".open
const dialogClose = document.querySelector(".clo
const dialog = document.querySelector('[role="di
const heading = dialog.querySelectorAll('[tabind
let trigger;

dialogOpen.addEventListener("click", open);
dialogClose.addEventListener("click", close);

function open() {
  trigger = document.activeElement; ❶
  dialogToggle();  ❷
}

function close() {
  dialogToggle();  ❸
}

function dialogToggle() {
  const isOpen = !dialog.hasAttribute("hidden");

  if (!isOpen) {
    dialog.removeAttribute("hidden");
    heading.focus(); ❹
  } else {
    dialog.setAttribute("hidden", "hidden");
    trigger.focus(); ❺
```

```
    }
  }
```

❶ Remember the element that triggered the action.

❷ Show the dialog.

❸ Hide the dialog.

❹ Move focus from the button to the dialog's heading.

❺ Move focus from the heading back to the button.

If you use the native modal dialog in HTML, you don't have to manage focus, as shown in Example 6-13.

Example 6-13. A button and a native modal dialog

```
<button class="open" aria-haspopup="dialog">Logi

<dialog aria-labelledby="heading">
  <form method="dialog"> ❶
    <button>Close</button>
  </form>

  <h1 id="heading">Login</h1>
</dialog>
```

```
<script>
const dialogOpen = document.querySelector(".open
const dialog = document.querySelector("dialog");

dialogOpen.addEventListener("click", open);

function open() {
  dialog.showModal(); ❷
}
</script>
```

❶ Closes the dialog without needing JavaScript.

❷ The native `showModal()` method opens the modal dialog.

Discussion

Example 6-12 illustrates the basic concept of focus management: you remember the last focused element by storing a reference to it in a variable using `document.activeElement`. Before you move focus, you have to decide where you want to put it. You have several options:

- The first focusable element, if there is one. That works, but users might land somewhere in the middle or end of the content, depending on its position in the DOM.
- The close button, if there is one. That's a safe option if you can't guarantee that there are other interactive elements. A downside is that it can be irritating when the first thing a screen reader user hears is "close button."
- If there's a heading, you can make it focusable using `tabindex=-1`. A heading is usually at the beginning of the content, and by focusing on it, a screen reader automatically announces its text content.
- Like the heading, you can make the labeled dialog focusable using `tabindex=-1`.

Your choice will depend on your possibilities and constraints and what works best for your users. I prefer focusing the heading.

Once the user is done interacting with the content in the dialog, you move focus back to the button that opened it.

Example 6-11 uses a custom dialog solution, but in HTML there's also a native dialog element (see Example 6-13). That element is handy because it builds in many of the things you used to have to do manually:

- It has an implicit dialog role.
- It provides methods for opening and closing.
- It makes the rest of the page inert.
- It manages focus.

The specification defines the rules for focus management as follows:

- If there is a focusable descendant, focus it.
- If not, focus the dialog element itself.
- If an element, including the dialog, is present with an `autofocus` attribute, take that instead.

Focus management is essential for making complex UIs accessible, but, as inclusive-design advocate Eric Bailey puts it, “99% of the time, you want to leave focus order alone.” You almost always want focus to follow the natural order and let users decide when and where to go next. You should programmatically manage focus only when users can’t reach parts of the interface without a lot of effort. That also applies to the autofocus attribute in HTML, which you can also use to move focus. You most certainly want to avoid moving focus in most navigations, disclosure widgets, and forms (except when you focus erroneous form controls).

Sometimes it's not the user who triggers the action. The first time you visit many European websites, a dialog opens automatically to show a cookie consent banner. When you don't manage focus in cases like that, users may interact with the page in the background without being able to close the overlay.

6.4 Keep Focus Contained

Problem

If you move focus, and with it the center of attention, you want to ensure that the rest of the page is inactive so that the user can interact only with the currently relevant content. If you don't do that, they can accidentally move out of that area, get lost, and have a hard time finding their way back, as shown in Examples 6-14, 6-15, and 6-16.

Solution

Contain or trap focus within the currently relevant area of the page and make the rest unreachable.

Example 6-14. Focus trapped within a custom modal dialog

```
<main>
```

```
    <button class="open">Login</button>
  </main>

  <div role="dialog" aria-modal="true" hidden aria
    <div>
      <button class="close">Close</button>

      <h1 id="heading" tabindex="-1">Login</h1>

      <a href="#">A link</a>
    </div>
  </div>

  <script>
    const dialogOpen = document.querySelector(".op
    const dialogClose = document.querySelector(".c
    const dialog = document.querySelector('[role="
    const heading = dialog.querySelectorAll('[tabi
    const focusableElements = dialog.querySelector

    let trigger;

    dialogOpen.addEventListener("click", open);
    dialogClose.addEventListener("click", close);

    dialog.addEventListener("keydown", (e) => {
      if (e.code !== "Tab") return; ❷

      const first = focusableElements[0]; ❸
```

```
    const last = focusableElements[focusableElem
    const active = document.activeElement; ❺

    if (e.shiftKey) { ❻
      if (first === active) { ❼
        e.preventDefault();
        last.focus(); ❽
      }
    } else if (last === active) {  ❾
      e.preventDefault();
      first.focus();
    }
  });

  function open() {
    trigger = document.activeElement;
    dialogToggle();
  }

  function close() {
    dialogToggle();
  }

  function dialogToggle() {
    const isOpen = !dialog.hasAttribute("hidden"


    if (!isOpen) {
      dialog.removeAttribute("hidden");
```

```
        heading.focus();
      } else {
        dialog.setAttribute("hidden", "hidden");
        trigger.focus();
      }
    }
  </script>
```

1. Find all focusable elements within the dialog.
2. Do nothing unless the user presses the `Tab` key.
3. This is the first focusable element within the dialog.
4. This is the last focusable element within the dialog.
5. This is the currently focused element.
6. When the users presses the `Shift` key…
7. …and the currently focused element is the first element…
8. …prevent the default behavior and move focus to the last focusable element.
9. If the user doesn't press `Shift` and the last item is the currently focused element, focus moves to the first

element.

Example 6-15. Focus contained within a custom dialog element

```
<div class="page-wrapper"> ❶
  <header>
    <a href="#">Home</a>
  </header>

  <main>
    <button class="open" aria-haspopup="dialog">
  </main>
</div>

<div role="dialog" aria-modal="true" hidden aria
  <div>
    <button class="close">Close</button>

    <h1 id="heading" tabindex="-1">Login</h1>

    <a href="#">A link</a>
  </div>
</div>


<script>
const pageWrapper = document.querySelector('.pag
const dialogOpen = document.querySelector(".open
```

```
const dialogClose = document.querySelector(".clo
const dialog = document.querySelector('[role="di
const heading = dialog.querySelectorAll('[tabind
let trigger;

dialogOpen.addEventListener("click", open);
dialogClose.addEventListener("click", close);

function open() {
  trigger = document.activeElement;
  pageWrapper.setAttribute('inert', 'inert') ❷
  dialogToggle();
}

function close() {
  pageWrapper.removeAttribute('inert') ❸
    dialogToggle();
}

function dialogToggle() {
  const isOpen = !dialog.hasAttribute("hidden");

  if (!isOpen) {
    dialog.removeAttribute("hidden");
    heading.focus();

  } else {
    dialog.setAttribute("hidden", "hidden");
    trigger.focus();
```

```
    }
  }
  </script>
```

1. All main page content is contained in a single element.
2. The main content wrapper becomes inert when the dialog opens.
3. The inert status is removed when the dialog closes.

Example 6-16. Focus contained natively within a dialog element

```
<button class="open">Login</button>

<dialog tabindex="-1" aria-labelledby="heading">
  <form method="dialog">
    <button>Close</button>
  </form>

  <h1 id="heading">Login</h1>
</dialog>


<script>
const button = document.querySelector("button");
const dialog = button.nextElementSibling;
```

```
button.addEventListener("click", (e) => {
  dialog.showModal(); ❶
});
</script>
```

❶ The `showModal()` method contains content automatically.

Discussion

There is a difference between trapping focus and containing focus. Example 6-14 shows a classic focus trap: When the user reaches the last focusable element and presses `Tab`, focus jumps to the first element. When they find the first element and press `Shift` + `Tab`, focus jumps to the last focusable item. There is no way of escaping the dialog other than closing it unless you use a screen reader. We've been using that technique for a long time for modal windows. It's an effective way of ensuring the user can't accidentally escape the element, but there are two potential issues with that.

First, users might not expect that behavior because, typically, when they reach the end or beginning of the page, focus moves to the browser UI.

Second, in Recipe 6.2 you learned about the elements that can receive focus. If you combine that list with all the attributes that can add or remove this ability, the list becomes long and hard to maintain, potentially missing some elements. The solution in Example 6-14 works only because it's based on the assumption that the dialog contains only buttons and links.

Many people still went for focus traps because until recently there was no other way of containing focus. In early 2023, all major browsers added support for the `inert` attribute. The custom dialog in Example 6-15 doesn't trap focus but contains it, by making the rest of the page inactive. Everything but the dialog is wrapped in a `div` with the `.page-wrapper` class. When the dialog is open, the script adds the `inert` attribute to that `div`, making all its descendants inactive. Visually they're still there, but you can't focus them, and they're temporarily not exposed in the accessibility tree. Doing that limits interaction to the currently relevant content without trapping the user, who can still escape the page. This solution is much closer to the native behavior and is safer and easier to implement. The native dialog element (see Example 6-16) comes with that behavior by default.

The `inert` attribute and dialog element make it much easier to ensure that users don't get lost elsewhere on the page, but focus

traps still may have advantages. The best way to determine whether you need one is to test with users and decide based on their feedback.

6.5 Preserve Order

Problem

A mismatch between the order of elements in the document and the visual order can result in confusion and unpredictability. That makes it hard for keyboard and screen reader users to navigate, get oriented, and understand the UI.

Solution

When you create layouts in CSS, let elements flow naturally on the x- and y-axes and refrain from using attributes and properties that affect visual or tabbing order, as shown in Examples 6-17 and 6-18.

Example 6-17. Content naturally aligned on the x-axis

```
ul {
  display: flex;
```

```
    gap: 1rem;
  }
```

Example 6-18. A two-column layout following the natural flow of the content

```
main {
  display: grid;
  grid-template-columns: 2fr 1fr;
  gap: 1rem;
}
```

Discussion

When you visit a page and press `Tab`, focus jumps to the first focusable element in the DOM. When you press it again, it jumps to the second element, etc. That's a predictable and expected behavior. When you change the layout of a page using CSS, you must ensure that the experience remains that way. The order must be sequential, even when you turn a vertical into a horizontal list or arrange items in a grid. For most natural languages, like English, French, or German, that means that focus moves from top to bottom, from left to right. In other languages, like Hebrew, it moves from top to bottom, from right to left.

The most important thing you must know is that changing visual order doesn't affect tab order. The first link that receives focus in Example 6-19 is both the first element in the document and on the page.

Example 6-19. A list of three links

```
<ul>
  <li><a href="#">One</a></li>
  <li><a href="#">Two</a></li>
  <li><a href="#">Three</a></li>
</ul>
```

That's also true when you let elements naturally flow on the x-axis using Flexbox (see Example 6-20). In Figure 6-6, you see how the link in the first list item is the first element to receive focus.

Example 6-20. Aligning the list horizontally

```
ul {
  display: flex;
  gap: 1rem;
  list-style-type: none;
}
```

Figure 6-6. The first focusable element in the document and on the page receives focus

If you break the natural flow and rearrange items manually, as shown in Example 6-21, you get a different visual order, but tab order stays the same. Figure 6-7 shows that the link in the first list item is still the first element to receive focus.

Example 6-21. Bad practice: Changing the natural order using the order property

```
ul {
  display: flex;
  gap: 1rem;
  list-style-type: none;
}

li:first-child {  order: 2 }
li:nth-child(2) { order: 3 }
li:last-child {   order: 1 }
```

Figure 6-7. The first focusable element in the document, visually the second element on the page, receives focus

If the visual order and the DOM order don't match, it can irritate and confuse users until the experience is so bad that the site is unusable. For instance:

- Visual order concerns keyboard users because they may have trouble predicting where focus will go next.
- It may irritate users of screen magnifiers if the enlarged portion of the screen skips around a lot.
- If a blind user works with a sighted user who reads the page in visual order, it may confuse them when they encounter information in different orders.

In Figure 6-8, I use the Polypane browser to visualize tab order on psg.fr. The numbered circles represent the current tab stop, and the lines show my path. The screenshot is hard to interpret because lines cross a lot. That indicates a mismatch between DOM and visual order.

Figure 6-8. Lines and numbered circles represent the tabbing order

Having learned about the `tabindex` attribute in Recipe 6.2, you may think that you can compensate for the issue caused by the code in Example 6-21 by adapting the tabindex of each element accordingly, as shown in Example 6-22. If you use the `Tab` key, DOM order and visual order now match again, but the virtual cursor in a screen reader ignores the attribute, still following the original DOM order. On top of that, now you have to manage and maintain a meaningful tabbing order for the whole page, which is cumbersome and error-prone.

Example 6-22. Bad practice: Changing the natural DOM order using the `tabindex` property

```
<ul>
  <li><a href="#" tabindex="2">One</a></li>
  <li><a href="#" tabindex="3">Two</a></li>
  <li><a href="#" tabindex="1">Three</a></li>
</ul>
```

Don't use tab indexes with a value larger than 0, and don't reorder interactive elements using CSS properties like `flex-direction`, `order`, `grid-auto-flow`, or techniques like explicit placement in Grid, absolute positioning, or negative margins. When you deal with interactive elements, visual order should always represent DOM order as well as possible, because tab order follows DOM order no matter how you arrange items visually.

6.6 Allow Users to Skip Elements

Problem

For people relying on keyboard accessibility, encountering parts of a web page with many interactive elements means they have to perform a physical action repeatedly, like pressing the `Tab` key on their keyboard, pressing a button on a switch device, or

operating a similar assistive technology with their hands, arms, mouth, or head. This can quickly become physically demanding.

Solution

Add links to areas in a page with many interactive elements, so users can skip those and continue navigating elsewhere, as shown in Example 6-23.

Example 6-23. A link in the header of the page that allows users to skip to the main content

```
<header>
  <a href="#content" class="skip-link">Skip to c
  <nav>
    …
  </nav>
</header>
<main id="content">

</main>
```

1. Visually hidden but accessible via keyboard and screen reader.

Example 6-24 shows how to hide the skip link from mouse and touch users and make it visible only when it receives keyboard focus.

Example 6-24. The skip link is visually hidden and visible only when it receives focus

```
.skip-link {
  background-color: #fff;
  position: absolute;
  padding: 0.2em;
  display: block;
}

.skip-link:not(:focus):not(:active) { ❶
  clip-path: inset(50%);
  height: 1px;
  overflow: hidden;
  white-space: nowrap;
  width: 1px;
}
```

❶ Hide the link only when it's not focused or active.

Discussion

Figure 6-9 is a screenshot of mit.edu. In the header at the top, there are 10 interactive elements, there are 7 more in the sidebar on the left. Depending on which page keyboard users accesses, they'll have to press the `Tab` key (or a button on a switch device) 10 to 17 times on every page they visit. That's annoying and can physically hurt and even aggravate repetitive strain injuries.

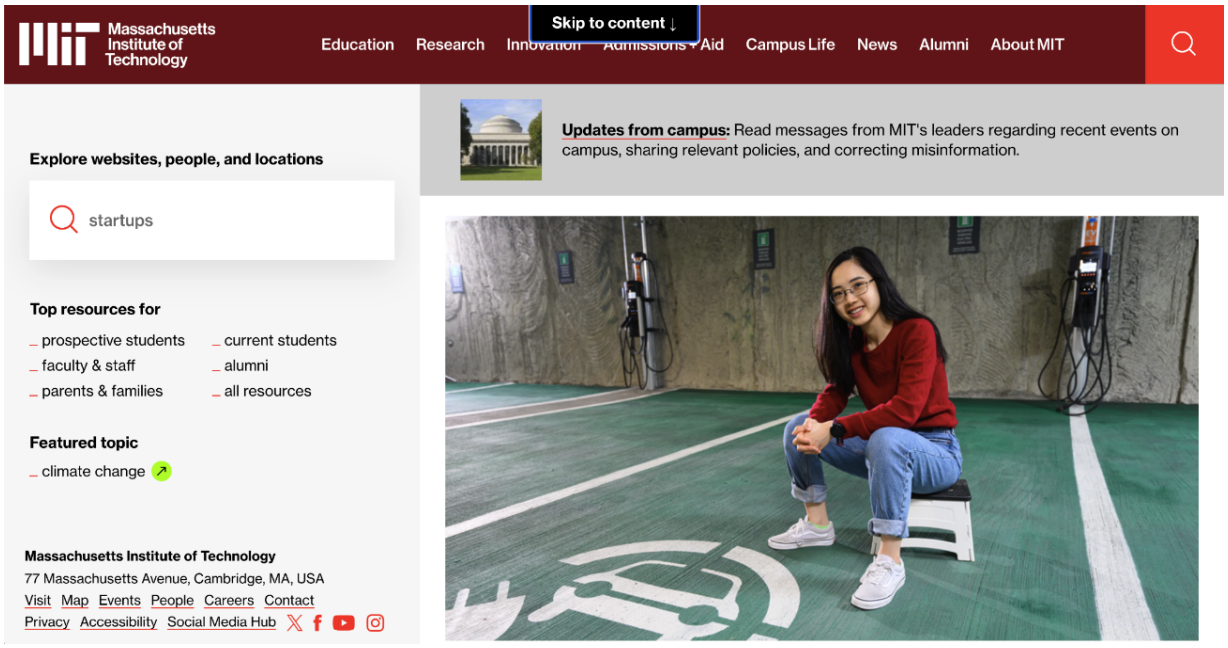


Figure 6-9. Skip link visible on mit.edu, a website with more than 15 focusable elements before the main content

Luckily, there's a skip link on the MIT website. A *skip link* is an anchor link that points to another part of the page, enabling you to jump to it by clicking the link. It's usually the first focusable element in the DOM, visually hidden and visible only

when it receives focus. Examples 6-23 and 6-24 illustrate a typical implementation.

Skip links can be incredibly useful when there are a lot of interactive elements before the page's main content. Using the example in Figure 6-9 again, with the skip link (visible in the top center of the page because it's currently focused) users have to press `Tab` only once instead of 17 times to get to the main content of the page. That being said, using a skip link may be counterproductive if there are only one or two focusable elements before the main content.

On most sites that implement skip links, you'll find a single "skip to content" link in the header, but some provide additional links for accessing the navigation, search, or footer. Whether you need that depends on the size of your site and the information density of your header and footer. The *Financial Times* implements four skip links that go to navigation, content, footer, and an accessibility statement (see Figure 6-10).

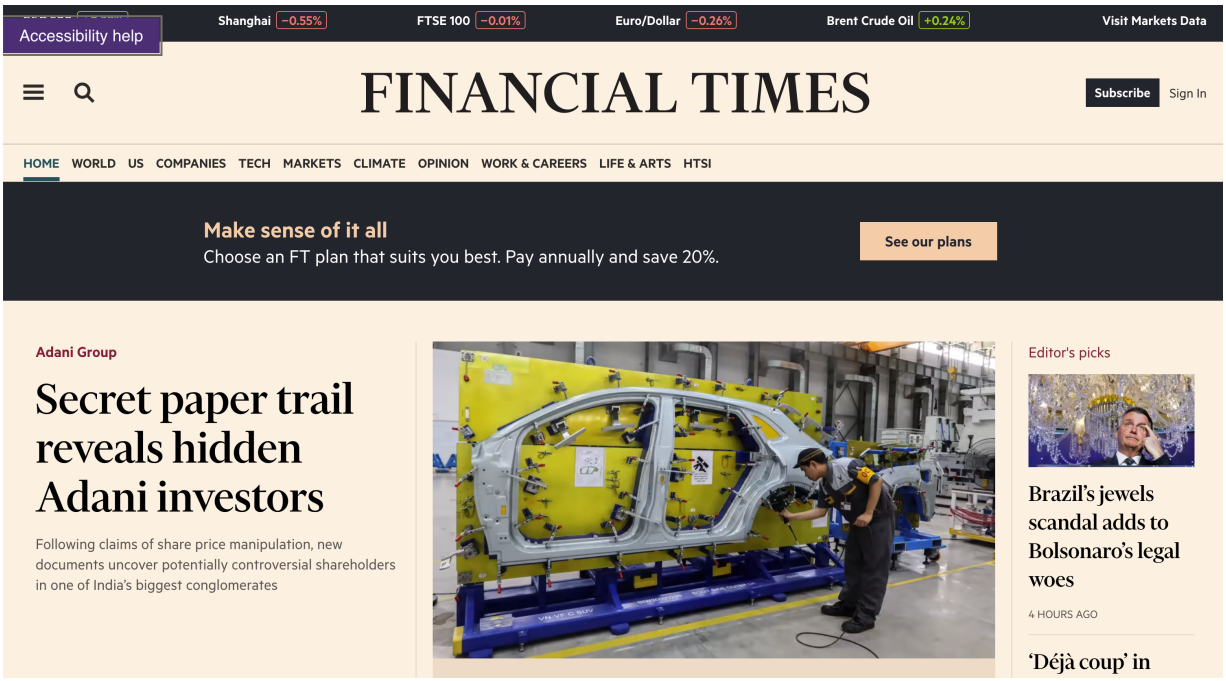


Figure 6-10. “Accessibility help,” one of four skip links on ft.com in the top left corner of the page

When you have sections with many interactive elements, adding skip links within the page can be helpful, too. Typical examples are social-media widgets, maps, and sidebars. In Figure 6-11, you can see many links in the sidebar of Manuel Baisch’s website. The skip link is always visible and allows you to access the page content directly.

Embedded maps often contain many interactive elements. Adding skip links before maps, as shown in Figure 6-12, can improve the user experience.

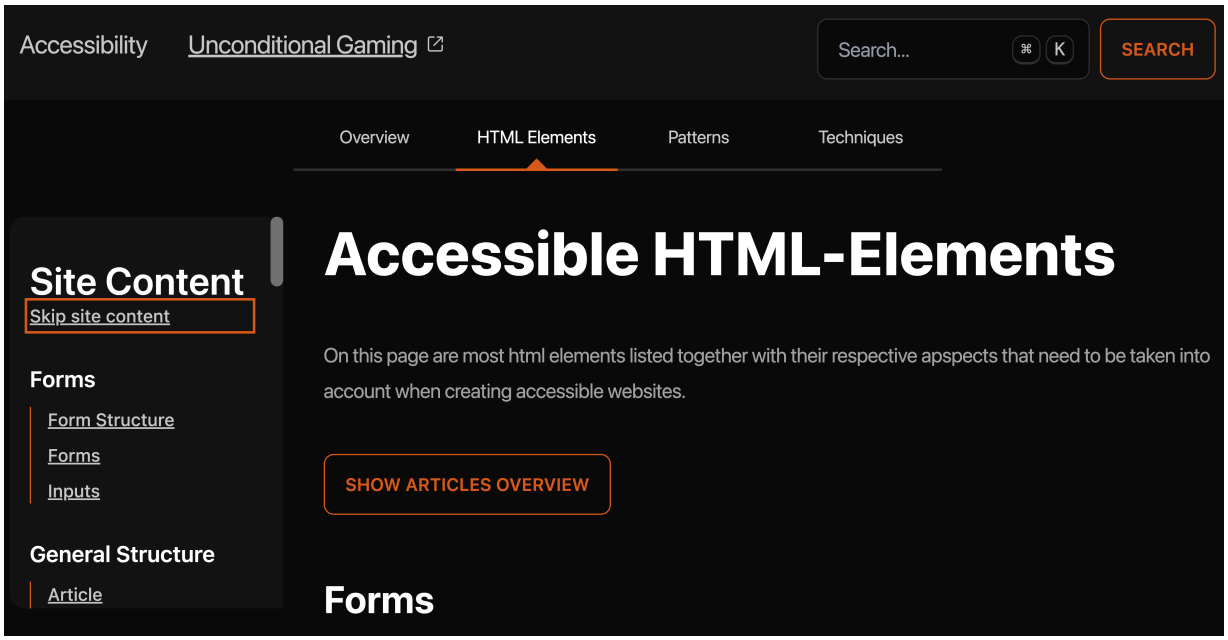


Figure 6-11. A visible skip link in the sidebar on manuelbaisch.com

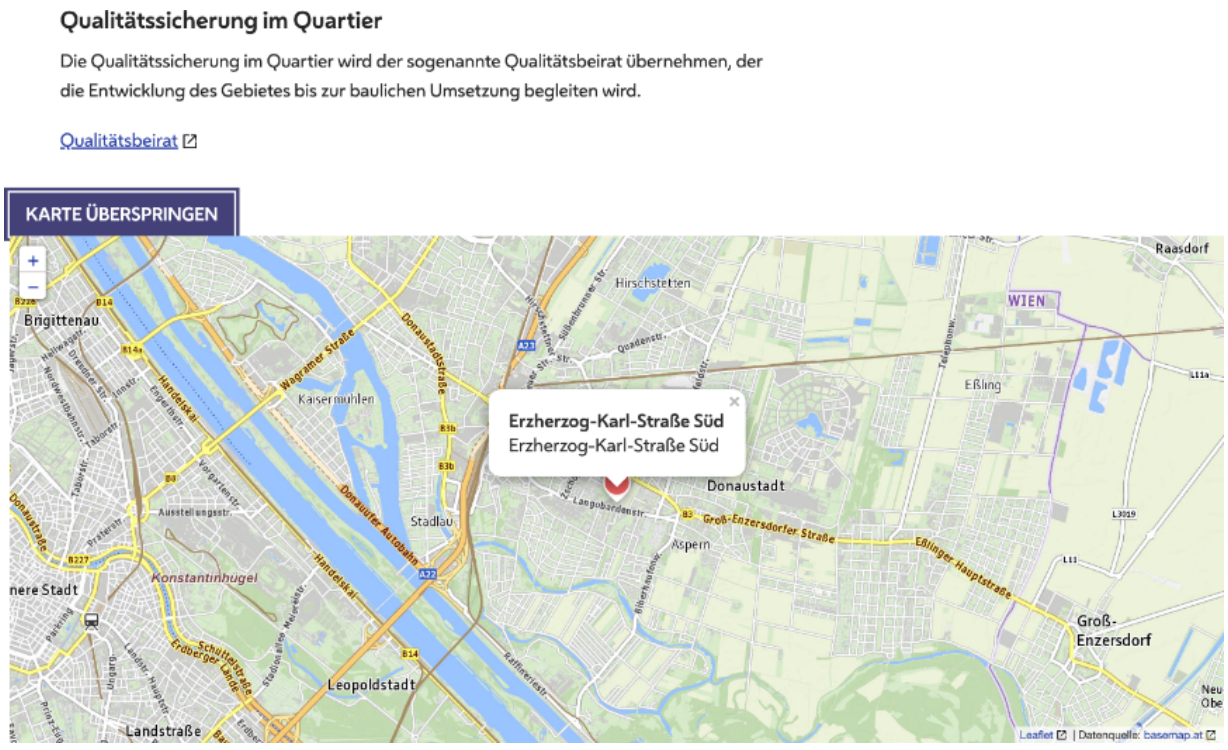


Figure 6-12. The skip link on wien.gv.at/stadtplanung is visible only when you focus it

According to the WebAIM Million 2024 report, 13.3% of tested websites had a skip link. However, one of every six of those links was broken because they were either incorrectly hidden or the target didn't exist. Using `display: none` doesn't qualify for hiding skip links. You have to use an accessible method, as illustrated in Example 6-24. I also recommend you show the skip link on focus, because focusing on a visually hidden link can confuse keyboard users. You can learn more about accessible hiding in Recipe 8.1.

The purpose of skip links is to make keyboard navigation more efficient. Base your decision about which skip links you need on how they can benefit users.

See Also

- "Giving users and developers more control over focus" by Chromium Blog
- Chrome Platform Status: "Use :focus-visible in the default UA style sheet"
- Safari TP 138 Release Notes
- Safari 15.4 Release Notes
- Bugzilla: "Remove focus-visible feature flag"
- "Dialog Focus in Screen Readers" by Adrian Roselli

- “Using JavaScript to trap focus in an element” by Hidde de Vries
- “Skip Navigation Links” by WebAIM

Chapter 7. Navigating Sites

This chapter looks at creating the main navigation for a website. It begins with the simplest possible solution, but with every recipe, you progressively improve the experience and functionality. You learn about each element's and attribute's responsibilities and different approaches to dealing with large, nested hierarchies. Toward the end of this chapter, you will discover why using too many or the wrong ARIA attributes can do more harm than good.

7.1 Create a Main Navigation

Problem

There are many different ways to build the main navigation of a website. Some solutions are over-engineered, while others don't serve users enough functionality and information. HTML and ARIA offer many tools to improve the user experience, but writing semantically poor or too-rich markup can complicate the usage.

Solution

When a website reaches a specific size, we organize pages in larger sections and link them in a central, easily accessible place. These sections can take different shapes; we usually refer to them as navigations or menus.

A website's main navigation can be as simple as a group of hyperlinks, as shown in Example 7-1.

Example 7-1. The simplest possible implementation of a main navigation

```
<a href="/home">Home</a>
<a href="/products">Products</a>
<a href="/team">Team</a>
<a href="/contact">Contact</a>
```

For small or medium-size websites, this simple pattern can be sufficient because it works well for most users, whether they're accessing the site with a mouse, keyboard, or screen reader. The experience should be good enough, but it can be improved using HTML and CSS.

How much functionality you want to add to this basic solution depends on the size and complexity of the navigation (see discussion). Let's look at some possibilities in HTML, CSS, and JavaScript.

HTML

HTML offers users additional ways of accessing the content within a navigation and helps them understand its purpose and structure better, as shown in Example 7-2.

Example 7-2. A semantically richer variation of the basic navigation pattern

```
<nav aria-label="Main">
  <ul>
    <li><a href="/home">Home</a></li>
    <li><a href="/products" aria-current="page">
    <li><a href="/team">Team</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>
```

We'll discuss the benefits of the additions in Example 7-2 and in Recipes 7.2, 7.3, and 7.4.

CSS

The anchor element `<a>` does a great job in terms of the semantic information and functionality it provides (see Chapter 3), but its default styling is too simple. CSS can improve

the readability, increase target sizes, and enhance keyboard navigation.

By increasing the font size and adding a padding to each link, you can expand the interactive area of the element, which makes clicking and tapping links easier and less error-prone (see Example 7-3).

Example 7-3. Larger font size and padding

```
a {
  display: inline-block;
  font-size: 1.4rem;
  padding: 0.3rem;
}
```

You can use the `:focus-visible` pseudoclass to overwrite the default focus styles of links and highlight the currently focused element clearly, as shown in Example 7-4 and Figure 7-1.

Example 7-4. Custom focus styles for links

```
a:focus-visible { ❶
  outline: 0.25em solid currentColor;
  outline-offset: 0.125em;
}
```

- You can learn about the difference between `:focus` and `:focus-visible` in Recipe 6.1.

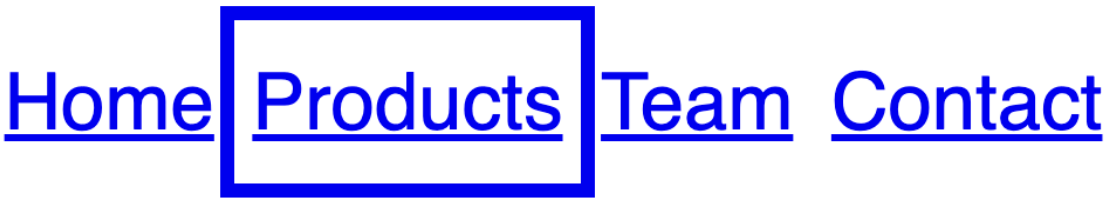


Figure 7-1. A focused link with custom focus styles

JavaScript

JavaScript can help present and structure large navigations, often with nested levels. Enhancing components with JavaScript usually goes hand in hand with using the ARIA specification. ARIA attributes can do a great job helping you create accessible custom JavaScript code, but these attributes are powerful. It's important to use them with caution. You can read more about the dangers of using ARIA in Recipe 7.8.

Discussion

In programming, a design principle called the the Rule of Least Power suggests choosing the least powerful language suitable for a given purpose. For frontend development, this usually means starting with HTML and adding layers of functionality as

needed. This approach makes sense especially for navigation patterns since they take on different shapes, serve various purposes, and can quickly become complex. To guarantee the best possible solution for your users, you have to understand the problems you're solving for them. Copying and pasting ready-made solutions can lead to many issues and confusion for users and developers.

You have to adapt technical implementations to the context in which the navigation is used: for example, small versus large viewport widths. The information hierarchy is another factor. It makes a difference whether you're dealing with 4 or 40 links. You also have to consider different input modalities. Users might access the navigation using a mouse, touch, keyboard, voice, or other means.

Styling

The styling of navigations must ensure that links and buttons are easily accessible, no matter how you access them. That includes the target size of links (more about that topic in Recipe 3.2) and focus styles (Recipe 6.1). Keyboard, switch device, and screen reader users rely on proper focus styling and focus management. Especially in complex navigations with multiple levels, missing or bad focus management can be an issue.

Chapter 6, offers general tips and solutions. The gist of it is that users must be able to access all items in a navigation, easily and quickly.

The recipes in this chapter focus mostly on the technical implementation of main navigations, but there are other things you have to consider when designing and implementing them.

The number of items

Too many options in a navigation can overwhelm users, while too few options can be too vague. The research conducted by cognitive psychologist George A. Miller in his 1956 paper "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" is often used to suggest that there should be no more than nine items in a main navigation. According to Miller's research most people can only keep between five and nine items in their short-term memory.

However, usability specialists from the UX research and consulting firm Nielsen Norman Group (NN/g) assert that this rule doesn't apply to main site navigations because users don't have to remember links in a navigation. They argue that it's more important to design a navigation according to the breadth of your content, to use meaningful and descriptive labels, and

to prioritize links correctly. Their findings are also in line with my personal experience. The navigation on gov.uk (Figure 7-2) contains significantly more than nine items, but it still works great due to the clear labeling and design.

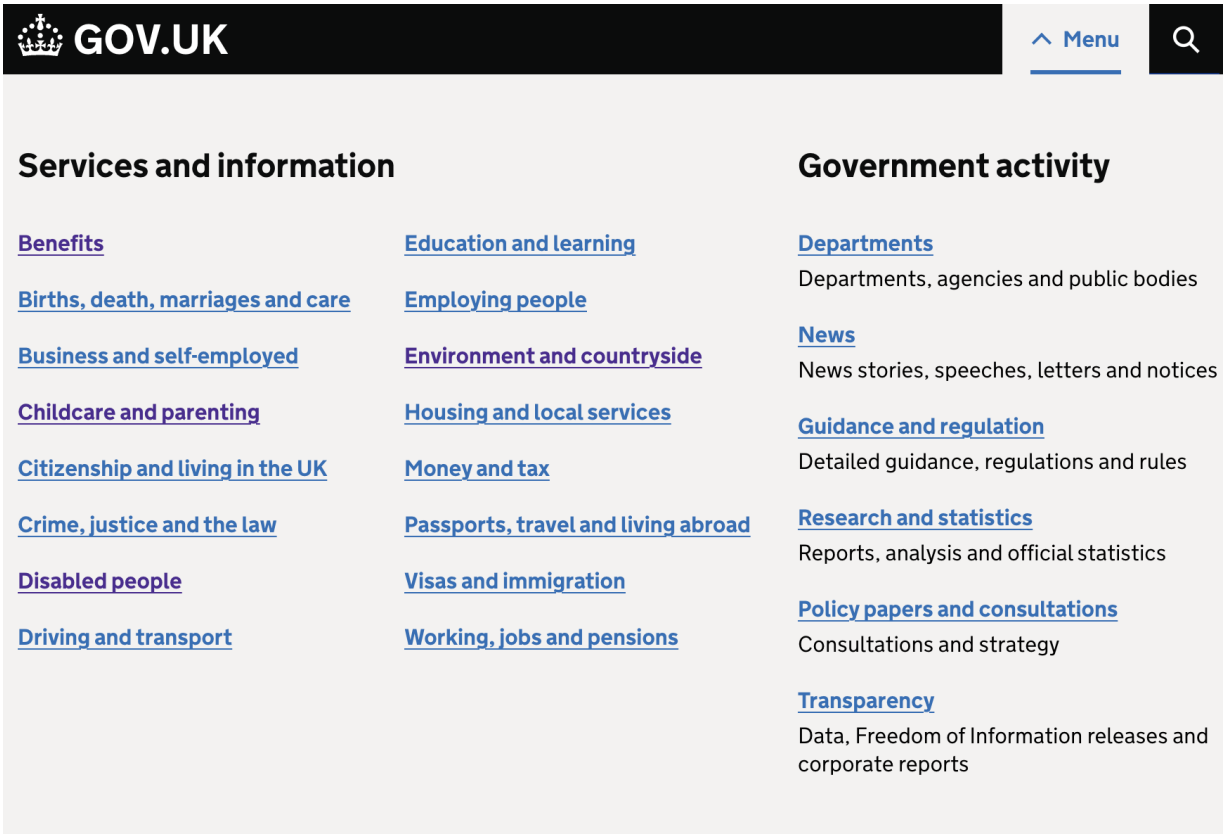


Figure 7-2. More than twenty links in the main navigation of the gov.uk website

Placement

Artistic freedom, creativity, and experimentation are generally welcome in web design. Creating an accessible website doesn't mean you have to make boring designs, but certain parts of a page should follow expected conventions and layouts. The main

navigation is one of them, and it should be placed in predictable areas of the page. This usually means putting it on the top edge of the screen or, for left-to-right languages, on the left edge.

7.2 Highlight the Active Page

Problem

Especially on complex websites with deeply nested structures, it takes time for users to get oriented and obtain an overview of their location within a set of pages. A blind screen reader user might be unable to tell the difference between the active page within the main navigation's list of links and other pages.

Solution

To help users with orientation, you should spotlight the currently active page. Highlighting is not a purely visual task. The active page must communicate its state both visually and semantically.

The `aria-current` attribute represents the current element within a set of related elements. The `page` value limits the

element to the current page within a set of pages, as shown in Example 7-5.

Example 7-5. The "products" link tagged as the active page within a set of links using the `aria-current` attribute

```
<a href="/home">Home</a>
<a href="/products" aria-current="page">Products
<a href="/team">Team</a>
<a href="/contact">Contact</a>
```

A more popular solution is to use the `class` attribute, which conveys which link within the set of related links is currently active, but only visually. A blind screen reader user couldn't tell the difference between the active page and other pages.

Using `class` over the `aria-current` attribute has no obvious advantages because the attribute can be used to style the active pages just as well. As you can see in Example 7-6, I'm using an attribute instead of a class selector to style the active page:

Example 7-6. Highlighting the active page visually using CSS

```
html {
```

```
    --text-color: hsl(0 0% 0%);
    --highlight-color: hsl(209 56% 45%);
  }

  a {
    border-block-end: 3px solid var(--border-color
    color: var(--text-color);
    text-decoration: none;
  }

  [aria-current="page"] {
    --border-color: var(--highlight-color); ❷
    --text-color: var(--highlight-color);
  }
```

❶ Transparent border color by default

❷ Blue border color for the current page

Discussion

The `aria-current` attribute is a great alternative to the more commonly used `class` attribute because it highlights the active link semantically. The attribute can also serve as a selector in CSS that has the same specificity as a class selector.

A link with purely visual highlighting would be announced to screen reader users like all the other links on the page. The ARIA attribute adds the additional information that the selected link is “the current page.”

Testing aria-current="page” with screen readers

Using different screen readers, you’ll get the following information focusing an active link with the keyboard (see Table 7-1).

Table 7-1. What screen readers announce on a link with `aria-current` set to `page`

Screen reader	Announcement
JAWS	Products, link, current page
NVDA	Products, link, current page
VoiceOver iOS	Products, current page, link
VoiceOver macOS	Current page, link, products
TalkBack	Current page, products, link

If you can’t or don’t want to use an attribute selector, you don’t have to. There’s no reason you can’t use both the attribute and

the class together, as shown in Example 7-7.

Example 7-7. A combination using a class for styling and the `aria-current` attribute for semantic highlighting

```
  <a href="/products" class="active" aria-current=
```

Styling

I don't recommend relying purely on semantic highlighting with no styling. Visual highlighting helps users with orientation on the page. This is especially true with deeply nested site structures.

A different text color and decoration visually highlight the "Products" link in Figure 7-3.

Figure 7-3. Products link highlighted visually

You have full artistic freedom with styling links in a main navigation, as long as users understand what you're trying to

achieve, but there are properties in CSS that are more suitable than others.

Color

Relying on color alone to highlight the active link is a poor practice because UIs must work even when color is not available. The page could be presented on a screen that doesn't support colors at all or displays them poorly. Color contrast could be bad because users might have reduced the brightness of their screen or are accessing the site on a mobile device outdoors, with bright sunlight deteriorating visibility. The user might have a type of low vision that affects their perception of color. There are many reasons to support color highlighting with different styling.

Relying on color alone can be problematic, as Figure 7-4 illustrates.

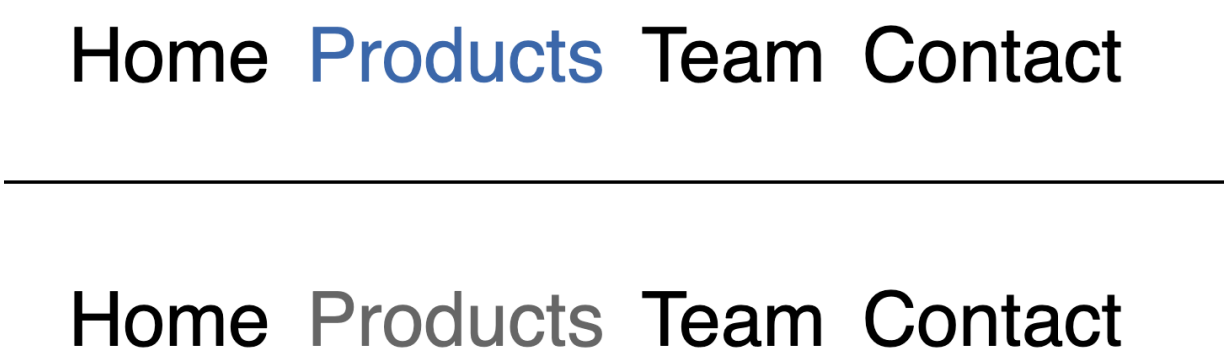


Figure 7-4. Highlighting the current page with color alone: the first example shows blue text color for the “Products” link; the second example emulates achromatopsia showing the current page in a lighter gray color

Box-shadow

`box-shadow` is often used instead of `border` to create similar effects as seen in Figure 7-5 because it’s easier to work with since it doesn’t affect the width or height of the element and it allows authors to create nested borders. A disadvantage is that it doesn’t work well with forced-color mode.

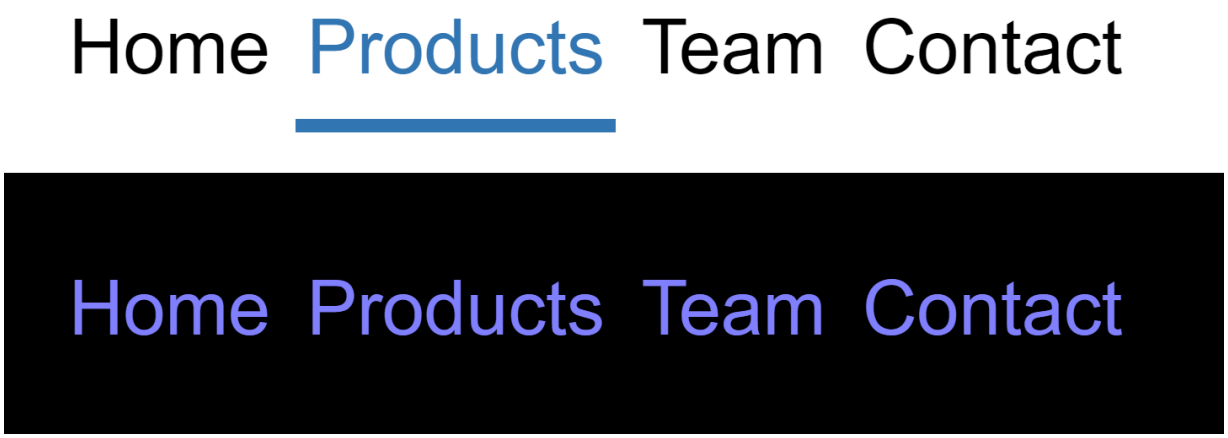


Figure 7-5. `box-shadow` used instead of border to highlight a link: the first example shows Windows with its default theme and the second example a dark high contrasting theme

7.3 Announce the Number of Items

Problem

Complex navigations can be hard to use and understand. When a sighted user looks at a navigation or interacts with it by clicking or hovering over some links or buttons, they can usually tell how complex and large it is. A blind screen reader user can't obtain this information as quickly.

Solution

Lists in HTML provide additional semantic information that tells users how many items exist in a list and which item in a set

of items the user is currently accessing (see Example 7-8).

Example 7-8. Each link is wrapped in a list item as part of an unordered list

```
<ul role="list">
  <li>
    <a href="/home">Home</a>
  </li>
  <li>
    <a href="/products" aria-current="page">Prod
  </li>
  <li>
    <a href="/team">Team</a>
  </li>
  <li>
    <a href="/contact">Contact</a>
  </li>
</ul>
```

The additional HTML also comes with different styling you must adapt to make the list look like it did before. In Example 7-9, you can see that you have to remove the list's default margin, padding, and list styling and you must turn the list into a flex container to align the items horizontally.

Example 7-9. Turning a vertical list of items into a horizontal list with no bullets

```
ul {
  display: flex;
  gap: 1rem;
  list-style: none;
  margin: 0;
  padding: 0;
}
```

Discussion

When a website reaches a certain size, it usually also offers different ways of accessing content. It might have a home page with teasers and call-to-action links, overview pages, a search widget and a search page, and one or more navigations. How users find the information they're looking for depends on preference and the available options.

The main navigation can be an excellent tool for accessing important pages quickly, but if it's too complex, it can be hard for some users to navigate and get oriented. Complexity is affected by the technical implementation, the structure, and by the number of links.

Just by looking or quickly interacting with a navigation, sighted users can usually tell with little effort how many links it contains and how it's structured. A blind screen reader user can't get this information as quickly. They may have to work their way through the entire list of links, which might not be an issue if the size of the list is manageable, but if it contains, for example, 30 links, this task can be cumbersome. If a screen reader user knows up front that the navigation contains a lot of options, they might decide to use a different, more efficient way of navigation.

Wrapping links in a list and list items seems like an unnecessary addition to the code that only increases the size and complexity. The proposed solution in this recipe even makes sure that the navigation looks like it did before the additional markup. That appears to be redundant, but there are many advantages to using lists. You can learn about them in Recipe 2.4.

Ordered versus unordered lists

You might wonder why I recommend using an unordered list `<ul>` and not an ordered list `<ol>`. The order of items in a navigation is usually not random. Teams put a lot of thought into which items to add to the list of most important pages.

They place them specifically, with the most relevant items at the beginning or end of the list. This means that the order matters, so it makes sense to use an ordered list. The thing is, that from a semantic perspective, the order is important only to the people who add and arrange the items. What matters to the *user* is that links have meaningful labels and that they can access them quickly.

There is a difference between the significance for you conceptually and for the user semantically. If you change the order of items in the navigation, it doesn't affect meaning, but that's not always the case. For example, if you're listing the steps needed to cook a dish in a recipe, the order is important to you and the user. It makes a difference whether they peel and cut an onion first and then put it in the pan, or the other way around (see Example 7-10).

Example 7-10. A list with items in an explicit order

```
<h1>Onion tart recipe</h1>
<ol>
  <li>Peel the onions.</li>
  <li>Slice them into small pieces.</li>
  <li>Put them in a frying pan with some oil.</l

  <li>…</li>
</ol>
```

That being said, it doesn't make a big difference whether you use an `<ol>` or `<ul>` in a navigation.

The explicit list role

You might have noticed that the solution in this chapter uses an ARIA role on the list element (see Example 7-11).

NOTE

The `role` attribute allows you to change the role of a semantic element or add one to a generic element, for example, `<div role="region"></div>`.

Example 7-11. An explicit list role on a `<ul>` that already has an implicit list role

```
<ul role="list">
        …
</ul>
```

If you validate this code, the validation service reports a warning like *"The `list` role is unnecessary for element `ul`,"* as shown in Figure 7-6. It's redundant to define the role of the `<ul>` explicitly, since it already has this role implicitly.

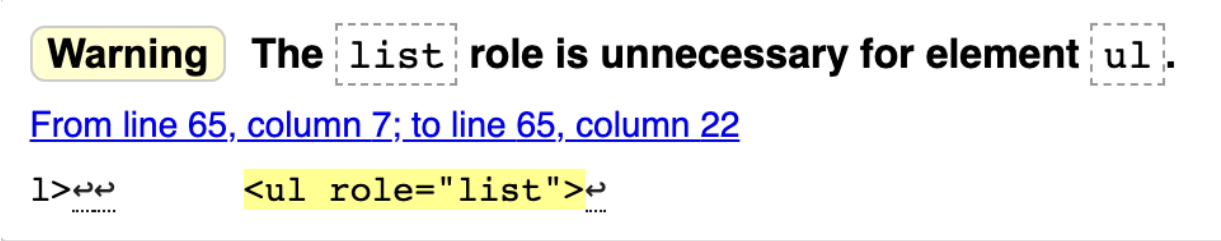


Figure 7-6. Warning from the W3C Markup Validation Service

You can safely ignore this warning because the attribute is there for a reason. If you set `list-style: none` or use any other property that removes the visual list indicators, you may lose all the advantages of using lists in some screen readers. Instead of "list, 4 items," the software may not announce the list as a list (the links are unaffected by that). In VoiceOver on Safari, this is by design. The WebKit team decided to remove list semantics when a list doesn't look like a list. Their reasoning is that if a sighted user doesn't need to know it's a list, a screen reader user doesn't need or want to know either. The explicit `list` role brings back the semantic information.

An alternative to combining `list-style: none` with `role="list"` is setting `list-style-type: ""`, which yields the same result visually but doesn't remove the semantic information of the list in Safari with VoiceOver (see Example 7-12). That works only if you don't use any other property that also removes the default list styling. For example, setting `display: flex` on list items removes the list's bullets and the

semantic information. The most robust way of maintaining the semantics in VoiceOver is using the explicit `list` role.

Example 7-12. Using `list-style-type: ""` instead of `list-style: none` to maintain semantic information of lists in VoiceOver on Safari

```
ul {
  display: flex;
  gap: 1rem;
  list-style-type: "";
  margin: 0;
  padding: 0;
}
```

Depending on the complexity of your navigation, this lack of announcement may or may not be an issue. On one hand, the navigation is still usable, and it affects only VoiceOver in Safari. VoiceOver with Chrome or Firefox still announces the number of items, as do other screen readers, like NVDA. On the other hand, the semantic information could be really useful in some situations.

To make that decision, test the navigation with actual screen reader users and get their feedback. If you decide you need VoiceOver in Safari to behave like all the other screen readers,

using the explicit role on the `<ul>` reverts the behavior to the state before we've removed the list styling. Visually, the list still looks the same.

7.4 Provide Quick Access

Problem

The main navigation is usually located in the page's header. Besides the navigation, there might be a search widget, additional secondary navigations, a language selection, ads, drop-downs, buttons, and more. For users of assistive technology, it can be frustrating to always have to tab through all of these items until they finally reach the main navigation.

Solution

There are two solutions to this problem, which don't have to be used exclusively.

Landmarks

You can turn an ordinary list into a navigational list by wrapping the `<ul>` in a `<nav>` element. This adds useful semantic information to the navigation, and it allows screen

reader users to jump directly into the main navigation using shortcuts, as shown in Example 7-13. The `aria-label` attribute labels the navigation.

Example 7-13. A labeled navigation landmark

```
<nav aria-label="Main">
  <ul>
    <li>
      <a href="/home">Home</a>
    </li>
    <li>
      <a href="/products" aria-current="page">Pr
    </li>
    <li>
      <a href="/team">Team</a>
    </li>
    <li>
      <a href="/contact">Contact</a>
    </li>
  </ul>
</nav>
```

Skip links

If you put a skip link at the beginning of the page, as early as possible in the DOM, users can skip any interactive elements that come before the navigation and interact with the navigation directly. Examples 7-14 and 7-15 illustrate how to implement a skip link in HTML and CSS.

Example 7-14. Implementing a skip link in CSS

```
html {
  --text-color: hsl(0deg 0% 0%);
  --text-color-light: hsl(0deg 0% 100%);
  --highlight-color: hsl(209deg 56% 45%);
}

.skip-link:is(:link, :visited) { ❶
  background-color: var(--highlight-color);
  color: var(--text-color-light);
  padding: 0.5rem;
  position: absolute;
  text-decoration: none;
}

.skip-link:not(:focus-visible):not(:active) { ❷
  clip-path: inset(50%);
  height: 1px;
  overflow: hidden;
  width: 1px;
  white-space: nowrap;
```

```
}
```

❶ Basic styling for the skip link.

❷ Hide the link visually if it's not focused.

The `.skip-link` class makes sure that the link is visible only when it's needed. Use the `href` attribute to enable jumping directly to an anchor you put on the `<nav>` using the `id` attribute.

Example 7-15. A skip link in a `<header>`

```
<header>
  <a class="skip-link" href="#main-nav">Jump to

  <a href="/">My website</a> ❷
  <a href="/">Some link</a>
  <a href="/">Another link</a>
  <button>A button</button>
  <button>Another button</button>

  <nav id="main-nav" aria-label="Main"> ❸
    <ul id="main-nav-list">
      <li>
        <a href="/home">Home</a>
      </li>
```

```
        <li>
          <a href="/products" aria-current="page">
        </li>
        <li>
          <a href="/team">Team</a>
        </li>
        <li>
          <a href="/contact">Contact</a>
        </li>
      </ul>
    </nav>
  </header>
```

1. Skip link at the very beginning of the page
2. Some random exemplary interactive elements that come before the main navigation
3. Labeled navigation landmark with an ID

Discussion

Landmarks

Wrapping links within a navigation in a list element improves the user experience, but it's still just an ordinary list. To give it a

special meaning, wrap it in a `<nav>` element.

Using the `<nav>` element has several advantages. Notably, a screen reader announces something like “navigation” when the user interacts with it. It also adds a *landmark* to the page: a special region like `<header>`, `<footer>`, or `<main>`, that screen reader users can access using shortcuts. In VoiceOver on macOS you can use a shortcut to list all landmarks, as shown in Figure 7-7.

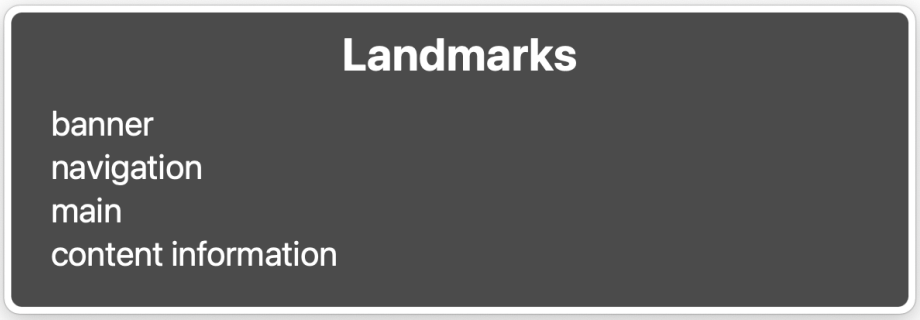


Figure 7-7. Rotor in VoiceOver listing all the landmarks on a page

If you have multiple navigations of the same type, you should label them using `aria-labelledby` or `aria-label` to make them distinguishable, as shown in Figure 7-8. You can learn

more about landmarks in Recipe 1.5 and labeling them in Recipe 2.3.

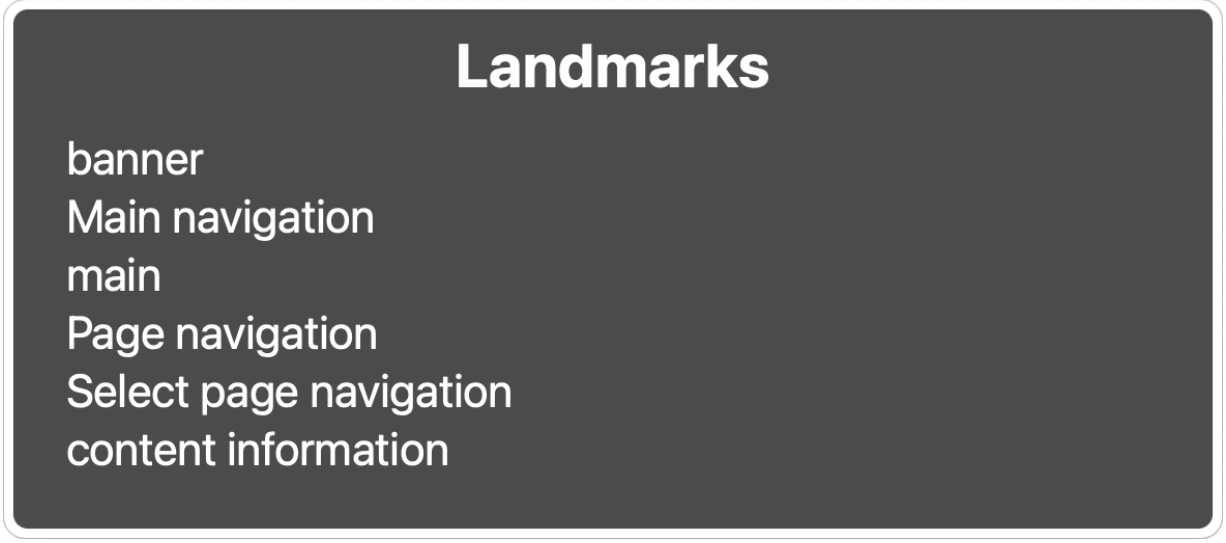


Figure 7-8. VoiceOver listing labeled and unlabeled landmarks

Another interesting detail about wrapping the list of links in a `<nav>` element is that it automatically brings back the native semantics of the list, even if it doesn't look like a list, as Scott O'Hara notes in "Fixing Lists". Therefore, you don't need to add an explicit `list` role to the `<ul>` element, like you did in Example 7-11.

Skip links

In most cases, websites that offer skip links (see Recipe 6.6) provide users with at least one skip link to the main content of the page. Depending on the complexity of the `<header>` or

wherever the navigation is located, it might be useful to offer an additional skip link that allows users to skip to the navigation.

Figure 7-9 shows that users must cycle through 15 tab stops before reaching the main navigation on *nytimes.com* if they don't use the skip link at the beginning of the page.

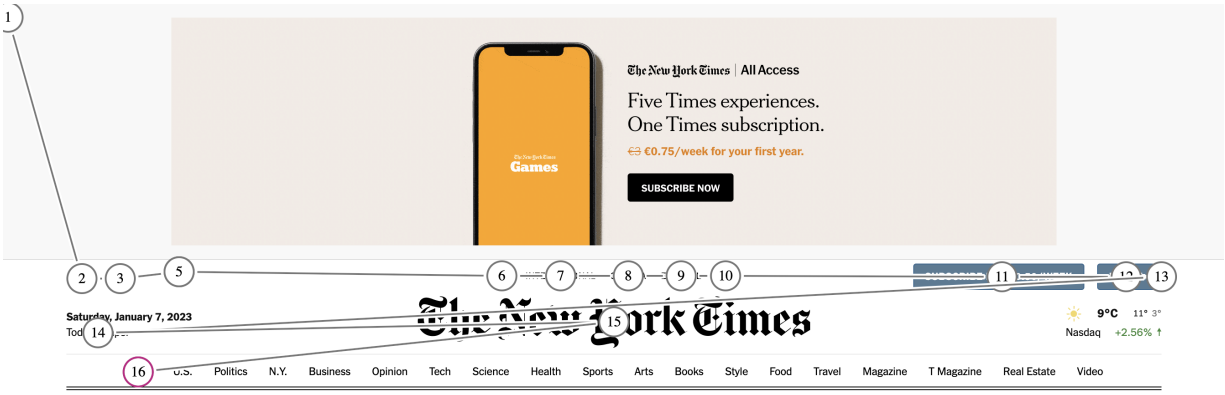


Figure 7-9. The encircled numbers mark each tab stop before the main navigation (number 1 is the visually hidden skip link)

A skip link is hidden by default and becomes visible only if the user focuses the link. The skip link in Figure 7-10 is present but not visible.

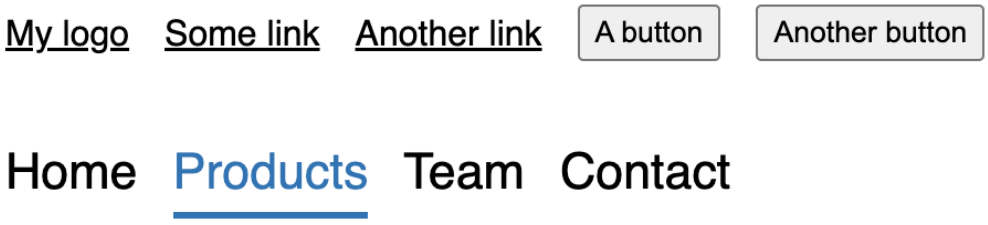


Figure 7-10. Header with a couple of interactive elements followed by the main navigation

The skip link pops up on focus and overlays the rest of the page without interfering with the layout, as shown in Figure 7-11.

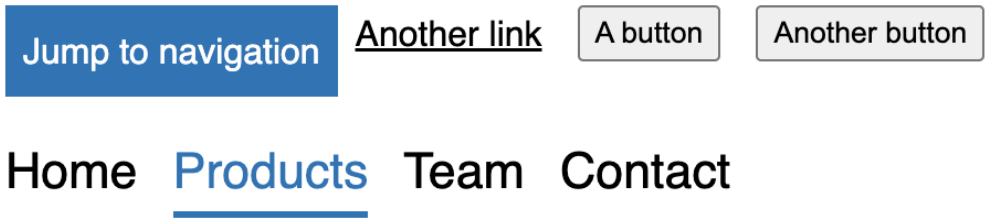


Figure 7-11. A “Jump to navigation” skip link

Whether it makes sense to add a “Jump to navigation“ skip link or not depends on the number of interactive elements that come before the navigation. If there’s only a single link before the navigation, a skip link would increase the number of tabs required to get to the navigation. If there are more than just one or two interactive elements, a skip link will probably be useful. You can learn more about skip links in Recipe 6.6.

7.5 Hide the Navigation on Narrow Viewports

Problem

When your navigation reaches a certain size in terms of the number of items, there's likely not enough space on the screen to display all of them without affecting UX negatively.

Solution

Hiding the list of links and allowing users to open and close can be a solution.

To hide the navigation on narrow viewports, you have to do three things: adapt the styling for narrow viewports, hide the list in a sidebar, and allow users to toggle its visibility.

CSS

You must ensure the layout works well on narrow screens and the list is hidden by default.

```
nav {
  position: var(--nav-position, fixed); ❶
```

```
  inset-block-start: 1rem; ❷
  inset-inline-end: 1rem; ❸
}

nav ul { ❹
  background: hsl(0 0% 100%);
  box-shadow: var(--nav-list-shadow, -5px 0 11px
  display: flex;
  flex-direction: var(--nav-list-layout, column)
  flex-wrap: wrap;
  gap: 1rem;
  height: var(--nav-list-height, 100dvh);
  list-style: none;
  margin: 0;
  padding: var(--nav-list-padding, 2rem);
  position: var(--nav-list-position, fixed);
  inset-block-start: 0;
  inset-inline-end: 0;
  width: var(--nav-list-width, min(22rem, 100vw)
  visibility: var(--nav-list-visibility, hidden)
}

[aria-expanded="true"] + ul {
  --nav-list-visibility: visible;  ❻
}

@media (min-width: 48em) {
  nav {
```

```
    --nav-position: static; ❼
    --nav-button-display: none; ❽
  }

  nav ul { ❾
    --nav-list-layout: row;
    --nav-list-position: static;
    --nav-list-padding: 0;
    --nav-list-height: auto;
    --nav-list-width: 100%;
    --nav-list-shadow: none;
    --nav-list-visibility: visible;
  }
}

nav ul:first-child { ❿
  --nav-list-layout: row;
  --nav-list-position: static;
  --nav-list-padding: 0;
  --nav-list-height: auto;
  --nav-list-width: 100%;
  --nav-list-shadow: none;
  --nav-list-visibility: visible;
}

nav button {
  all: unset; ⓫
  display: var(--nav-button-display, flex);
```

```
    position: relative;
    z-index: 1;
  }

  nav button:focus-visible { ❷
    outline: 4px solid currentColor;
    outline-offset: 2px;
  }
```

❶ Fixed position for the nav by default.

❷ Logical property. Equivalent to top: 0.

❸ Logical property. Equivalent to right: 0.

❹ The list looks like a sidebar on narrow viewports.

❺ The list is hidden by default.

❻ Show the list when the navigation's burger button's `aria-expanded` attribute is set to true.

❼ Position the navigation static.

❽ Hide the burger button.

❾

Adapt the list's styling so that it doesn't look like a list anymore.

11. Repeat the settings from the previous steps to provide a different styling when JavaScript is disabled. The selector assumes that the button doesn't exist without JS, making the list the first child within the navigation.

12. Reset button styles.

13. Bring back unset focus styles for the button.

HTML

You can store the markup for the toggle button in a template element.

```
<nav aria-label="Main" id="main-nav">
  <ul id="main-nav-list">
    <li>
      <a href="/home">Home</a>
    </li>
    <li>
      <a href="/products" aria-current="page">Pr
    </li>
    <li>
      <a href="/team">Team</a>
```

```
      </li>
      <li>
        <a href="/contact">Contact</a>
      </li>
    </ul>

    <template id="burger-template">
      <button type="button" aria-expanded="false" a
        aria-controls="main-nav-list">
        <svg viewBox="-5 0 10 8" width="40" aria-h
          <line y2="6.5" stroke="#000" stroke-widt
        </svg>
      </button>
    </template>
  </nav>
```

JavaScript

Clone the button and add event listeners to the navigation.

```
  const nav = document.querySelector("nav");
  const list = nav.querySelector("ul");
  const burgerTemplate = document.querySelector("#
  const burgerClone = burgerTemplate.content.clone
  const button = burgerClone.querySelector("button

  button.addEventListener("click", (e) => { ❶
```

```
    const isOpen = button.getAttribute("aria-expan
    button.setAttribute("aria-expanded", !isOpen);
  });

  nav.addEventListener("keyup", (e) => { ❷
    if (e.code === "Escape") {
      button.setAttribute("aria-expanded", false);
      button.focus();
    }
  });

  nav.insertBefore(burgerClone, list); ❸
```

❶ Toggle `aria-expanded` attribute. `aria-expanded="true"` signals that the menu is currently open.

❷ Hide list on keydown Escape.

❸ Add the button to the page.

Discussion

That was a lot of code, so let's break it down.

Styling

First, remove the `<nav>` from the natural flow of the page and place it at the top end corner of the viewport, as shown in Example 7-16.

Example 7-16. Removing the nav from the natural flow of the page on narrow viewports

```
@media (min-width: 48em) {
  nav {
    --nav-position: static;
  }
}

nav {
  inset-block-start: 1rem;
  inset-inline-end: 1rem;
  position: var(--nav-position, fixed);
  z-index: 1;
}
```

NOTE

In CSS, `inset-block-start` and `inset-inline-end` are the logical equivalent of the physical properties `top` and `right`.

Next, change the layout of the list on narrow viewports by adding a new custom property `(—-nav-list-layout)`. The layout is `column` by default and switches to `row` on larger screens, as shown in Example 7-17 and Figure 7-12.

Example 7-17. Display the list in a vertical or horizontal layout depending on the viewport width

```
ul {
  display: flex;
  flex-direction: var(--nav-list-layout, column)
  flex-wrap: wrap;
  gap: 1rem;
  list-style: none;
  margin: 0;
  padding: 0;
}

@media (min-width: 48em) {
  ul {
    --nav-list-layout: row;
  }
}
```

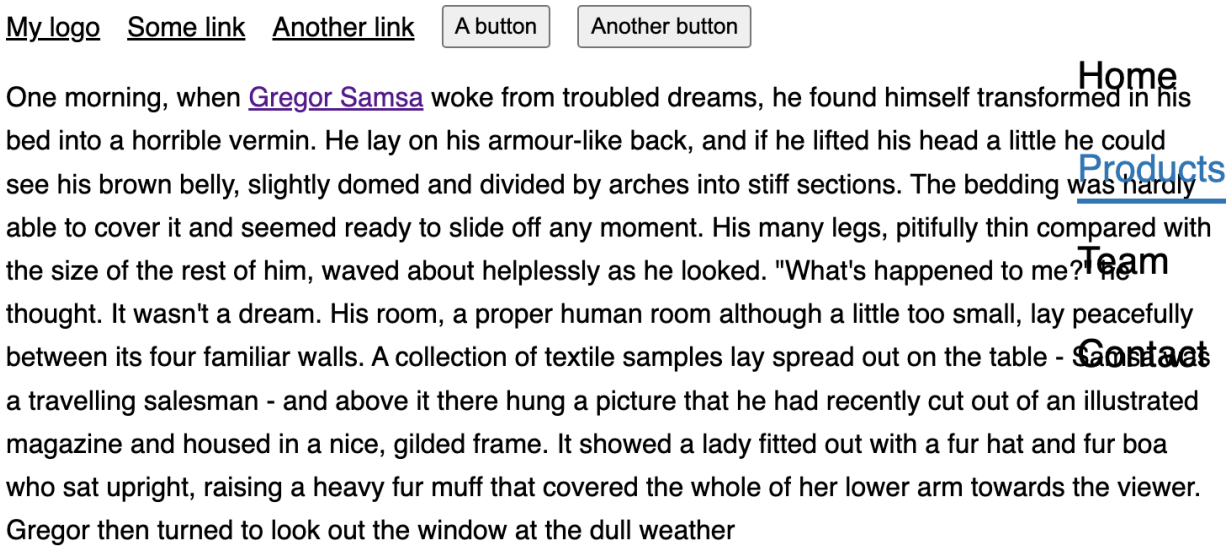


Figure 7-12. Work in progress: the list removed from the document flow and placed above the rest of the content

You've removed the list from the document flow and you've changed its layout, but it obviously needs more styling to look like a proper sidebar. Move it up to the top end corner, make it fill the whole screen vertically, add a `max-width` and some `padding`, and apply a `background-color` and a `box-shadow`, as shown in Example 7-18.

Example 7-18. Styling the list to look like a sidebar on narrow viewports

```
ul {
  background: hsl(0 0% 100%);
  box-shadow: var(--nav-list-shadow, -5px 0 11px
  display: flex;
  flex-direction: var(--nav-list-layout, column)
```

```
    flex-wrap: wrap;
    gap: 1rem;
    height: var(--nav-list-height, 100dvh);
    list-style: none;
    margin: 0;
    padding: var(--nav-list-padding, 2rem);
    position: var(--nav-list-position, fixed);
    inset-block-start: 0;
    inset-inline-end: 0;
    width: var(--nav-list-width, min(22rem, 100vw)
  }

  @media (min-width: 48em) {
    ul {
      --nav-list-layout: row;
      --nav-list-position: static;
      --nav-list-padding: 0;
      --nav-list-height: auto;
      --nav-list-width: 100%;
      --nav-list-shadow: none;
    }
  }
```

The list should look something like Figure 7-13 on narrow viewports, more like a sidebar than a simple list.

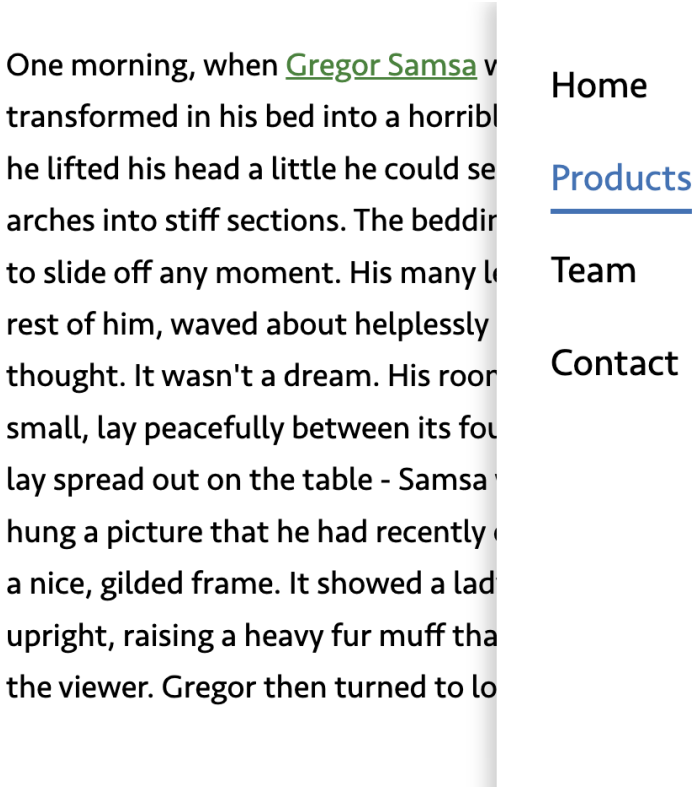


Figure 7-13. Styled list on narrow viewports

Hiding the list

Next, you have to hide the list, and *just* the list, as shown in Example 7-19. That's important, because hiding the entire navigation would also mean hiding an important landmark. There are several ways in CSS to hide content, but not all of them are accessible.

It's important that you use a property declaration like `visibility: hidden` or `display: none` instead of `opacity: 0` or `translateX(100%)`. These properties make sure that the links are not focusable when the navigation is

hidden. Using `opacity` or `translate` will remove content visually, so the links would be invisible yet still accessible using the keyboard, which would be confusing and frustrating. Using `visibility` or `display` hides the list visually and makes it unreachable to all users.

You can learn more about hiding content in Chapter 8.

Example 7-19. Hiding the list on narrow viewports

```
ul {
  visibility: var(--nav-list-visibility, hidden)
}

@media (min-width: 48em) {
  ul {
    --nav-list-visibility: visible;
  }
}
```

Toggling the visibility of the list

You want to give users the ability to show and hide the list whenever they want. The first thing you need is a button. There are different ways of adding it to the DOM; using a template in

HTML and cloning it in JavaScript is one of them, as shown in Example 7-20.

Example 7-20. Using the template element to prepare markup in HTML for later usage in JavaScript

```
<nav id="main-nav">
  <ul id="main-nav-list">…</ul>

  <template id="burger-template"> ❶
      <button
        type="button"
        aria-expanded="false" ❷
        aria-label="Menu" ❸
        aria-controls="main-nav-list" ❹
      >
        <svg viewBox="-5 0 10 8" width="40" aria
        <line y2="6.5" stroke="#000" stroke-widt
      </svg>
    </button>
  </template>
</nav>
```

There are a lot of important features in this short code snippet.

- ❶ The entire markup for the button is in a template so I don't have to create it in JavaScript. It's easier to work

with and read.

❷ The `aria-expanded` attribute tells assistive technology if the element the button controls is expanded.

❸ `aria-label` gives the button an "accessible name," a text alternative for the menu icon.

❹ `aria-controls` creates a reference to the list.

❺ Hide the `<svg>` from assistive technology using `aria-hidden` because it already has a text label provided by `aria-label`.

The button needs some basic styling and you want to make sure that you show it only on narrow screens, as shown in Example 7-21.

Example 7-21. Resetting the default button styling and showing it on only narrow viewports

```
@media (min-width: 48em) {
  nav {
    --nav-button-display: none;
  }
}
```

```
button {
  all: unset;
  display: var(--nav-button-display, flex);
  position: relative;
  z-index: 1;
}

button:focus-visible {
  outline: 0.25em solid currentColor;
  outline-offset: 0.125em;
}
```

Next, select and clone the template and query the button, as shown in Example 7-22.

Example 7-22. Query the template and clone it

```
const nav = document.querySelector('nav')
const list = nav.querySelector('ul');
const burgerTemplate = document.querySelector('#
const burgerClone = burgerTemplate.cloneNode(tru
const button = burgerClone.querySelector('button
```

When the user clicks the button or presses `Enter` or `Space`, toggle the value of the `aria-expanded` attribute, indicating whether the list is expanded or not, as shown in Example 7-23.

Example 7-23. Click event on the button that toggles the value of the `aria-expanded` attribute

```
button.addEventListener('click', e => {
  const isOpen = button.getAttribute('aria-expan
  button.setAttribute('aria-expanded', !isOpen);
});
```

Optionally, you can also allow users to close the navigation by pressing `Escape`. It's convenient for users to have the ability to close the navigation whenever they want, as shown in Example 7-24.

Example 7-24. Keyup event on the navigation that closes it when the user presses the Escape key

```
nav.addEventListener('keyup', e => {
  if (e.code === 'Escape') {
    button.setAttribute('aria-expanded', false);
    button.focus();
  }
});
```

Next, attach the button to the navigation, as shown in Example 7-25. It's critical to use `insertBefore` instead of

`appendChild` because the button should be the first element in the navigation. If a keyboard or screen reader user presses `Tab` after clicking the button, they expect to focus the first item in the list. If the button comes after the list, that would not be the case.

Example 7-25. Attaching the button at the beginning of the navigation, before the list

```
  nav.insertBefore(burgerClone, list);
```

As you can see, the JavaScript code doesn't affect the visibility of the list directly. You're only toggling the `aria-expanded` attribute on the button. Since the `<ul>` is a direct sibling of the `<button>`, you can use this information to toggle the visibility in CSS, as shown in Example 7-26.

Example 7-26. Using an attribute selector in CSS to toggle the visiblity of the list

```
ul {
  visibility: var(--nav-list-visibility, hidden)
}

[aria-expanded="true"] + ul {
```

```
    --nav-list-visibility: visible;
  }
```

Figure 7-14 shows the navigation with the menu button.

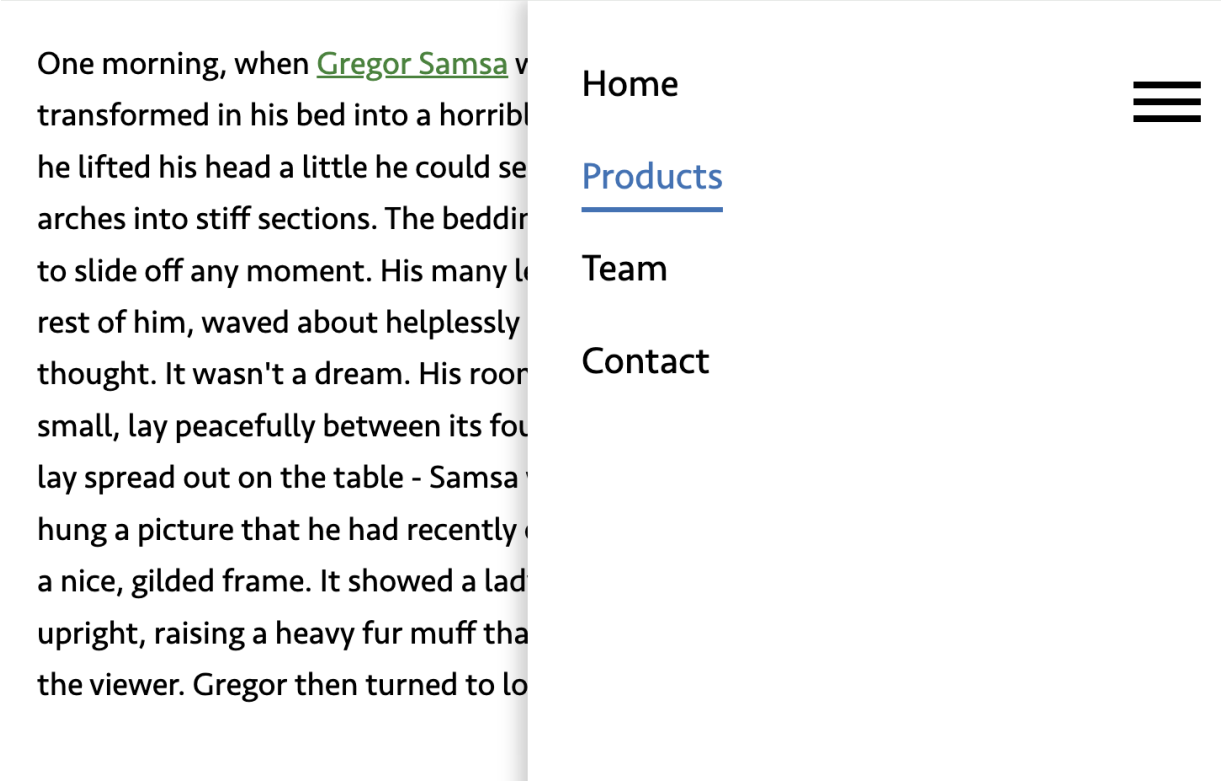


Figure 7-14. List with a burger button on narrow viewports

To burger or not to burger

When you're hiding the navigation "behind a burger," you're adding complexity. Users might not recognize the icon, might miss the button, or might not understand the pattern at all. Only hide content or adapt a conventional pattern as a last

resort. Prefer displaying navigations directly, even on narrow viewports.

> *Discoverability is cut almost in half by hiding a website's main navigation. Also, task time is longer and perceived task difficulty increases.*
>
> —Kara Pernice and Raluca Budiu of NN/g

The burger icon itself can cause issues, too. Users might not understand its purpose. Combining the icon with a visual label like "Menu" can make the solution more accessible for people less familiar with the icon.

Narrow viewports versus narrow screens

You might have noticed that this chapter refers to narrow *viewports* and not narrow *screens* or *smartphones*. That might seem like a tiny detail, but it's essential for how we approach design, development, and testing.

Most modern websites are responsive, which means that the layout adapts to the width of the viewport. The width of the viewport often equals the width of the screen, for example, on a smartphone or a desktop browser in full-screen mode, but there are exceptions.

Some browsers have sidebars

In Safari, for example, bookmarks are located in a sidebar. Another example is the Arc browser, which has no UI on the top edge of the screen. Most of the interaction happens in a sidebar on the left.

Some browsers support screen splitting

The Arc browser allows splitting the screen in two or more separate viewports, as shown in Figure 7-15.

Users don't always surf full-screen

They might organize their windows in a way that allows them to use two or more applications at the same time.

Zoom affects layout

People with low vision, who need to enlarge text and read it in a single column, might zoom the page up to 500%. This causes the content to reflow, which means that media queries optimized for narrow viewports are applied to large viewports.

This is important because many people have the misconception that "narrow screen" means "mobile screen" and that "mobile" means "touch only." Someone might use a keyboard or switch

device on a page that is zoomed up to 400%. Our layout must be accessible with all input modalities, no matter the width of the viewport.

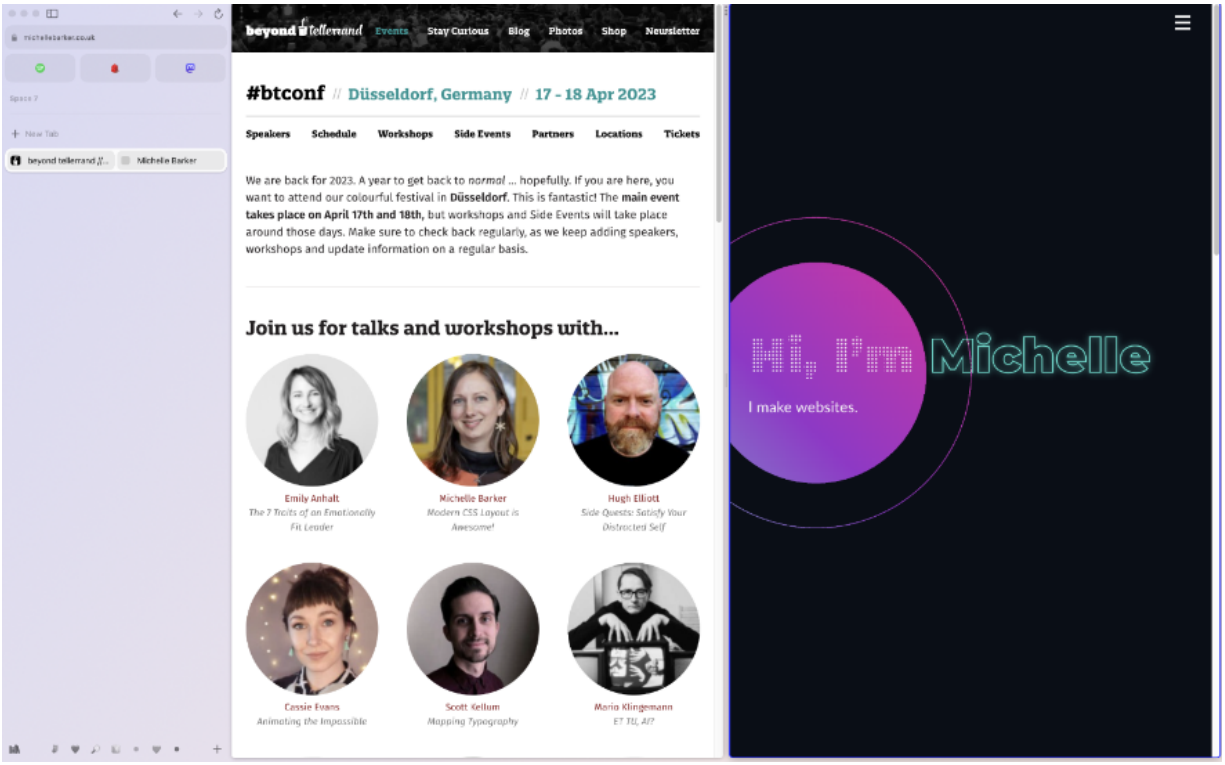


Figure 7-15. The Arc browser with the sidebar open and two websites in split-screen

Progressive enhancement

Web accessibility is not just about users' cognitive or physical abilities; their technology is also a factor. This includes network speeds, capabilities of the device used, and constraints within the browser, operating system, or network.

A controversial topic in web development is whether websites must be operable with JavaScript turned off. I won't go into detail here, but for a website to be truly accessible, it must adapt to its environment and function even under difficult conditions. That's why the solution in this recipe builds upon a pattern that doesn't rely on client-side scripting and that adds functionality and UI elements only needed when JavaScript is available.

If you're working with JavaScript frameworks in an environment that relies on client-side scripting, for example, you can adapt this recipe to render UI elements and attributes directly in the component without adding them dynamically.

7.6 Add a Slide-in Animation

Problem

Animations on the web are sometimes annoying, but they may also cause nausea, dizziness, and headaches in some users. The vestibular system is the mechanism in the inner ear that provides an internal sensor to communicate our body's physical position and orientation in the world. It processes the sensory information involved with controlling balance and eye

movements. For people with vestibular disorders, animation can cause pain and make them feel so bad that they have to stop using the computer, needing time to recover.

Solution

When you add animation to a web page, you must respect user preferences. Create a subtle fade-in animation first for users who prefer reduced motion, then add a more dynamic animation for users with no preference.

The baseline is a simple fade-in and -out animation, as shown in Example 7-27.

Example 7-27. The list fades in and out when the user clicks the button

```
ul {
  opacity: 0;
  transition: opacity 0.3s ease-in-out, visibili
  visibility: var(--nav-list-visibility, hidden)
}

[aria-expanded="true"] + ul {
  --nav-list-visibility: visible;
  opacity: 1;
}
```

Use the `prefers-reduced-motion` media feature to detect whether users are okay with more motion on the screen, so it will replace the fading with a sliding animation, as shown in Example 7-28.

Example 7-28. On top of the fade-in and -out animation, the list moves into the screen for users with no preference for reduced motion

```
ul {
  opacity: 0;
  transition: opacity 0.3s ease-in-out, visibili
  visibility: var(--nav-list-visibility, hidden)
}

@media (prefers-reduced-motion: no-preference) {
  ul {
    opacity: 1;
    transform: var(--nav-list-transform, transla
    transition: transform 0.6s cubic-bezier(.68,
      visibility 0.3s linear;
  }
}

[aria-expanded="true"] + ul {
  --nav-list-visibility: visible; ❺
```

```
    --nav-list-transform: translateX(0); ❻
    opacity: 1;
  }


  @media (min-width: 48em) {
    ul {
      --nav-list-visibility: visible;
      --nav-list-transform: translateX(0);  ❼
    }
  }
```

❶ Animate the opacity of the list.

❷ The list is hidden by default.

❸ Change the type of animation if the user has no preference for reduced motion.

❹ Move the list out of the viewport.

❺ If the list is expanded, show the list.

❻ If the list is expanded, move the list.

❼ On large viewports, always show the list in the viewport.

In this solution, you've decided to start with a simple animation and replace it if the user has no preference for reduced motion. It's possible to write the same pattern the other way around, as shown in Example 7-29. You can apply the transition first and remove it if users prefer reduced motion. I recommend the first solution because animation runs only in browsers that support the media feature and that do *not* express a preference for reduced motion.

Example 7-29. Alternative solution that removes animation if the user expresses a preference

```
ul {
  transition: transform 0.6s cubic-bezier(.68,
    visibility 0.3s linear;
}

@media (prefers-reduced-motion: reduce) {
  ul {
    transition: opacity 0.3s ease-in-out, visi
  }
}
```

If you have to make adjustments in JavaScript, you can use the `matchMedia()` method to determine whether the media query matches, as shown in Example 7-30.

Example 7-30. Querying the media feature in JavaScript

```
const mediaQuery = window.matchMedia("(prefers-r

if (mediaQuery.matches) { ❷
  console.log("add animation");
}

mediaQuery.addEventListener("change", () => { ❸
  if (mediaQuery.matches) {
    console.log("add animation");
  } else {
    console.log("reduce animation");
  }
});
```

❶ Query `prefers-reduced-motion` in JavaScript.

❷ Check if the query matches.

❸ Listen for changes.

Discussion

One advantage of using `visibility: hidden` is that it hides content visually and in terms of interactivity. Another

advantage is that you can animate the hidden element in CSS. It has only two states, `hidden` and `visible`, but you can combine it with another property like `transform` or `opacity` to create a slide- or fade-in effect. That wouldn't work with `display: none` because the `display` property is not animatable.

No motion versus less motion

As the name of the feature suggests, `prefers-reduced-motion` enables you to reduce motion. Some solutions online suggest turning off all animations and transitions completely (see Example 7-31), but reducing doesn't mean removing.

Example 7-31. Setting the animation and transition property to "none" on all elements

```
@media (prefers-reduced-motion: reduce) {
  *,
  *::before,
  *::after {
    animation: none !important;
    transition: none !important;
  }
}
```

Not all animations are the same. Motion on the screen isn't automatically bad, because it may help users, especially people with cognitive disabilities, understand the relationship between seemingly disparate objects, and it can improve decision making. It can also reduce cognitive load by making the path of a moving element visible, which has the benefit that users don't have to keep track of the movement themselves.

According to Val Head, an expert in animation on the web, the physical size of a screen matters less than the size of the motion relative to the space available on the screen. Animations that move items on the page across a large amount of space are more likely to be problematic. Subtle animations that don't involve a lot of movement, like transitions of opacity or color, aren't usually an issue. For users who prefer less motion, avoid scrolljacking, parallax effects, elements that move at different speed and position, large zooms, spinning effects, and pronounced animations.

Testing reduced motion

There are at least two ways of testing reduced motion.

In browsers based on Chromium, such as Google Chrome, Microsoft Edge, or Opera, you can emulate reduced motion in

the “Rendering” drawer in DevTools (see Figure 7-16).

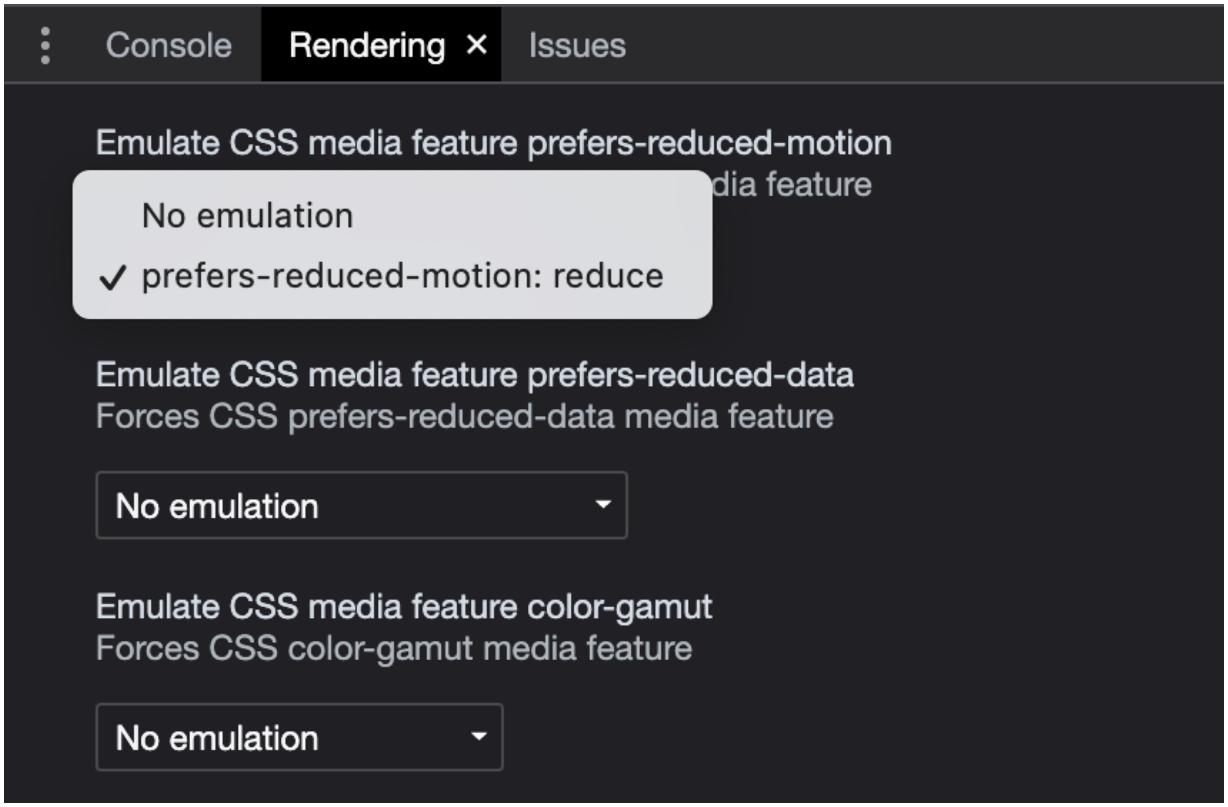


Figure 7-16. Emulating reduced motion in Chrome’s DevTools

Instead of emulating reduced motion, you can also apply it to the entire operating system.

macOS 13

`System Settings` → `Accessibility` → `Display` → `Reduce Motion`

iOS 16.1

`Settings` → `Accessibility` → `Motion` → `Reduce motion`

Windows 11

`Settings` → `Accessibility` → `Visual Effects` → `Animation Effects`

Android 10

`Settings` → `Accessibility Features` → `Accessibility` → `Advanced Visual Effects`

Android 13

`Settings` → `Accessibility` → `Color and Motion` → `Remove animations`

7.7 Add Submenus

Problem

Showing all items in large, nested menus at all times can be overwhelming for users, take up too much space, and add noise to a design.

Solution

Putting items in submenus and hiding them by default helps with that. If you're dealing with pages that contain subordinate pages and you want to display them in the navigation, you can do that by nesting lists, as shown in Examples 7-32 and 7-33.

Solution 1: Link and button

Example 7-32. A list item with a nested list

```
<nav aria-label="Main">
  <ul class="nav-list">
    <li>
      <a href="/home">Home</a>
    </li>

    <li>
      <a href="/products">Products</a>

      <ul class="nav-sublist">
        <li>
          <a href="/products/electronics">Electr
        </li>
        <li>
          <a href="/products/sports">Sports and
        </li>
        <li>
          <a href="/products/toys">Toys</a>
        </li>
```

```
        </ul>
      </li>
    </ul>
  </nav>
```

To give users the ability to access these nested lists on demand, you can hide them in CSS and add a button that toggles the visibility.

Example 7-33. Adding a button between the link and the nested list that toggles the visibility of the list

```
  <li>
    <a href="/products" id="mainnav-2">Products</a>

    <button
      type="button"
      aria-expanded="false" ❶
      aria-labelledby="mainnav-2"  ❷
      aria-controls="mainnav-2-sub" class="mainnav
    >
      <span aria-hidden="true">&amp;#9207;</span> ❹
    </button>


    <ul class="nav-sublist" id="mainnav-2-sub">
      <li>
        <a href="/products/electronics">Electronic
```

```
      </li>
      <li>
        <a href="/products/sports">Sports and Outd
      </li>
      <li>
        <a href="/products/toys">Toys</a>
      </li>
    </ul>
  </li>
```

- The `aria-expanded` attribute tells assistive technology if the element the button controls is expanded.
- `aria-labelledby` creates a reference to the parent link and uses its accessible name as the label.
- `aria-controls` creates a reference to the sublist.
- `&#9207;` is the HTML entity for a downward pointing triangle.

The nested list is hidden if the `aria-expanded` attribute of the preceding button is set to `false`, as shown in Example 7-34.

Example 7-34. Hiding the list in CSS

```
[aria-expanded="false"] + .nav-sublist {
  display: none;
}
```

You can see how the list is hidden by default in Figure 7-17.

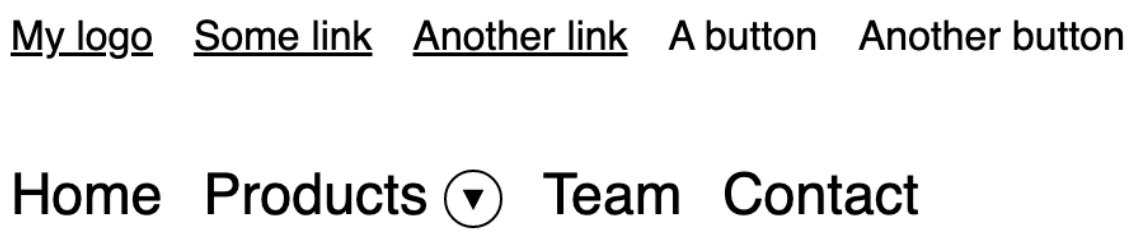


Figure 7-17. The navigation showing a button next to the “Products” link

Clicking the button toggles the `aria-expanded` attribute, as shown in Example 7-35 and Figure 7-18.

Example 7-35. Click event that toggles the value of the `aria-expanded` attribute

```
const nav = document.querySelector("nav");
nav.addEventListener("click", (e) => {
  if (e.target.classList.contains("mainnav-toggl
    const isOpen = e.target.getAttribute("aria-e

    e.target.setAttribute("aria-expanded", !isOp
  }
});
```

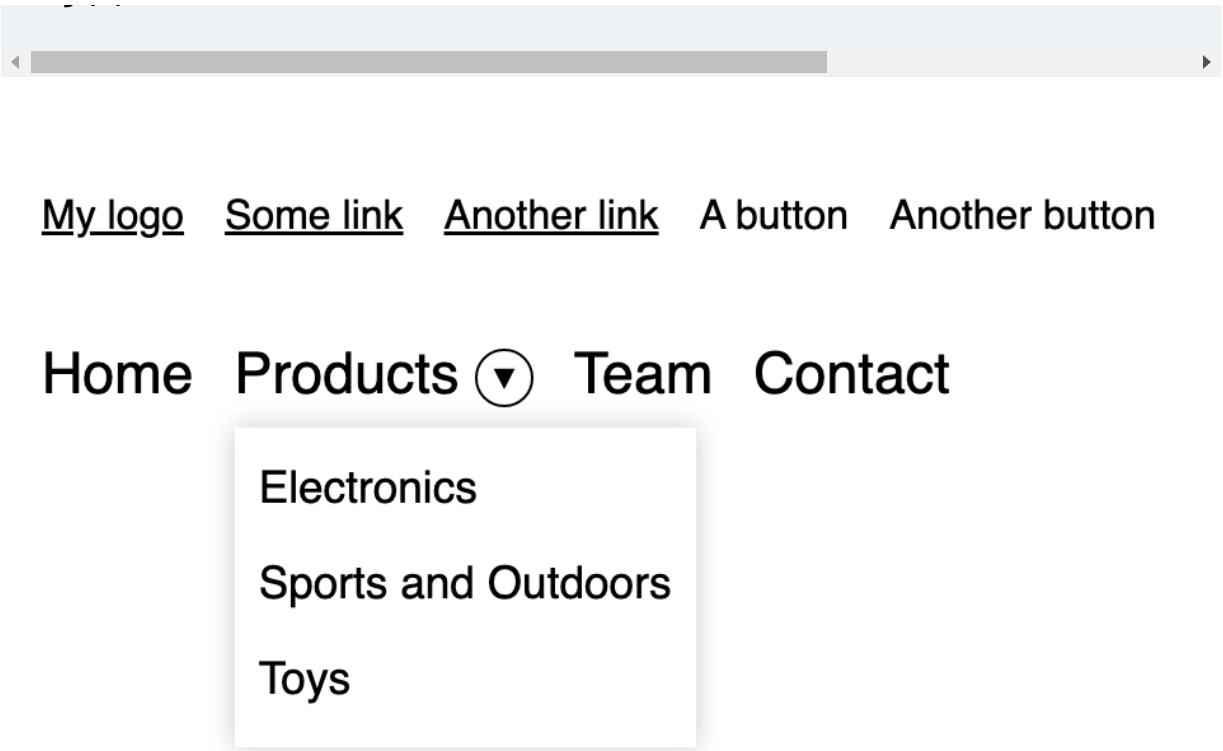


Figure 7-18. The navigation with the nested list of links expanded below the "Products" link

Solution 2: Button only

You might not want to have two interactive elements with different purposes next to each other in the main navigation. Users might not expect that clicking the text navigates to a new page and that clicking only the button toggles the navigation. On top of that, this solution adds an extra tab stop to each item. An alternative solution is to put a button only in the first level of the navigation and move the link into the submenu, as shown in Example 7-36 and Figure 7-19.

Example 7-36. Instead of a link and button, this solution uses a button only and moves the link into the list

```
<nav aria-label="Main">
  <ul class="nav-list">
    <li>
      <a href="/home">Home</a>
    </li>

    <li>
      <button aria-expanded="false" aria-controls
        class="mainnav-toggle-sub">
        Products
      </button>

      <ul class="nav-sublist" id="mainnav-2-sub">
        <li>
          <a href="/products">All Products</a>
        </li>
        <li>
          <a href="/products/electronics">Electr
        </li>
        <li>
          <a href="/products/sports">Sports and
        </li>
        <li>
          <a href="/products/toys">Toys</a>
        </li>
```

```
        </ul>
      </li>
    </ul>
  </nav>
```

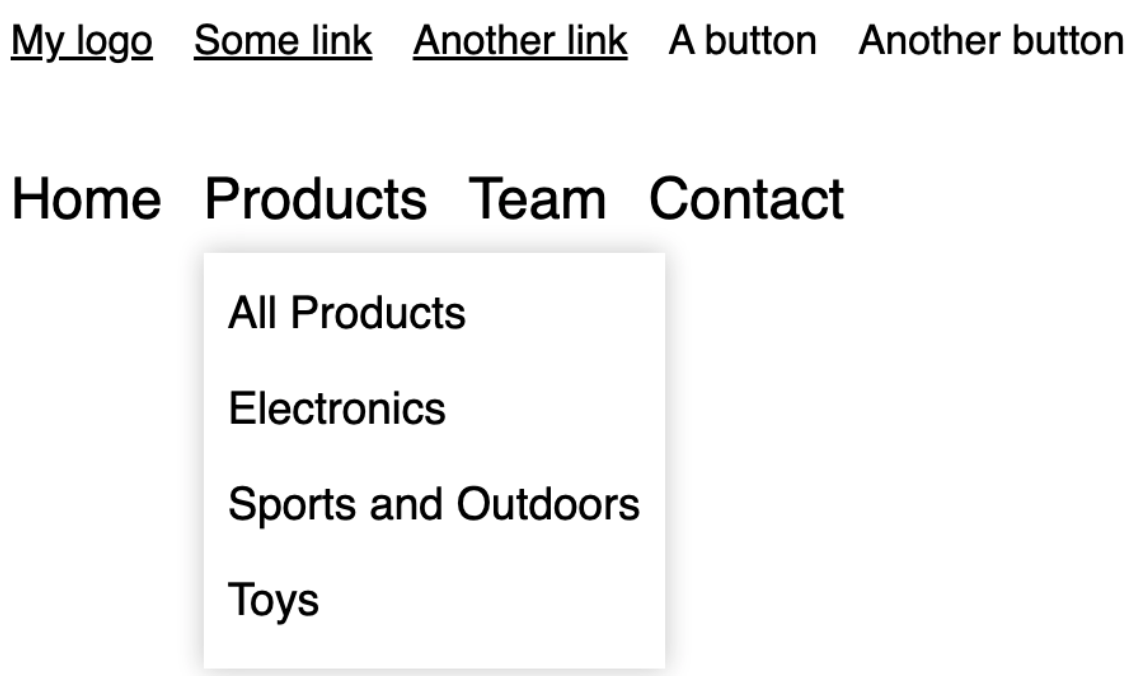


Figure 7-19. The navigation with the nested list of links expanded below the "Products" button

Discussion

Automatic activation

Both proposed solutions require the user to click a button to open the related list. Generally, you want to give users control and you want build components that are predictable, but you

also want to make the user's experience as pleasant as possible. It might be convenient for mouse users to show the submenus automatically on hover. For keyboard and screen reader users, on the other hand, it can be frustrating if submenus open automatically. Just because they're focusing a parent item doesn't mean that they also want to interact with the submenu. It's important to find a good balance between different ways of interaction. A solution that works well for some users might not be ideal for others.

Links versus buttons

I'm distinguishing between links and buttons and their responsibilities. *Links* navigate to a new page and *buttons* perform actions in JavaScript. This is important because it matches the user's expectations.

Especially with nested navigations, you often see solutions that mix up both elements. This often makes it difficult or even impossible for the user to interact with these elements at all. Avoid any solution that turns links into buttons or misuses links as buttons, as shown in Example 7-37.

Example 7-37. Bad practice: A broken hyperlink semantically turned into a button

```
<a href="#" role="button">Products</a>
```

You can read more about how to implement links in Chapter 3 and buttons in Chapter 4.

Animation

To animate opening and closing of submenus, follow the rules described in Recipe 7.6:

- Hide sublists properly from all users by combining properties like `opacity` or `transform` with `visibility: hidden`.
- Design subtle animations or provide an alternative for users who prefer reduced motion.

Testing with screen readers

If you test the first solution with screen readers and you try to access the link and button, you get the results shown in Table 7-2.

Table 7-2. Menu link and button announced in different screen readers

Screen reader	Announcement
JAWS	Products, link. (for the link) / Products, button, collapsed. (for the button)
NVDA	Products, link. / Products, button, collapsed.
VoiceOver iOS	Products, link. / Products, button, collapsed.
VoiceOver macOS	Link, Products, 2 of 4. / Products, collapsed, button.
TalkBack	Products, link, 2 of 4. / Collapsed, Products, button.

7.8 Avoid Confusion with Menus

Problem

Many tutorials suggest that to make a navigation accessible, you need ARIA. As illustrated in Recipes 7.4, 7.5, and 7.7, ARIA attributes can be useful, but sometimes they do more harm than good. Using the wrong ARIA attributes can confuse users,

distort their expectations, or make the whole navigation unusable for some.

Solution

Keep it simple and use ARIA only where it's needed. Navigations are a collection of links. If you have a lot of links, you might hide some of them and show them conditionally. To make these actions accessible, use attributes like `aria-expanded`, `aria-labelledby`, or `aria-controls`.

In Example 7-38, you use ARIA in only two places.

Example 7-38. Main navigation with four list items; one of them has a nested list that can be toggled by pressing a button

```
<nav aria-label="Main"> ❶
  <ul class="nav-list">
    <li>
      <a href="/home">Home</a>
    </li>

    <li>
      <a href="/products" id="mainnav-2">Product

      <button aria-expanded="false"
```

```
                aria-labelledby="mainnav-2"
                aria-controls="mainnav-2-sub"
                class="mainnav-toggle-sub">  ❷
          &#9207;
        </button>

        <ul class="nav-sublist" id="mainnav-2-sub">
          <li>
            <a href="/products/electronics">Electr
          </li>
          <li>
            <a href="/products/sports">Sports and
          </li>
          <li>
            <a href="/products/toys">Toys</a>
          </li>
        </ul>
      </li>
      <li>
        <a href="/team">Team</a>
      </li>
      <li>
        <a href="/contact">Contact</a>
      </li>

    </ul>
  </nav>
```

1. `aria-label` gives the navigation landmark a unique label.
2. Attributes necessary to make the custom toggle button accessible.

Many sites use `menu` roles to turn navigations into menus. This is an anti-pattern.

Discussion

Navigations often get confused with menus. They're not the same and they serve different needs. According to the ARIA 1.1 specifications, *navigations* are collections of links for navigating related documents, and *menus* are collections of actions to perform in a document. More specifically, the menu bar is the presentation of a menu, usually always visible, laid out horizontally, and similar to menu bars found in operating systems like Windows or Mac. Authors *should* ensure that menu bar interaction is similar to the interaction in a desktop UI.

The specification is very clear: you should use `navigation` for navigating the document or related documents, and `menu` only for a list of actions or functions similar to menus in desktop applications. Most of the time, it's easy to make this distinction,

but sometimes different interactive elements in navigations have different tasks. You might have a navigation that also includes a button that performs an action, like opening a modal window, or a menu where one action is navigating to another page, like a help page. When that's the case, it's important not to mash up ARIA roles. Instead, identify the component's main purpose and pick the markup and roles accordingly.

The `<nav>` element has an implicit ARIA role of `navigation` that suffices to communicate that the element is a navigation, but often sites also use `menu`, `menubar`, and `menuitem`, as shown in Example 7-39. People sometimes use these terms interchangeably, thinking that combining them could improve the experience for screen reader users, but misusing these roles can have serious implications.

Example 7-39. Bad practice: Example of a navigation that uses menu roles

```
<nav aria-label="Main">
  <ul class="nav-list" role="menubar">
    <li role="presentation">
      <a href="/home" role="menuitem">Home</a>

    </li>
    …
  </ul>
```

```
  </nav>
```

Screen readers that support these roles will announce the menu as such. Savvy users expect certain keyboard shortcuts to work with menus and menu bars. Based on the APG, this includes using:

- `Enter` and `Space` to select menu items
- Arrow keys in all directions to navigate between items
- The `Home` and `End` keys to move focus to the first or last item, respectively
- `a-z` to move focus to the next menu item with a label that starts with the typed character
- `Esc` to close the menu

Even if you implement these keyboard commands correctly, users might not know that they can use them. These shortcuts are not exposed, and users might not expect that certain commands now perform different actions.

Entering a menu might instruct the screen reader to switch modes. In JAWS, for example, you have browse mode and forms mode. Browse mode allows you to use shortcuts like `Tab` to jump from one interactive element to another, or to jump from heading to heading by pressing `H`. In forms mode, these

shortcuts don’t work anymore, but pressing these keys performs their default action. This can be confusing if it’s not expected.

There’s a lot to consider when you create menus and menu bars. Starting with whether it’s appropriate to use them in the first place. When you’re building a typical website, a SPA, or a web app, the `<nav>` element with a list and links is all you need. The underlying stack doesn’t matter. Unless you’re building something very close to a desktop application, avoid menu roles in navigations.

Chapter 8. Toggling Content Visibility

Limited space and the client's desire to fit as much information as possible onto a page pushes designers to their limits because there's only so much you can show at a time. They often reach for conditional content hiding to solve that problem and use design patterns like tabs, accordions, fly-out navigations, or disclosure widgets. While it's usually just better to show content, sometimes these solutions are unavoidable. When implementing them in HTML, CSS, and JavaScript, you must use the proper hiding technique and communicate the state accordingly.

8.1 Hide Content

Problem

CSS offers many different solutions for hiding content. If you don't pick the proper technique for the correct use case, it can affect users negatively depending on how they access your site.

- It can be frustrating and confusing for keyboard and screen reader users if they have to navigate content they don't want to, have to, or should be able to access.

- Tabbing through dozens of “invisible” interactive elements can be cumbersome.
- Screen readers might be unable to retrieve semantic information from an incorrectly hidden element.

Solution

You have to pick the correct hiding technique for the right use case.

Visually hidden

Suppose you want to hide content visually but keep it accessible to screen reader and keyboard users. In that case, you must use a combination of CSS properties (see Example 8-1) because it’s impossible to do it natively in CSS.

Example 8-1. A custom “visually hidden” class

```
.visually-hidden:not(:focus) {
  clip-path: inset(50%);
  height: 1px;
  overflow: hidden;
  position: absolute;
  white-space: nowrap;
  width: 1px;
}
```

Visually and semantically hidden

If you want to show content under only certain conditions (such as when the user clicks a button), you must hide it from everyone, as shown in Examples 8-2, 8-3, and 8-4.

Example 8-2. Invisible, not machine-readable, and doesn't take up any space in the document

```
.more-content {
  display: none;
}
```

Example 8-3. Same functionality as `display: none`, but in HTML

```
<div class="more-content" hidden></div>
```

Example 8-4. Invisible, not machine-readable, but takes up space in the document

```
.more-content {
  visibility: hidden;
}
```

Visible but semantically hidden

When you have decorative content, usually images and icons, you want to display them but also make them machine-unreadable, as shown in Examples 8-5 and 8-6.

Example 8-5. An empty `alt` attribute removes an element from the accessibility tree

```
<img src="/illustrations/floral.png" alt="">
```

Example 8-6. Hiding an element from the accessibility tree in HTML by setting the `aria-hidden` attribute to true

```
<button>
  <svg aria-hidden="true">
    <use href="sprite.svg#send">
  </svg>
  Send
</button>
```

Discussion

The whole purpose of hiding content is to expose users to only certain contents or parts of the UI if or when they're relevant to

them. The term *hiding* has several different meanings in this context.

You can *hide content visually* when you want to exclude elements from the visual representation of the page but keep it accessible to keyboard or screen reader users. Good examples of that are skip links that become visible (see Recipe 6.6) or live regions (see Recipe 3.7) that have no special meaning for keyboard users. There's no standard property in CSS or attribute in HTML for that; you have to use a combination of properties, as shown in Example 8-1. You can learn what each of these properties does in James Edwards's "The anatomy of visually-hidden".

This solution is not suitable for hiding content from everyone.

You can hide *content from everyone* under certain conditions. That can be useful if the content is not primarily important (see Recipe 8.2) or if you don't have enough space and need to hide parts of the UI conditionally (see Recipe 7.5). The solutions in Examples 8-2, 8-3, and 8-4 work well in this case.

Purely decorative or redundant content can be *hidden semantically*. This usually applies to decorative images or

illustrations (see Example 8-5) and icons used to support text, as shown in Example 8-6.

Consider hiding information for only some users as a last resort because a lot can go wrong. You may use an improper hiding technique and make content inaccessible to more people than intended. You might use the right approach, but your users could still miss the information because they don't notice it. Also, our decisions are sometimes based on wrong assumptions. For example, not all screen reader users are blind. Before you hide something, try writing and structuring your content differently or tweaking the design.

Hiding in CSS and HTML

Besides the solutions in this recipe, other ways of hiding content in HTML and CSS exist. Depending on what you're trying to achieve, not all of them are useful. For example, the solution in Example 8-7 hides content visually. However, it still takes up space on the page, and content within the element might be accessible via keyboard and screen readers.

Example 8-7. Hiding an element only visually

```
div {
  opacity: 0;
```

```
  }
```

Kitty Giraudel provides an overview of different hiding solutions and how they affect the user experience in their blog post “Hiding content responsibly”. See Table 8-1.

Table 8-1. An overview of different hiding solutions and their outcome (Source: Kitty Giraudel)

Method	Visible	Available
.sr-only class	No	Yes
aria-hidden="true”	Yes	No
display: none	No	No
visibility: hidden	No, but space remains	No
opacity: 0	No, but space remains	Depends
transform: scale(0)	No, but space remains	Yes

Incorrect hiding

Whichever hiding strategy you use, there are specific rules you must follow.

- Only hide content when you can't fix the problem by adapting the content or design.
- Don't put `aria-hidden="true"` or `role="presentation"` on focusable interactive elements (see "Fourth Rule of ARIA Use"). Screen reader users might be unable to access a button removed from the accessibility tree using the virtual cursor (see Example 8-8). However, it is still available to screen reader users using the keyboard.
- When you hide content visually using `opacity`, `height`, or `transform`, combine them with a property that also hides the element semantically, such as `visibility: hidden;`
- Don't hide elements semantically that are referenced elsewhere in the document (see Example 8-9).

Example 8-8. Bad practice: An interactive element removed from the accessibility tree only

```
<button aria-hidden="true">
  Share
</button>
```

Example 8-9. Bad practice: A referenced element removed from the accessibility tree

```
<input type="checkbox" style="display: none" id=
<label for="toc">I accept the terms of service</
```

Incorrectly hidden content is a common accessibility issue that you can avoid by first figuring out what you're trying to achieve and how it affects your users and then deciding how to get the desired result.

8.2 Create a Native Disclosure Widget

Problem

When you have a lot of content on a page, especially content that's relevant to only some users or only in specific situations, your pages can become cluttered. That can negatively influence orientation and navigation for all users.

Solution

You can hide irrelevant content by default and allow users to toggle its visibility as needed, as shown in Example 8-10.

Example 8-10. Native disclosure widget in HTML, content hidden by default

```
<details>
  <summary>Show details</summary>

  <p>Detailed content goes here…</p>
</details>
```

You can also show the content by default and allow users to close it, as shown in Example 8-11.

Example 8-11. Native disclosure widget in HTML, content visible by default

```
<details open>
  <summary>Show details</summary>

  <p>Detailed content goes here…</p>
</details>
```

You can change the styling of the marker, as shown in Example 8-12. Be aware that some screen reader/browser combinations announce the marker.

Example 8-12. Styling the toggle button in CSS

```
summary::marker {
  content: "+ ";
```

```
}

details[open] summary::marker {
  content: "– ";
}
```

You can use the `toggle` event in JavaScript to keep track of the element's state, if you need to, as shown in Example 8-13.

Example 8-13. Listening to the toggle event in JavaScript

```
const details = document.querySelector("details"

details.addEventListener("toggle", (e) => {
  console.log(details.open);
});
```

Discussion

HTML offers a native disclosure widget from which users can obtain additional information or controls (see Figure 8-1). `<details>` represents that widget. The nested `<summary>` element provides the legend for the widget as well as the accessible name for the trigger.

Figure 8-1. Screenshot of the details element: closed and open

Pros

A native disclosure widget has many advantages:

- It works without JavaScript.
- Most browsers support it.
- The "find in page" feature in Chromium-based browsers allows searching the hidden content.
- You can open it by default using the `open` attribute (see Example 8-11).
- You can style it using CSS (see Example 8-12).
- You can keep track of its state using the `toggle` event and the `open` property (see Example 8-13).

Cons

The `details` element is well supported, but you should be aware of its inconsistencies across different browsers and screen readers before you use it. Accessing the element with a screen reader, you will get significantly different results

depending on the browser and software you're using. In addition to Scott O'Hara's research in "The details and summary elements, again", I did more testing and summarized it on my website in "details/summary inconsistencies".

A summary of my tests:

- Announcements go from little information ("Show Details" in VoiceOver with Safari 16 on iOS) to too much information ("Right pointing triangle, Show details, collapsed, summary, group" in Firefox on macOS).
- Removing or changing the triangle doesn't seem to affect any screen reader/browser pairing, except Firefox, with all tested screen readers.
- VoiceOver on macOS with Chrome/Edge and Safari and TalkBack on Android with Chrome provide the most consistent experience across browsers.
- VoiceOver on iOS with Safari 16 is also very consistent but in a bad way. It doesn't announce any role or state.
- Details only expands in Chromium-based browsers when you search with Cmd/Ctrl + F (find-in-page).
- To remove the triangle in Safari, you must set `::-webkit-details-marker` to `display: none`. `::marker` or `list-style: none;` don't work.

When not to use it

In Chromium-based browsers the “find in page” feature can find content within collapsed details elements. It will open the element and highlight the matching string. That’s not the case in Firefox and Safari, as illustrated in Figure 8-2.

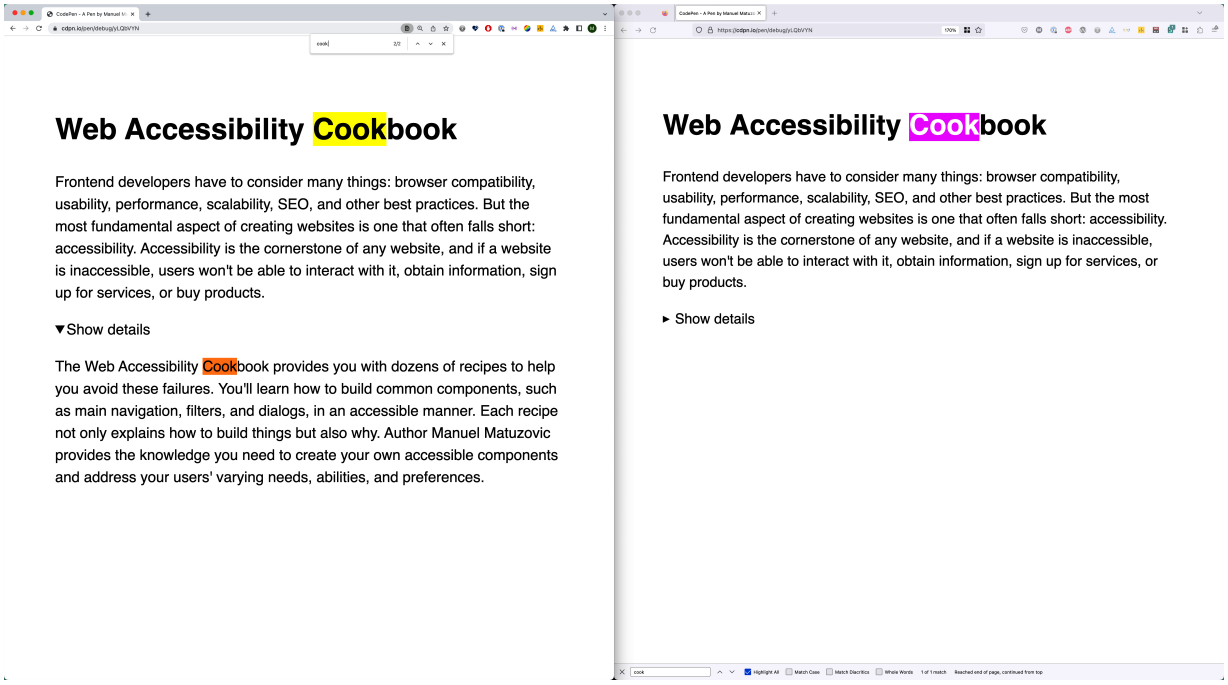


Figure 8-2. Searching for the term “cook”: Chrome on the left and Firefox on the right

This feature is an important detail you should consider when deciding whether the `details` element is the right solution for a given problem. Adrian Roselli states in “Details / Summary Are Not [insert control here]” that this browser behavior can be especially problematic for command-centric components like navigations, menus, or dialogs because you probably don’t

want these elements to open randomly when the user performs a search.

Depending on your needs, a custom disclosure widget might be an option, especially when you're aiming for consistent behavior across browsers, you want to prevent the "find in page" behavior, or you need more features than the default widget offers. You can learn how to build one accessibly in Recipe 8.3.

Using the `details` element has a lot of benefits, but there are also many inconsistencies between platforms, which may or may not be an issue. Depending on the experience you want to provide users and how acceptable the discrepancies are, you can rely on either the native element or a custom solution.

8.3 Create a Custom Disclosure Widget

Problem

As described in Recipe 8.2, the quality and quantity of information a screen reader user gets when interacting with a native disclosure widget depends on the software they're using.

The `details` element is not an option if you want control over the information screen reader users get or if you need to ensure a consistent user experience across platforms.

Solution

Instead of relying on the native element, you can build a custom disclosure widget, as shown in Examples 8-14, 8-15, 8-16, and 8-17.

Example 8-14. A basic disclosure needs a button and a container for the content

```
<div class="disclosure">
  <button
    aria-expanded="false" ❶
    aria-controls="content" ❷
  >
    Show details
  </button>

  <div class="disclosure-content" id="content">
    <p>Detailed content goes here…</p>


  </div>
</div>
```

- ❶ The `aria-expanded` attribute tells assistive technology if the element the button controls is expanded.
- ❷ `aria-controls` creates a reference to the element it controls.

You can use the information about the widget's state provided by the `aria-expanded` attribute as a condition for styling.

Example 8-15. Hiding the content via CSS based on the state of the widget

```
[aria-expanded="false"] + .disclosure-content {
  display: none;
}
```

You can use CSS to make the transition smoother.

Example 8-16. Alternative: Animating the height via CSS based on the state of the widget

```
.disclosure {
  --_height: 0fr;

  display: grid; ❶
  justify-content: start;
```

```
    grid-template-rows: 1.4rem var(--_height); ❷
  }

  @media (prefers-reduced-motion: no-preference) {
    .disclosure {
      transition: visibility 0.3s, grid-template-r
    }
  }

  .disclosure > [aria-expanded] {
    width: fit-content;
  }

  .disclosure > [aria-expanded="false"] + .disclos
    visibility: hidden;
  }

  .disclosure:has([aria-expanded="true"]) { ❹
    --_height: 1fr;
  }

  .disclosure .disclosure-content {
    overflow: hidden;
  }
```

❶ It's impossible to animate a zero height to an auto height in CSS, but it's possible to animate grid rows and columns.

You're using `Grid` here solely to animate the widget.

❷ Define two rows: one for the button and one for the content. The height of the content row is 0 by default.

❸ Only animate when the user has no preference for reduced motion.

❹ If the content is expanded, change the height to one fraction (`1fr`).

To open and close the custom widget, you have to toggle the value of the `aria-expanded` attribute on click, as shown in Example 8-17.

Example 8-17. Toggle the state of the widget on click

```
const button = document.querySelector("button");

button.addEventListener("click", (e) => {
  button.setAttribute(
    "aria-expanded",
    button.getAttribute("aria-expanded") === "fa
    );
});
```

Discussion

The solution to this problem is short and straightforward, but its tiny details are essential.

The button should come before the content in the DOM because you want to ensure it's always the first focusable element in your widget.

It depends on how you're using the custom disclosure element, but most of the time, you want to avoid moving focus and leave it to the user to decide what they want to do next.

You have to hide the content correctly for everyone (see Recipe 8.1). You can use `display: none` or, if you want to animate the toggling action, `visibility: hidden`.

The target size of the button must be large enough (see "Target size" in Recipe 3.2), and you can use an icon to illustrate its state visually, as shown in Examples 8-18 and 8-19).

Example 8-18. Button with a semantically hidden icon

```
<button aria-expanded="false" aria-controls="con
  Show details
```

```
    <svg aria-hidden="true">
      <use href="sprite.svg#chevron-down">
    </svg>
  </button>
```

Example 8-19. The icon rotates 180deg when the widget is expanded

```
svg {
  transition: transform 0.3s;
}

[aria-expanded="true"] > svg {
  transform: rotate(180deg);
}
```

This pattern is simple but versatile. You can use it to toggle text, the site navigation on narrow viewports (see Recipe 7.5), to show and hide submenus in large navigations (see Recipe 7.7), or toggle other interactive content like tooltips or modals.

You can find another interesting implementation of a custom disclosure widget in a web component called `details-utils` by Zach Leatherman. He takes the native `details` element as a baseline and enhances it progressively by wrapping it in a custom element, as shown in Example 8-20.

Example 8-20. The native `details` element wrapped in a custom element with custom attributes

```
<details-utils force-open="(min-width: 48em)" fo
  <details open>…</details>
</details-utils>
```

8.4 Create Groups of Disclosure Widgets

Problem

If you have a lot of structured content on a single page, the amount of information can increase the cognitive load on users. That can overwhelm them, making it hard to focus and scan content.

Solution

You can hide structured groups of content in disclosure widgets and let the user decide which information they want to see, as shown in Examples 8-21, 8-22, 8-23, and 8-24.

Example 8-21. Headings and related content grouped in a section

```
<section aria-labelledby="faq_heading" class="fa
  <h2 id="faq_heading">
    Frequently asked questions
  </h2>

  <h3> ❷
    First question
  </h3>
  <div class="faq-content">
    <p>
      First answer…
    </p>
  </div>

  <h3>
    Second question
  </h3>
  <div class="faq-content">
    <p>
      Second answer…
    </p>
  </div>

   ❸
</section>
```

❶ Section labeled by its heading

❷ Group of question and answer

❸ Additional groups

Example 8-22. Alternative: Each content wrapper can be a landmark

```
    <h3 id="faq_q1">
      First question
    </h3>
    <div class="faq-content" aria-labelledby="faq_
      <p>
        First answer…
      </p>
    </div>
```

Example 8-23. Turning a simple group of headings and content into an accordion (a group of disclosure widgets) in JavaScript

```
const faq = document.querySelector(".faq");
const headings = faq.querySelectorAll("h3");
```

```
  for (let i = 0; i < headings.length; i++) { ❶
    const button = document.createElement("button"
    const heading = headings[i];
    const content = heading.nextElementSibling;
    const id = `faq_${i}`; ❸

    button.setAttribute("aria-expanded", false); ❹
    button.setAttribute("aria-controls", id);
    button.textContent = heading.textContent;
    heading.innerHTML = "";  ❺
    heading.append(button);

    content.setAttribute("id", id); ❻
  }

  faq.addEventListener("click", (e) => { ❼
    const button = e.target.closest("[aria-expande
    const isOpen = button.getAttribute("aria-expan

    if (button) {
      button.setAttribute("aria-expanded", isOpen)
    }
  });
```

❶ Iterate over all headings in the group.

❷ Create a button for each heading.

❸ Define a unique ID.

❹ Set ARIA attributes (see Recipe 8.3 for details).

❺ Replace the content of the headings with the button.

❻ Connect the button to the content.

❼ Add a click event on the whole component.

❽ Toggle the `aria-expanded` attribute.

Just like in Recipe 8.3, the `[aria-expanded]` attribute selector controls whether the respective content is visible.

Example 8-24. Resetting button styles and hiding content

```
.faq [aria-expanded] {
   all: unset; ❶
}

.faq [aria-expanded]:focus-visible {
  outline: 0.25em solid;  ❷
}

h3:has([aria-expanded="false"]) + .faq-content {

  display: none;
}
```

❶ Remove default button styles.

❷ Show an outline around the button for keyboard users.

❸ Hide content that follows a heading that contains a nonexpanded button.

Discussion

An accordion can be an excellent tool for removing nonessential information in an interface, enabling the user to focus on one thing at a time, as illustrated in Figure 8-3.

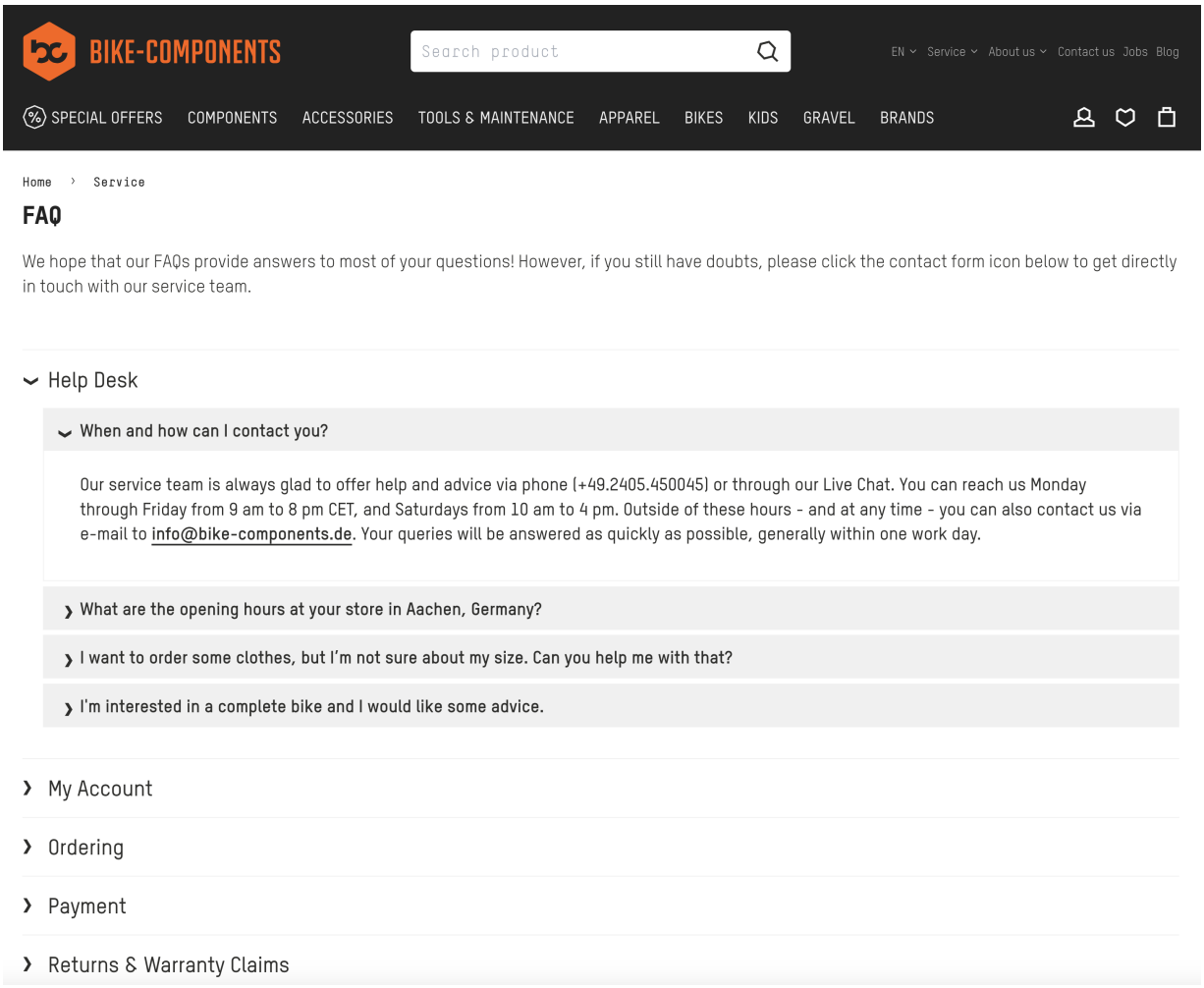


Figure 8-3. Structured and clean FAQ page on bike-components.de

Accordions form related entities visually, and they should do that semantically and functionally, too. That is also the most significant difference between simple disclosure widgets and accordions. Each segment doesn't stand for itself, but it's connected to its siblings within a structured group of segments.

You express this semantic connection using HTML and JavaScript in different ways: the section element and its

heading group them topically, and the consistent structure (heading followed by content) groups them semantically (see Example 8-21). You can allow users to expand all segments or show only one at a time, closing the expanded element automatically, which groups them functionally.

Progressive enhancement

An accordion is a perfect example of a component you can progressively enhance. You start with a labeled section with several headings followed by related content that works without JavaScript, as shown in Example 8-21. You add interactive functionality by wrapping the content within each heading in a button. You don't replace the heading element with the button because the headings are an additional way to navigate the accordion using a screen reader. The button is connected to the content element via `aria-controls`, and it communicates whether the respective section is open using the `aria-expanded` attribute, which you toggle on click (see Example 8-23).

Navigation

You've learned about the advantages of using a button for click events in Chapter 4, and the accordion illustrates them well. All

you have to do is attach a click event to your component, and you get keyboard accessibility for free. As a keyboard user, you can navigate the accordion using the `Tab` and `Shift` + `Tab` key. Optionally, you can add keyboard support by listening for the arrow up and down keys, as described in the ARIA Authoring Practices Guide.

Screen reader users also benefit from these shortcuts. In addition to keyboard commands, they can navigate to the component by using the "navigation via landmark" feature in their screen reader (more about that in Recipe 1.5). Each content wrapper can be a landmark, too, enabling users to jump to the open accordion elements (see Example 8-22 and Figure 8-4). Be cautious with that because navigation via landmark can become tedious if there are too many landmarks.

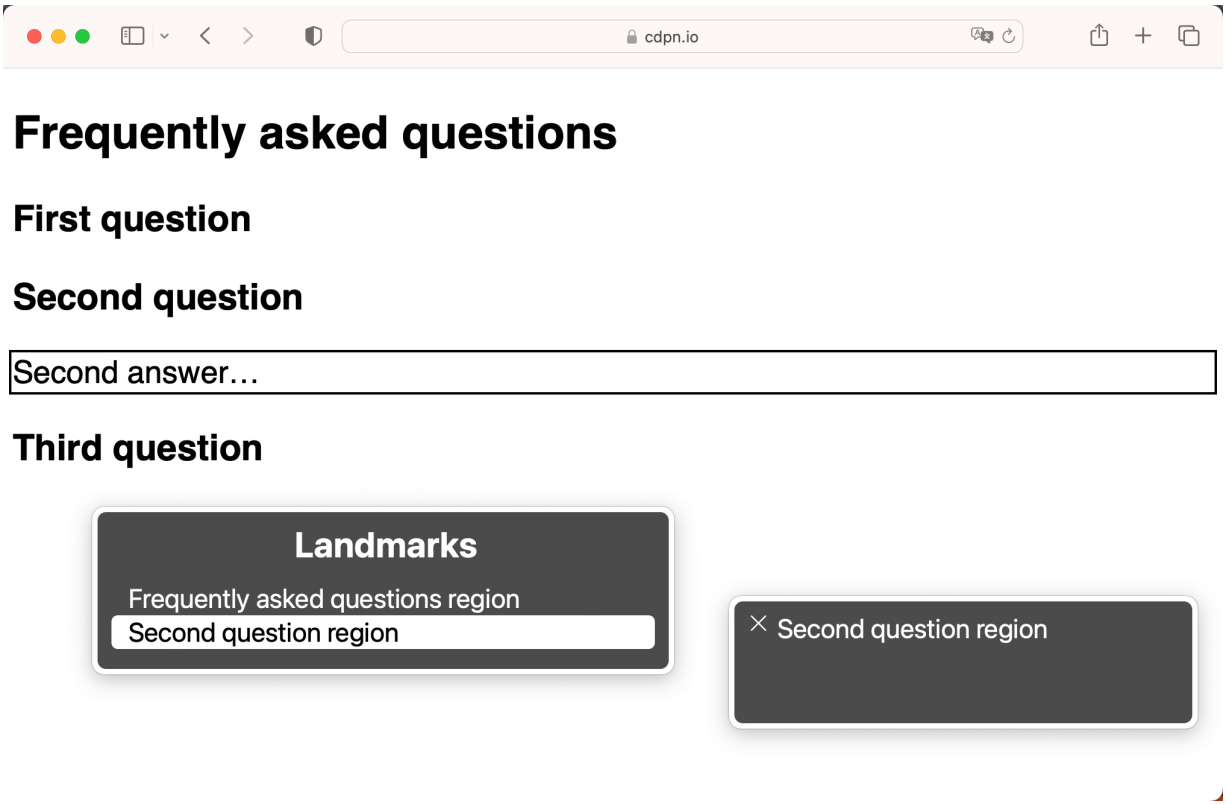


Figure 8-4. Landmark navigation in VoiceOver showing the FAQ region and the expanded question and answer

The implementation in this accordion is pretty basic, but regarding accessibility, it checks the most important boxes. You can learn more about user experience considerations related to accordions in Vitaly Friedman's "Designing the Perfect Accordion".

See Also

- "Inclusively Hidden" by Scott O'Hara

Chapter 9. Constructing Forms

Forms are an integral part of the web. Besides links, they're the most important tool for providing interactivity to a website. They allow a user to sign up for newsletters, log into their bank account, book a hotel room, or post an idea on social media.

Form design is complex, and with complexity comes the potential for mistakes and bad decisions. I could write an entire book about the do's and don'ts of form design. This chapter only scratches the surface focusing on the most common and significant problems users face.

9.1 Create Forms

Problem

If forms are not designed well, users may have a hard time performing critical tasks like filling out an application or contact form or ordering from an online store. Bad choices in form design affect all users, especially less savvy users and those with cognitive disabilities, motor disabilities, and low vision.

Solution

When you create a form, follow certain basic principles.

- Use native form elements, if possible.
- Use the proper form element for the intended purpose.
- Keep forms as short and straightforward as possible.
- Label and describe all fields.
- Inform users about changes to the form.

Figure 9-1 and Example 9-1 illustrate using different elements and attributes to create a simple but feature-rich form in HTML.

Figure 9-1. A simple sign-up form using labeled `form` fields and `fieldsets`

Example 9-1. A simple sign-up form using labeled `form` fields and `fieldsets`

```
<form>
  <div>
    <label for="username">Username</label>
    <input type="text" id="username" autocomplet
  </div>

  <div>
    <label for="email">E-Mail</label>
    <input type="email" id="email" autocomplete=
  </div>

  <fieldset>
    <legend>T-Shirt size</legend>

    <div>
      <input type="radio" id="s" name="shirt">
      <label for="s">Small</label>
    </div>

    <div>
      <input type="radio" id="m" name="shirt">
      <label for="m">Medium</label>
    </div>

    <div>
```

```
        <input type="radio" id="l" name="shirt">
        <label for="l">Large</label>
      </div>
    </fieldset>

    <button>
      Sign up
    </button>
  </form>
```

Discussion

It's usually wrong to make assumptions about your users, but one assumption I'm willing to make is that users are generally not excited about filling out forms. It doesn't matter whether it's offline or online: filling out forms isn't fun. So, when you create a form, make the experience as pleasant as possible by avoiding complexity and providing clarity.

To achieve that, follow basic usability and accessibility principles. The following sections outline some of the most important.

Use well-established patterns

There's nothing wrong with trying to find creative solutions for problems in web design, but originality isn't always the answer. One of the Inclusive Design Principles (defined by Heydon Pickering, Léonie Watson, Henny Swan, and Ian Pouncey) is "be consistent:" use familiar conventions and applying them consistently. Instead of developing new form-design patterns, stick to what already exists. Provide users with familiar patterns instead of forcing them to learn new ones. As Frank Chimero writes:

> *Many sites will share design solutions, because we're using the same materials. The consistencies establish best practices; they are proof of design patterns that play off of the needs of a common medium, and not evidence of a visual monoculture.*

Unfamiliar patterns increase the cognitive load on users. This is especially problematic with forms because users are already performing tasks that take a lot of mental effort. Pick a simple solution and a pattern users likely already know from other websites.

For example, instead of using a toggle switch, use a checkbox. In his article, Joel Holmberg explains in detail why toggle switches are problematic. One reason is that their state isn't always

straightforward. What does the toggle in Figure 9-2 communicate? Do the green color and the label ON indicate a status or an action?

Figure 9-2. A switch toggle that communicates either a state or an action

Instead of making text input fields look special, use the rectangular shape users are familiar with. In his book *Form Design Patterns*, Adam Silver explains that a text box should look like a text box because an empty box signifies “fill me in.” Removing borders or using only a bottom border removes the signifiers (design aspects that suggest how to use the object) that communicate the perceived affordance. The input fields in Figure 9-3 are barely recognizable.

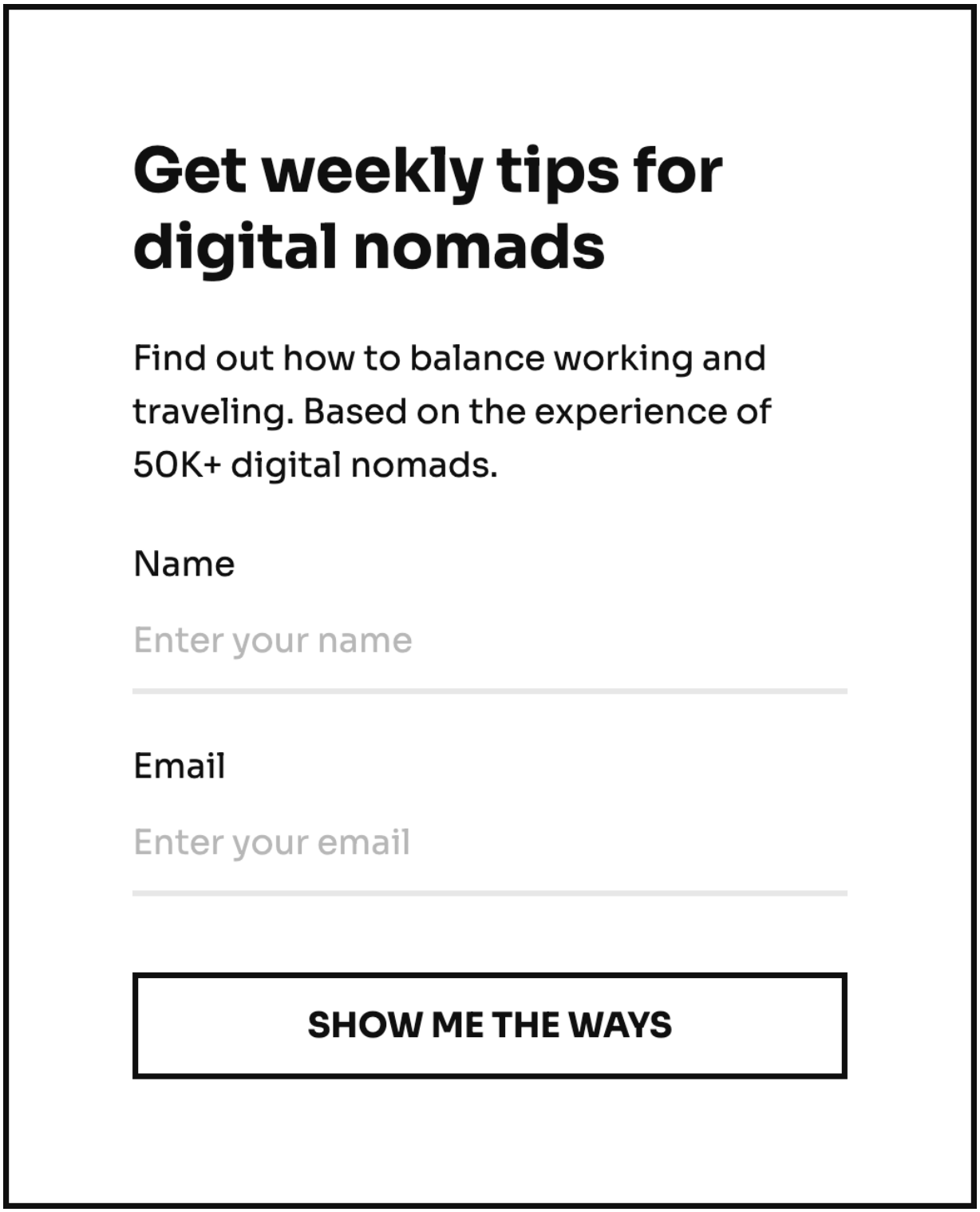


Figure 9-3. A template for a newsletter sign-up form: instead of a rectangular shape, they use a light gray bottom border with light gray placeholder text

Keep it short

The *gov.uk* design manual for structuring forms advises you to create a question protocol, a list of all the information you need from your users, before you design a form. It forces you to question why you're asking users for each item of information and gives you a way of challenging unnecessary questions. They suggest you add a question to your form only if you know:

- That you need the information to deliver the service
- Why you need the information
- What you'll do with the information
- Which users need to give you the information
- How you'll check that the information is accurate
- How to keep the information up-to-date and secure

A great way of filtering out unnecessary questions is to add questions only if they have a purpose and you can't derive the answers some other way or ask them later. That may require more time from you initially, but it reduces effort for your users, which results in increased completion rates. Kathryn Whitenton emphasizes the business case for keeping forms short in "Website Forms Usability: Top 10 Recommendations", stating that every time you cut a field or question from a form, you increase the form's conversion rate.

Use the right field for a given purpose

Once you've narrowed down the questions you need to ask, decide how you want to ask them. Use a `textarea` for long answers and the `input` field for short ones. Set the type of the input field to `text`, `email`, `tel`, or `url`, depending on the question. On desktop, it may not make a difference, but on mobile the type of the input field affects the keyboard layout, making typing easier. In Figure 9-4, you can see the difference between keyboard layouts for text, email, telephone, and URL input on an Android device.

Use radio buttons for single-choice questions, as shown in Example 9-1. You can also use the `<select>` element, but one of its disadvantages is that it hides choices by default. On the other hand, it can be helpful if you know that users can benefit from accessing options by typing. For multiple-choice, use checkboxes.

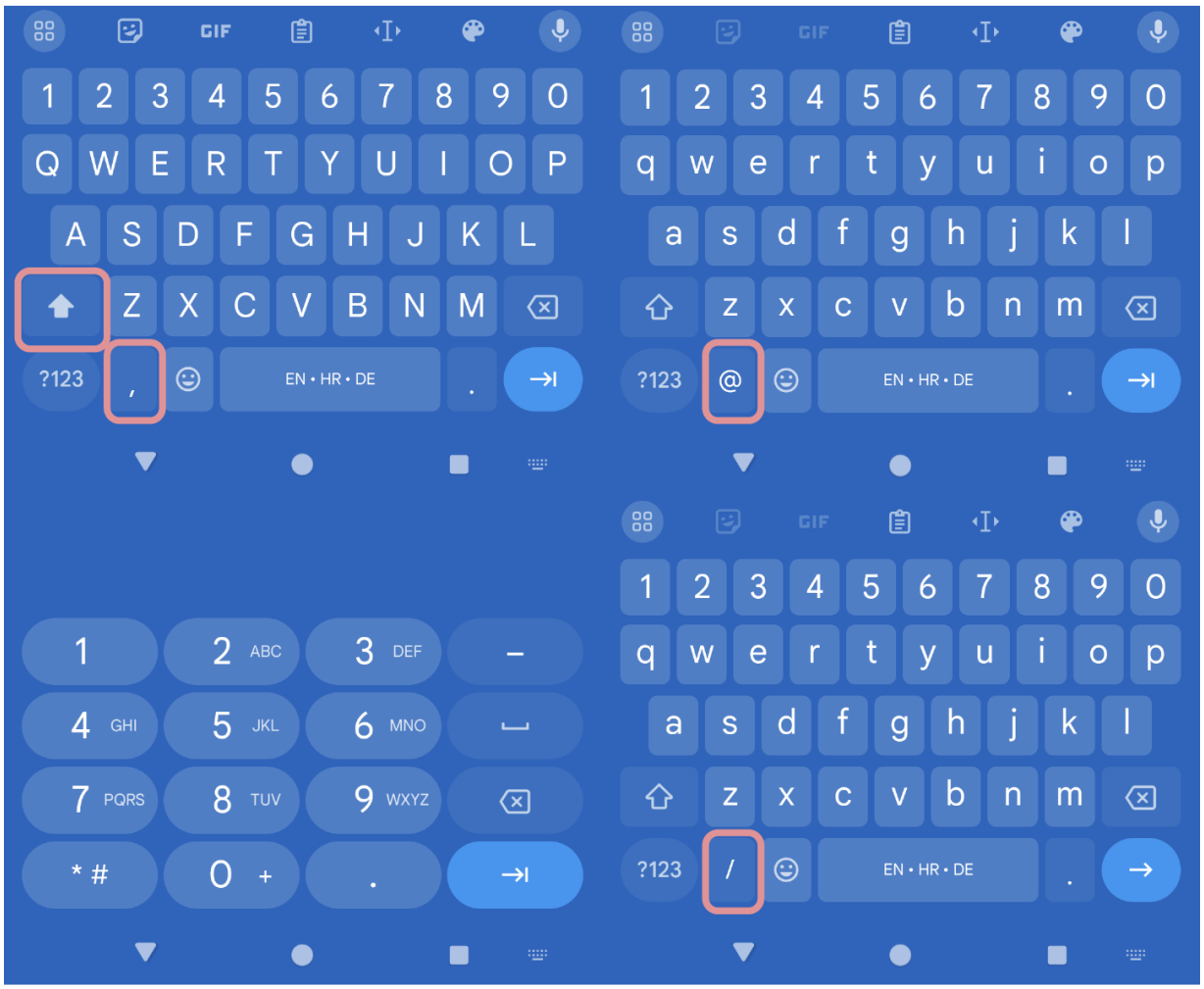


Figure 9-4. Keyboard layouts for different input types (top: text, email; bottom: tel, url)

Use native form elements

There are some exceptions, but generally, most native HTML elements are accessible by default. That's also true for most form elements. They come with a lot of features and shortcuts out of the box. Trying to re-create those without missing important details is hard. You can make a `div` look like a

button with a few lines of CSS, but to make it behave like a native button, it needs:

- A `button` role
- To be focusable
- Focus styles
- A `keydown` event listening for the Enter key
- A `keyup` event listening for the Space key

And that's just a simple button. The list of features for a `select` element is much longer. If possible, try to avoid custom solutions and use native HTML elements.

Inform users and give them control

Provide users with the information they need to fill out your form, and don't force them to figure out what will happen if they perform certain actions. For instance:

- If they need specific information to complete the form that they may not have at hand, tell them to prepare it beforehand.
- In a multistep form, tell them what will happen when they proceed to the next step. Allow them to jump back and forth between steps and make changes, and don't disable buttons (see Recipe 4.5) or other form elements.

- Describe fields, especially if they cause a change of context. Inform users what will change and how.
- Allow users to correct mistakes.
- Summarize the entered data before you submit it.

There's much more advice to give, but again, web forms are an extensive topic. The following recipes focus on practical implementations and include some of the suggestions in this recipe.

9.2 Identify Form Elements

Problem

Form elements need visual labels, or at least accessible names. If you don't provide this essential feature, screen reader users can't identify the purpose of a field or will have a hard time doing it, depending on the type and size of the form.

If you provide a label, it's crucial to make it clearly visible and place it close to its corresponding field. If you don't, understanding its purpose can be troublesome, especially for people with cognitive disabilities or low vision.

Solution

Provide a (visible) label for form controls. In most cases, it's best to provide an accessible name for form fields and also show the label, as shown in Examples 9-2 and 9-3.

Example 9-2. Labeling an input field using the `for` and `id` attributes

```
<label for="username">Username</label>
<input type="text" id="username" autocomplete="u
```

Example 9-3. Labeling an input field by nesting it inside a `label` element

```
<label>
  Username
  <input type="text" autocomplete="username">
</label>
```

If your elements don't have a visible label, you can create a reference to an existing element, as shown in Examples 9-4 and 9-5.

Example 9-4. A search field with no visual label, labled by a button

```
<header>
  <form>
    <input type="text" aria-labelledby="btn_sear
    <button id="btn_search">Search</button>
  </form>
</header>
```

Example 9-5. Labeling multiple elements

```
<table>
  <caption>Players</caption>

  <thead>
    <tr>
      <th id="username">Username</th>
      <th id="name">Name</th>
      <th id="level">Level</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td><input type="text" aria-labelledby="us
      <td><input type="text" aria-labelledby="na
      <td><input type="text" aria-labelledby="le
    </tr>
    <tr>
      <td><input type="text" aria-labelledby="us
```

```
        <td><input type="text" aria-labelledby="na
        <td><input type="text" aria-labelledby="le
      </tr>
    </tbody>
  </table>
```

Discussion

According to the WebAIM Million 2024 report, 48.6% of tested websites contained form elements with missing labels. That number is alarmingly high, considering how severe the impact is on users. It's awful because the test doesn't just include elements with no visual label, but elements with no accessible name *at all*. For screen reader users, it's impossible to identify a form element simply by accessing it because all they get is something like "edit, blank."

Inaccessible Gallery tries to visualize that and the other most common errors in the WebAIM Million report. In Figure 9-5, you can see how the experience "looks" for a screen reader user when there's no accessible name or label. It's impossible to guess which information you must enter to complete the form.

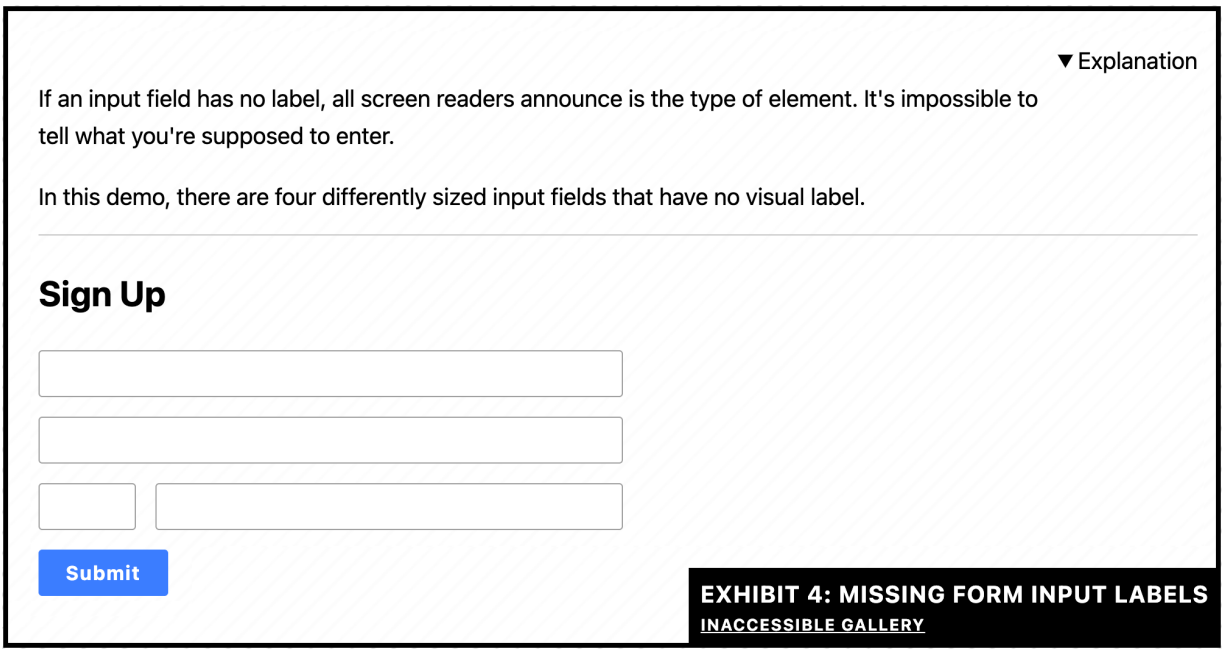


Figure 9-5. A visualization of inaccessible input fields (source: Inaccessible Gallery)

Labeling

The best thing you can do for all users is to label all form elements. Your label should give a clear and concise description of the purpose of the field and establish a clear visual and programmatic connection. In Example 9-2, you can see that the label comes directly before the field in the DOM, and the label is concise and descriptive. The two elements are connected through the `for` and `id` attributes. Instead of using explicit labeling, you can also use implicit labeling, as shown in Example 9-3. The biggest downside of that technique is that some voice-control software doesn't compute the accessible name correctly. I recommend using explicit labeling, if possible.

Labels should always be visible because you don't want your users to have to guess what they should enter. There are occasional exceptions, though. If the purpose of a form field is clear from context, you can omit it. The position, styling, and label or styling of the form control sometimes provides enough information for users to understand the purpose. Example 9-4 shows a search input field inside a header. It has no visual label, but the button provides an accessible name you can reuse with `aria-labelledby`. Usually, that's sufficient because users understand the purpose of the single input field placed in a search region.

Another exception is form controls inside a table, as shown in Figure 9-6. The input fields in the table shown in Example 9-5 are visually labeled by their corresponding table header. To establish a semantic connection, you also use the `aria-labelledby` attribute on each input field with a reference to the column header.

Players

Username	Name	Level
panierer	Peter	21

Figure 9-6. A table with input fields in each cell

Label position

According to eye-tracking tests conducted by Matteo Penzo, placing labels above form controls, as illustrated in Figure 9-7, works best in most cases because it's the best way of maintaining a close and distinct visual relationship. Left-aligned labels placed to the left (in left-to-right languages) impose a heavy cognitive workload on users, according to Penzo. The WAI emphasizes those recommendations by stating that placing labels above fields reduces horizontal scrolling for people with low vision who zoom into interfaces.

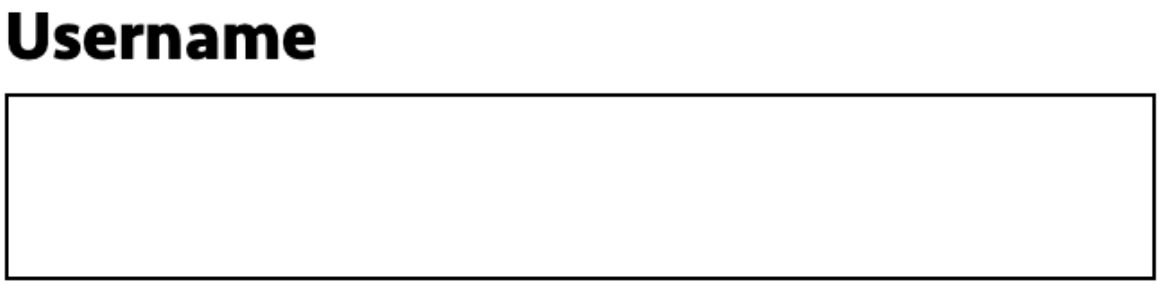


Figure 9-7. A label placed above the input field

That doesn't apply to radio buttons and checkboxes, where horizontal placement is better.

Placeholders and float patterns

Designers and developers like to use the `placeholder` attribute instead of labels (see Figure 9-8) because its value is placed inside each field, which saves space. However, if you don't use a `label`, `aria-label`, or other ways of labeling an element, browsers fall back to the `placeholder` attribute.

Figure 9-8. Bad practice: the placeholder text serves as the label of the input field

From a semantic perspective, this is a safe approach, but according to Adam Silver in *Form Design Patterns*, there are good reasons to avoid that pattern:

- The placeholder text disappears when users type, which increases cognitive load because users have to remember what they're supposed to put in the field or even how to fill it.
- The browser's autofill feature may prepopulate fields, meaning users can confirm only that the entered data is correct by removing the text and entering it again.
- Placeholder text is light gray by default, which usually doesn't provide enough contrast against the background for users with low vision.
- Long placeholder text may get cut off.

- Users often mistake placeholder text for a value. Tests by Nielsen/Norman confirm that “people with cognitive disabilities tend to have issues understanding placeholder text because they think it is pre-populated text and will try to submit the form without entering their specific information.”

A work-around for some of the issues is the float label pattern (see Figure 9-9), initially designed by Matt Smith, where the placeholder stays in the field when users start typing but gets smaller and moves to the top left corner. It’s arguably better than using placeholders alone, but there are still issues: they need space to move into, and small text is hard to read.

Figure 9-9. Float label pattern: the label moves into the top left corner as soon as the user starts typing

The float label pattern sounds like a good solution, but it mitigates only some issues and takes a lot of effort, when you could just put labels above the fields without any of the problems previously listed.

The placeholder attribute is also unsuitable for help text and other critical field descriptions. You can learn more about

providing help text and instructions in Recipe 9.3.

9.3 Describe Form Fields

Problem

Recipe 9.1 and Recipe 9.2 established three rules:

- Only ask a question if you need to.
- Pick the right element for a task.
- Identify each element properly by labeling it.

Still, that might not be enough for people to fully understand the purpose of a field or how to fill it out. Users interpret labels differently; they may not know the format you expect or the consequences of the choices they make. That makes filling out forms tedious and sensitive to errors.

Solution

Describe forms and fields by providing additional information, as shown in Examples 9-6, 9-7, and 9-8.

Example 9-6. Overall instructions

```
<ul>
```

```
    <li>All fields marked “required” must be compl
    <li>Dates should all be typed in the format dd
    <li>You need a copy of your passport.</li>
  </ul>

  <form>
    <label for="bday">Birthday (dd.mm.yyyy)</label>
    <input type="text" id="bday" autocomplete="bda
    …
  </form>
```

Example 9-7. Inline instructions in the label

```
  <label for="bday">Birthday (dd.mm.yyyy)</label>
  <input type="text" id="bday" autocomplete="bday">
```

Example 9-8. Inline instructions following the input field

```
  <label for="password">Password</label>
  <input type="text" id="password" autocomplete="n
  aria-describedby="password-hint">
  <p id="password-hint">At least 8 characters, upp
```

Discussion

Some forms are straightforward and don't need additional description. A simple login form that asks users for their email and password can provide all necessary information if they're clearly labeled.

Suppose you ask for a username or customer ID instead. Users may not remember their username or where to find their ID. Instructions can help them fill out the form. The gov.uk service manual "Designing good questions" suggests using help text or instructions to explain things like:

- Legal jargon
- Where to find obscure information
- In what format the information should be given
- What you'll do with personal information
- The consequences of making one choice over another

There are several places where you can put instructions.

You can provide overall instructions before or at the beginning of the form, as shown in Example 9-6. That's a good place to tell users how the form works and mention any specific prerequisites, such as if they need to upload copies of legal documents.

You can use the label element for short inline descriptions of form fields, as shown in Example 9-7. The advantage is that it's tied to the form element and will be announced no matter how the user accesses it. That works only for short descriptions, though, because you don't want to clutter labels.

You can put longer inline instructions after the form field and use `aria-describedby` to connect them with the element they're describing, as shown in Example 9-8. In his book *Form Design Patterns*, Adam Silver suggests putting form hints between the label and input field to emphasize the label's relation to the field and keep autocomplete suggestions from hiding it (see Figure 9-10).

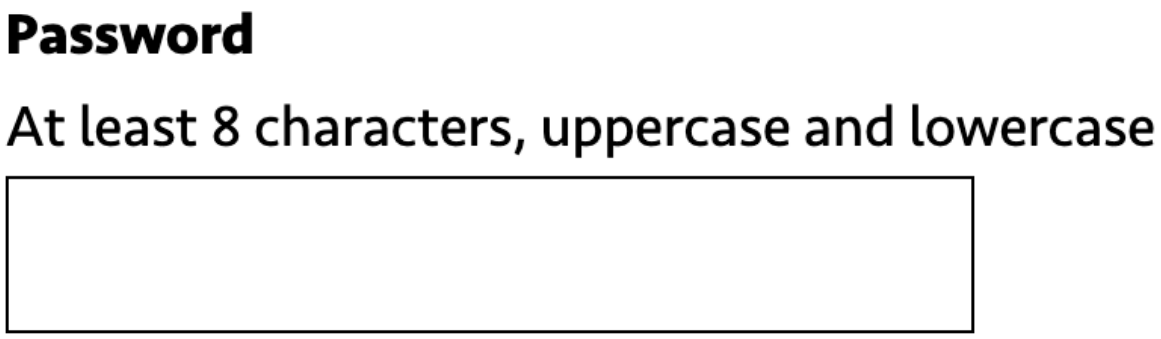


Figure 9-10. Instructions placed close to the label and input element

All these options are great because the descriptions are visible at any time, and you usually have enough space to display them. Websites often use the `placeholder` attribute to

provide descriptions and instructions, but it's unsuitable for providing important help text and hints because of the many reasons described in Recipe 9.2.

There's another attribute in HTML that's much more useful and allows you to provide more details about the type of information you're collecting. The `autocomplete` attribute specifies form fields in greater detail and simplifies filling out forms. It can hint to the user agent how and whether to prefill input fields based on earlier input. Example 9-7 shows how to specify that the text input field is a birthday field. If you've entered your birthday before and the browser saved that information, it will prefill the field for you. Sometimes it does that automatically by factoring in the field's label or other attributes. Sometimes it doesn't—and that's where the `autocomplete` attribute can be helpful.

According to the WAI, the attribute can be a powerful tool for people with language and memory-related disabilities or disabilities that affect executive function and decision-making because they don't need to remember the information. People with a motor disability also benefit from reducing the need for manual input when filling out forms.

The attribute has many options, like `current-password`, `country`, `postal-code`, or `username`, and it supports multiple values like `shipping street-address`. You can find a full list in the specification.

9.4 Highlight Erroneous Fields

Problem

When a user doesn't fill in all required fields or doesn't do it correctly, you must give them feedback. To enable them to fix the errors, they need to know what went wrong and where. Otherwise, they may not be able to complete the form.

Solution

Different places and techniques can help communicate errors. You should place error messages close to the affected fields, as shown in Examples 9-9 and 9-10.

Example 9-9. Inline error message

```
<style>
```

```
    .error {
      color: #D52A2A;
    }
  </style>

  <label for="email">Your e-mail address</label>
  <input type="email" id="email" required value="o
         aria-invalid="true" aria-describedby="err
  <div id="error" class="error">
    <svg viewBox="0 0 640 640" width="16" fill="cu
      <path d="M640 128 512 0 320 192 128 0 0 128l
      512l128 128 192-192 192 192 128-128-192-192
    </svg>

    Please enter a valid e-mail address
  </div>
```

- ❶ `aria-invalid="true"` marks the field as invalid and `aria-describedby` connects it to the corresponding error message using the message's ID as the value.

Example 9-10. Inline instructions and error message on an input field

```
  <label for="password">Password</label>
  <div id="password-hint">At least 8 characters, u
```

```
<input type="text" id="password"
       autocomplete="new-password"
       aria-invalid="true"
       aria-describedby="password-hint password-
<div id="password-error" class="error">
  <svg viewBox="0 0 640 640" width="16" fill="cu
    <path d="M640 128 512 0 320 192 128 0 0 128l
    512l128 128 192-192 192 192 128-128-192-192
  </svg>
  Passwords must contain at least one number.
</div>
```

- `aria-describedby` referencing the hint and the error message.

If there are many errors, you can summarize them at the beginning of the form, as shown in Example 9-11.

Example 9-11. Error summary

```
<div role="region" aria-labelledby="error_headin
  <h2 id="error_heading">2 issues found</h2>

  <ul>
    <li><a href="#email">Please enter a valid e-
    <li><a href="#password">Passwords must conta
  </ul>
```

```
</div>

<form>
  <label for="email">Your e-mail address</label>
  <input type="email"
         id="email"
         autocomplete="email"
         required
         aria-invalid="true"
         aria-describedby="email-error"
         value="philippa@">
  <div id="email-error" class="error">
    <svg viewBox="0 0 640 640" width="16" fill="
      <path d="M640 128 512 0 320 192 128 0 0 12
      512l128 128 192-192 192 192 128-128-192-19
    </svg>
    Please enter a valid e-mail address.
  </div>

  <label for="password">Password</label>
  <input type="password"
         id="password"
         autocomplete="new-password"
         required
         aria-invalid="true"
         aria-describedby="password-error">

  <div id="password-error" class="error">
    <svg viewBox="0 0 640 640" width="16" fill="
```

```
        <path d="M640 128 512 0 320 192 128 0 0 12
        512l128 128 192-192 192 192 128-128-192-19
      </svg>
      Passwords must contain at least one number.
    </div>
  </form>
```

It can be helpful if the page's title also reflects the current status, as shown in Example 9-12.

Example 9-12. Number of errors in the title element

```
  <title>2 errors - Sign Up - Johanna's Toy Store<
```

Discussion

Users will eventually make mistakes, no matter how good your labels and descriptions are. Disabled users may be more likely to make them. For instance, a user with a motor disability may accidentally hit keys, or someone with a reading disability may mix up numbers and letters. They also may have a more challenging time detecting mistakes and recovering from them.

When that happens, you must provide users with the best possible assistance to fix those issues.

There are several general rules you should follow:

- Don’t rely on color alone to communicate.
- Inform users that something went wrong.
- Write clear and meaningful error messages.
- Provide instructions on how to fix errors.
- Allow users to access erroneous fields.
- Don’t rely on native form validation.
- Allow users to (re)submit the form.

In her article “A Guide To Accessible Form Validation”, Sandrina Pereira explains that using color alone is insufficient to communicate issues because people perceive color differently. Blind users need a semantic cue that something is wrong with the entered data in a field. In Figure 9-11, you see a combination of color and icon to indicate an error. In addition, you must also identify those fields semantically by using the `aria-invalid` attribute, as shown in Example 9-9.

Figure 9-11. The *x* icon supports the error message and the color in indicating an error

Error reporting

If there's only one issue or if the form is short, you can put focus on the first erroneous field. If you use the pattern in Example 9-9, a screen reader announces that the field contains invalid data, followed by the error message, connected via `aria-describedby`. The `aria-describedby` attribute accepts a single ID or a list of space-separated IDs. You can connect a hint and an error message to a field simultaneously, as shown in Example 9-10.

If there are multiple issues, look at how you're submitting the data. If you're using server-side rendering, listing all issues in a region at the beginning of the form and focusing it can be a good solution. The region in Example 9-11 displays the number of errors and announces it when it receives focus. It also lists linked error messages, allowing users to get an overview of all

errors and access them directly by linking them to their corresponding field. Screen reader users can access the region at any time, since `role="region"` turns it into a landmark (you can learn more about region landmarks in Recipe 2.4). You can also update the page title to reflect the current state of the form, as shown in Example 9-12.

If you're using client-side scripting, you can do the same or use a live region to inform users about the errors when they submit the form. See Recipe 3.7 to learn more about live regions. When you work with live regions, it's good to know that there is a bug/feature in Chromium-based browsers on Windows that automatically announces content added to elements referenced via `aria-describedby`.

If you're considering using the default validation in HTML, keep in mind that there are several downsides to using native validation, as Gerardo Rodriguez notes in his article "Progressively Enhanced Form Validation". Here are some examples:

- You cannot customize the styling of error messages.
- Some form controls cannot be validated.
- There are challenges in styling erroneous fields in some browsers.

- The error messages may not always be clear or provide a helpful suggestion.
- The error-message bubble text doesn't always resize when you zoom the page.
- Error messages are not correctly associated with the corresponding field.

You can use the native API for validation, as described in Rodriguez's article, but it's better to provide your own styling and markup.

Error messages

In *Form Design Patterns*, Adam Silver highlights the importance of well-crafted error messages and explains how to create messages that provide clarity in as few words as possible:

- Be concise, but don't omit words at the cost of clarity.
- Be consistent by using the same tone and punctuation throughout.
- Avoid pleasantries like "Please" because they imply choice.
- Don't use generic messages; be specific. Instead of saying, "There's an error," explain what went wrong.
- Use plain natural language. Avoid jargon like *invalid*, *forbidden*, and *mandatory*.

- Use the active voice: “Enter your name” instead of “A name must be entered.”
- Let users know what’s gone wrong and how to fix it, but don’t blame them: “Enter your email” instead of “You didn’t enter an email.”

In his talk “Is Design Metrically Opposed?” Jared Spool says that it takes one line of code to take a phone number and strip out all the dashes, parentheses, and spaces, but it takes 10 lines of code to write an error message informing users about their mistake. Silver advises following Postel’s Law and forgiving trivial mistakes by writing code that corrects them. That, combined with well-designed labels, descriptions, and error messages, makes filling out forms much more accessible.

NOTE

Postel’s Law, named after Jon Postel, states that you should be conservative in what you send but be liberal in what you accept.

9.5 Group Fields in a Form

Problem

If your form contains many similar questions or groups of fields of the same type that belong to different questions, users may have trouble telling them apart and understanding how they're related. Screen reader users may not understand the purpose or affiliation of a field. Sighted users may not, either, if the form is poorly designed.

Solution

Group related fields together using the `fieldset` and `legend` elements, as shown in Examples 9-13 and 9-14.

Example 9-13. A `fieldset` groups radio buttons

```
<fieldset>
  <legend>Do you have pets?</legend>

  <input type="radio" id="pets_yes" name="pets">
  <label for="pets_yes">Yes</label>

  <input type="radio" id="pets_no" name="pets">
  <label for="pets_no">No</label>
</fieldset>
```

Example 9-14. A `fieldset` groups similar questions asked in the same form

```
<fieldset>
  <legend>Billing details</legend>
  <label for="billing_name" autocomplete="name">
  <input type="text" id="billing_name" name="bil

  <label for="billing_address">Address</label>
  <input type="text" id="billing_address" name="
      autocomplete="billing street-address">
</fieldset>

<fieldset>
  <legend>Shipping details</legend>
  <label for="shipping_name" autocomplete="name"
  <input type="text" id="shipping_name" name="sh

  <label for="shipping_address">Address</label>
  <input type="text" id="shipping_address" name=
      autocomplete="shipping street-address">
</fieldset>
```

Discussion

Let's say you have a form, like the one in Example 9-15, that asks users multiple yes-or-no questions.

Example 9-15. Bad practice: Ungrouped radio buttons

```
<strong>Do you have pets?</strong>
<input type="radio" id="pets_yes" name="pets">
<label for="pets_yes">Yes</label>

<input type="radio" id="pets_no" name="pets">
<label for="pets_no">No</label>
```

If a screen reader user uses the virtual cursor, the software first announces, *"Do you have pets?"* followed by the options *"Yes, radio button, checked, 1 of 2"* and *"No, radio button, checked, 2 of 2."* (The announcement differs across different screen readers.) It should be clear that the two options belong to the question. However, the virtual cursor is not the most convenient way to navigate a form. Using the `Tab` key is much more efficient, but if you do, you'll only hear the answer, *"Yes, radio button, checked, 1 of 2,"* not the question. That's because the radio buttons are semantically unrelated to the strong element that contains the question.

To connect the radio buttons with the question, you need to group them. You can use the `fieldset` and `legend` elements, as shown in Example 9-13. That establishes a connection between the radio buttons and the corresponding question. The

screen reader announces something like *“Do you have pets?, grouping. Yes, radio button, not checked, 1 of 2.”*

The `fieldset` element is an excellent tool for making related form controls more understandable. Screen reader users can identify them more easily, and its distinguishable design (see Figure 9-12) helps sighted users understand relationships between fields. In long forms, it can also help users focus on smaller and more manageable groups rather than the entire form.

Figure 9-12. A `fieldset` grouping radio buttons

Using a `fieldset` is always a good idea for groups of radio buttons or checkboxes. It can also make sense for other types of form controls, especially if you’re asking the same or similar questions multiple times in the same form, as shown in Example 9-14. The `fieldset` and `legend` associate each name and address with the corresponding group.

Dos and don’ts

You don't have to put *every* form element in a `fieldset`; use it only when a higher-level label is necessary. You don't need grouping for single checkboxes or radio buttons that make sense from their labels alone.

Nesting fieldsets is possible, but you should avoid it because it can cause odd screen reader behavior. The software may not understand when one fieldset ends and the next starts.

Always provide a legend. Otherwise, using the fieldset will make little sense. The `legend` element should be the first child of the fieldset and must not be nested. In Example 9-16, you can see a broken fieldset. The legend doesn't provide an accessible name for the `fieldset` because the `legend` is nested in a `div`.

Example 9-16. A broken fieldset: The legend doesn't provide an accessible name for the fieldset

```
<fieldset>
  <div>
    <legend>Do you have pets?</legend>
  </div>

  <input type="radio" id="pets_yes" name="pets">
  <label for="pets_yes">Yes</label>
```

```
    <input type="radio" id="pets_no" name="pets">
    <label for="pets_no">No</label>
  </fieldset>
```

Depending on the screen reader and how users are accessing the group, the software will announce the legend when users enter it, when they access the first option, or with every option. To avoid redundancy, keep the legend short but descriptive.

9.6 Split Forms into Steps

Problem

If a form contains many questions, it can become overwhelming and stressful for people with cognitive disabilities to understand and complete it.

Solution

Split long forms into multiple smaller forms that constitute a series of logical steps. In each step, communicate the current step within the process, as shown in Examples 9-17, 9-18, and 9-19.

Example 9-17. A progress indicator highlighting the current step

```
<ol>
  <li>
    <a href="/checkout/shipping.html">
      Shipping address
    </a>
  </li>
  <li aria-current="step">
    <span>
      Payment
    </span>
  </li>
  <li>
    <span>
      Review order
    </span>
  </li>
  <li>
    <span>
      Finish
    </span>
  </li>
</ol>
```

Example 9-18. The title of the page indicating the current step

```
  <title>Step 2 of 4: Payment - Checkout - Johanna
```

Example 9-19. The main heading of the page indicating the current step

```
  <h1>Shipping Payment (Step 2 of 4)</h1>
```

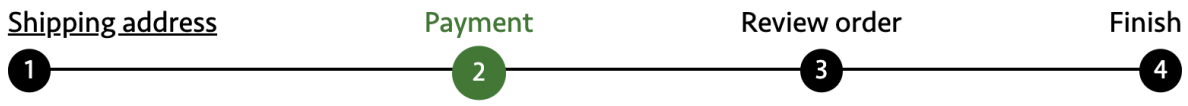
Discussion

Long forms can be intimidating and complex. The WAI suggests dividing long forms into multiple smaller forms to make it easier for users to complete them and make the experience less daunting. The team at gov.uk even suggests asking only one question or providing users with only one piece of information per page. That doesn't always work, but the "one thing per page" approach is a good starting point. It reduces cognitive load, helping people understand what you're asking for. They can focus on one task at a time without being distracted by unrelated questions. Another benefit is that fixing form errors becomes more manageable, reducing the chances of users giving up.

Some people need more time to fill out forms. Multiple steps make it easier to save users' answers, allowing them to

complete the forms later. Adam Silver lists more benefits of this approach in "Better Form Design: One Thing Per Page (Case Study)".

When splitting a form, organize its elements in logical groups. In the checkout page of an online shop, that could be shipping address, payment, review, and confirmation. As shown in Figure 9-13, a step-by-step indicator can help users orient themselves. In Example 9-17, you can see that the ordered list communicates the number of steps. Completed steps are linked, and the `aria-current` attribute highlights the current step. Users should always know how much they've completed and how much will follow.



Payment (Step 2 of 4)

Figure 9-13. A progress indicator showing the previous, current, and remaining steps

The first step of the process should inform how many steps the form has and about any needed preparations or information. Figure 9-14 shows the *gov.uk* form to check if a vehicle is taxed. The intro page of the form informs you how long it takes them

to process the application and that you'll need your car's registration number and a reference number to fill it out. That makes the experience less exhausting and allows users to finish the form quicker.

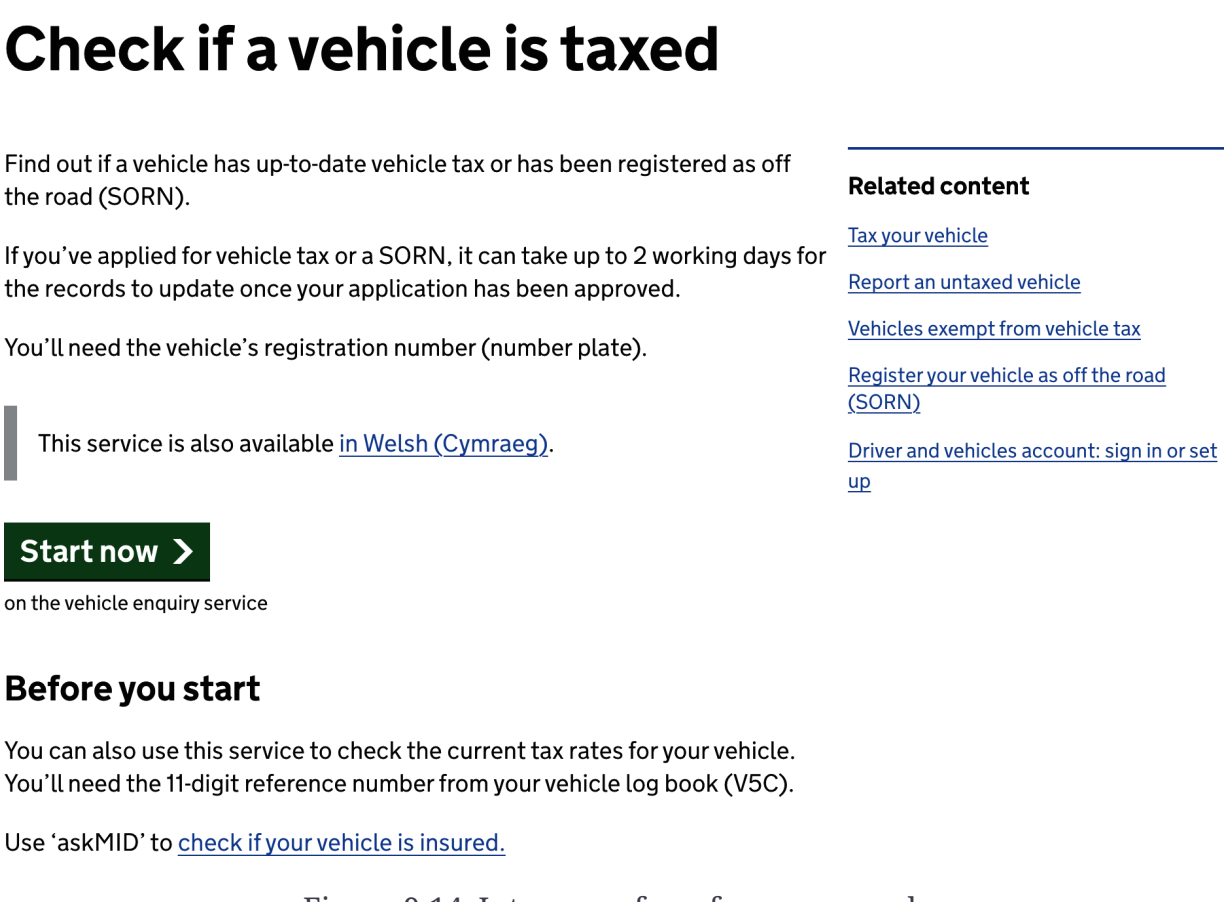


Figure 9-14. Intro page for a form on gov.uk

Designing a good progress indicator can be challenging, primarily because of the limited space on mobile devices. If you have difficulty getting it right or learn that your users don't notice it or find it distracting, other techniques you can use

include the `title` element (see Example 9-18) and the `h1` of the page (see Example 9-19).

Dividing a form into multiple steps shouldn't make it harder to use. Allow users to go back to previous steps to review and change the entered data. Link completed steps in the step-by-step indicator and provide a back button at the beginning or end of each step.

If your page includes overall instructions for filling out the form, repeat them on every page so that users don't have to go back to the first step.

Some people need more time than others. Don't set a time limit to fill out the form. If a time limit is required, provide a feature that allows the user to adjust it, to minimize the risk that they will lose their work.

Give users the option to review and correct their data before they submit a form. Confirm their submissions when they are done.

See Also

- "web.dev Lean Accessibility—Forms" by web.dev

- “Understanding SC 1.3.5: Identify Input Purpose (Level AA)” by WAI
- “Understanding SC 3.2.1: On Focus (Level A)” by WAI
- “Creating Accessible Forms” by WebAIM
- “Designing good questions” by gov.uk
- “There is a lot more to autocomplete than you think” by Saptak Sengupta
- “Create accessible forms” by The A11y Project
- “Using the fieldset and legend elements” by Léonie Watson
- “Fieldsets, Legends and Screen Readers again” by Steve Faulkner
- “Forms Tutorial: Grouping Controls” by WAI

Chapter 10. Filtering Data

Online shops, marketplaces, real estate websites, and large blogs often have one thing in common: a lot of data the user can access. Working with large datasets can quickly become daunting and overwhelming. Filters allow users to narrow down results, making finding what they're looking for easier. At the same time, they can also increase the complexity of your interface. When you create a filter form, it's crucial to keep it as simple as possible so that users can apply their settings quickly and efficiently. No matter how people access the web, they must always know which filters are active and whether they've been applied.

10.1 Create a Form

Problem

When you present users with a list of hundreds or thousands of items, it can be difficult for them to find what they're looking for. If the list contains interactive elements, navigating through it can become physically demanding for people who use an assistive technology like a switch device.

Solution

Allow users to filter out irrelevant results. Example 10-1 shows an exemplary filter in HTML.

Example 10-1. A filter form with various types of form elements

```
<form role="form" aria-label="Filter" id="filter
  <label for="artist">Artist</label>
  <select id="artist" name="artist">
    <option value="">All</option>
    <option>AFI</option>
    <option>Absolute Beginner</option>
    <option>Akne Kid Joe</option>
    <option>Bad Religion</option>
    <option>Beastie Boys</option>
    <option>Bilderbuch</option>
    <option>Billy Joel</option>
    <option>Bring me the Horizon</option>
    <option>Dead Kennedys</option>
    …
  </select>

  <fieldset>
    <legend>Country</legend>

    <input type="checkbox" name="country" id="co
```

```
    <label for="country_at">Austria</label>

    <input type="checkbox" name="country" id="co
    <label for="country_ca">Canada</label>

    <input type="checkbox" name="country" id="co
    <label for="country_us">USA</label>

    <input type="checkbox" name="country" id="co
    <label for="country_de">Germany</label>

    <input type="checkbox" name="country" id="co
    <label for="country_uk">United Kingdom</labe
  </fieldset>

  <fieldset>
    <legend>Shipping</legend>

    <input type="radio" name="shipping" id="ship
    <label for="shipping_eu">Europe</label>

    <input type="radio" name="shipping" id="ship
    <label for="shipping_world">Worldwide</label

    <input type="radio" name="shipping" id="ship
    <label for="shipping_us">USA</label>
  </fieldset>
```

```
    <button>Search</button>
  </form>
```

Discussion

A filter form helps users narrow down the results in a large dataset, but it also adds complexity. Before you add a filter, consider whether you need it. They're helpful only when the user's search is likely to return a vast set of results.

In Example 10-1, you can see a filter form on "Bob's Records." Let's break it down.

Form elements

Bob's form uses different types of elements, depending on the use case:

- For instance, users can search records by artist. Since there are potentially hundreds of items in that list, showing only one at a time using a `select` element makes sense.
- The filter lists the artist's country of origin with checkboxes because there are only five options, and the user can select either none, all, or some countries.

- There are only three shipping options, and you can only select one at a time. Radio buttons are a good choice for that.

Of course, there are also other ways of presenting the same options. Whatever you choose, ensure you pick the most straightforward solution for the user:

- Pick form elements according to their requirements.
- Use native HTML form elements because they usually have the best baseline accessibility.
- Stick to familiar patterns, and don't obscure the styling of form elements. For example, don't make radio buttons look like ordinary buttons.
- Be consistent to help users build familiarity and understand the site.

Grouping

Grouping controls using the `<fieldset>` element can make forms more understandable because it makes the controls easier to identify and associate. They inform users about the grouping both visually and semantically, as illustrated in Figure 10-1.

Not every form element needs to be in a group. The country checkboxes in Example 10-1 belong together. The form groups

them using a `fieldset` element, and the `legend` gives them a group label. The artist widget, on the other hand, stands for itself and needs no additional grouping.

Figure 10-1. A form with two groups of controls

Form landmark

The `role="form"` on the `<form>` turns a simple form into a landmark (for details about form landmarks, see Recipe 2.2), and the `aria-label` attribute provides an accessible name. That can be handy for screen reader users because it allows them to access the filter using shortcuts without interacting with the rest of the page.

Form submission

In his book *Form Design Patterns*, Adam Silver explains that there are two ways of letting users filter: using one filter at a time (interactive filters) or selecting multiple filters at once (batch filtering).

Interactive filters

> Interactive filters update when the user clicks a filter. The user gets immediate results and feedback. One disadvantage is that it slows down the filtering process, because the whole page, or parts of the page, must reload every time the user clicks.

Batch filters

> Batch filters let users set several options before submitting the form. That approach is faster, but combining filters without immediate feedback can yield zero results.

Example 10-1 uses batch filtering to give users more control. Also, selecting form elements sets certain expectations: When clicking radio buttons or checkboxes, users usually don't expect an immediate response.

10.2 Filter the Data

Problem

Especially in interfaces that rely heavily on client-side scripting, you must inform users about changes to the document. If you don't, screen reader users might not know that filtering worked as expected. On top of that, users must be able to access search results easily.

Solution

Give users easy access to filtering and inform them about changes using live regions or focus management, as shown in Examples 10-2 and 10-3.

Example 10-2. An ordered list within a region

```
<div role="region" aria-labelledby="results_head
  <h2 id="results_heading">Results</h2>
  <div role="status">Showing 40 of 40 records</d

  <ol class="list"> ❸
    <li>
      <strong>Joy as an Act of Resistance (2018)
```

```
        Idles
      </li>
      <li>
        <strong>Licensed to Ill (1986)</strong><br
        Beastie Boys
      </li>
      <li>
        <strong>Paul's Boutique (1989)</strong><br
        Beastie Boys
      </li>

      …
    </ol>
  </div>
```

❶ An ARIA region labeled by its parent heading. The negative `tabindex` makes it focusable via JavaScript.

❷ An ARIA live region.

❸ An ordered list containing the results.

Example 10-3. A simple filter script

```
const form = document.getElementById("filter");
const results = document.getElementById("results
const list = results.querySelector("ol");
```

```
const liveRegion = document.querySelector("[role=
let records,
filtered;

function finishQuery() {
  ❹
}

function showResults() { ❸
  list.innerHTML = "";
  for (let i = 0; i < filtered.length; i++) {
    const record = filtered[i];
    const item = document.createElement("li");
    const title = document.createElement("strong
    title.textContent = `${record.title} (${reco
    item.append(title, record.artist);
    list.append(item);
  }
}

function filterForm(e) { ❷
  e.preventDefault();

  const formData = new FormData(form);

  filtered = records.filter((record) => {
    const artist = formData.get("artist");
    const countries = formData.getAll("country")
```

```
      const shipping = formData.getAll("shipping")

      if (artist && record.artist !== artist) {
        return;
      }

      if (countries.length && !countries.includes(
        return;
      }

      if (shipping.length && !shipping.includes(re
        return;
      }

      return true;
    });

    showResults();
    finishQuery();
}

async function getRecords() {
    /*
      The JSON files looks like this:
      [
        {
          "artist": "Absolute Beginner",
          "title": "Bambule",
```

```
        "year": 1998,
        "country": "DE",
        "format": ["LP", "CD"],
        "shipping": "eu"
      },
      {
        "artist": "AFI",
        "title": "Very proud of ya",
        "year": 1996,
        "country": "US",
        "format": ["LP", "CD"],
        "shipping": "us"
      }]
  */
  const response = await fetch("/assets/data/rec
  return await response.json();
}

getRecords().then((data) => { ❶
  records = data;
  filtered = data;
  form.addEventListener("submit", filterForm);
});
```

❶ Fetches all records up front and attaches an event listener to the form.

❷ Simple filter functionality that narrows down results based on user input.

❸ Takes the filtered results, clears the list, and creates a new list.

❹ This is where you announce changes to the DOM. See Examples 10-4 and 10-5 for options.

Example 10-4. Option A: Focusing the region after the results are shown

```
function finishQuery() {
  results.focus()
}
```

Example 10-5. Option B: Writing the number of results to a live region

```
function finishQuery() {
  const total = records.length;
  const found = filtered.length;
  liveRegion.textContent = `Showing ${found} of 
}
```

Discussion

There are two essential aspects to client-side filtering: you want to announce changes to screen reader users and give them quick and easy access to the results.

Feedback

If you use server-side rendering and processing for your filter, the feedback users get when submitting the form is a full-page reload. Screen readers also announce the page title. That's usually enough for users to understand that the page has changed.

With client-side rendering, it's different, because when you make changes to the DOM, there's only visual feedback. Users may see the content change if the results are visible in the viewport, but you can't rely on that—for example, when the user is blind and therefore uses a screen reader, or uses a high zoom level, or when the filter form is particularly long and moves the results out of the viewport. Two common techniques for announcing page changes are focus management and live regions. You can learn more about both in Recipe 3.7.

Example 10-4 shows the first option: after filtering and repopulating the list, you focus the results region by calling the `focus()` method. For that to work, the region needs a

`tabindex="-1"`. Once you focus it, focus moves from the submit button to the region. That affects users in two ways. First, focus is no longer on the form; users can interact with the results immediately. However, they have to tab back if they want to adjust the filter again. Second, screen readers may announce the role and label of the region (for example, “Results, region”).

In Example 10-5, you output the number of results to the live region instead of focusing the results region. If the content in the live region changes, a screen reader announces it (e.g., “Showing 3 of 40 records”). Focus stays on the button, meaning users must move to the list manually.

Test with users to determine which option is best for your use case. Of course, you can also combine both solutions, but be careful with what and how much you announce. You want to communicate as little as possible and only as much as necessary. Screen reader and Braille display users are more efficient with concise and specific commands.

Before you make your decision, ask yourself two questions:

- Would it be helpful to move focus, or would it interrupt the experience?

- How much information does the user need? Is the name of the new page or region enough, or do they need more context?

Structure

The structure of the results region (Example 10-2) is pretty simple, but feature-rich. The labeled region enables screen reader users to access the results directly using landmark navigation. The heading, which also serves as the accessible name for the region, adds to the page's visual hierarchy and gives screen reader users access via heading navigation. The ordered list also comes with additional keyboard commands for screen reader users and announces the number of displayed items.

10.3 Paginate Results

Problem

Even with a filter, the list of search results, products, images, and so on, can still be overwhelmingly hard to scan and navigate. When it contains images, it can also degrade performance.

Solution

Break the results down into multiple pages as shown in Examples 10-6, 10-7, and 10-8.

Example 10-6. An ordered list within a labeled navigation

```
<nav aria-labelledby="pagination_heading" class=
  <h2 id="pagination_heading">Select page</h2>
  <ol>
    <li>
      <a href="/results/1" aria-current="page">1
    </li>
    <li>
      <a href="/results/2">2</a>
    </li>
    <li>
      <a href="/results/3">3</a>
    </li>
    <li>
      <a href="/results/4">4</a>
    </li>
    <li>
      <a href="/results/5">5</a>
    </li>
    <li>
      <a href="/results/6">6</a>
    </li>
```

```
      <li>
        <a href="/results/7">7</a>
      </li>
    </ol>
  </nav>
```

1. A labeled navigation landmark
2. Highlighting the current page

Example 10-7. Using `aria-label` to overwrite the accessible names of links (for example, “Page 2” instead of “2”)

```
  <li>
    <a href="/filter.html/2" aria-label="Page 2">2
  </li>
```

Example 10-8. Styling the pagination

```
  .pagination ol {
    display: flex;
    gap: 0.5em;
    list-style: none;

    margin: 0;
    padding: 0;
```

```
}

.pagination a {
  align-items: center;
  aspect-ratio: 1;
  border: 1px solid;
  border-radius: 50%;
  display: flex;
  justify-content: center;
  text-decoration: none;
  width: 2em;
}

.pagination a:is([aria-current="page"], :hover,
  background-color: #3c843f;
  color: #ffffff;
}
```

Discussion

Pagination is similar to filters: use it only when you need it. It adds complexity to the page and requires extra steps to get results. On the other hand, when filtering still yields hundreds of results, splitting them up can be helpful. According to David Kieras, reaching an endpoint gives users a sense of control. Unlike seemingly endless or infinite lists, with a fixed number

of items per page and pagination, users know the number of results. They can estimate how long it'll take to find what they're looking for and whether they need to refine the filter.

Besides performance benefits, which also affect accessibility, pagination makes finding and searching content easier. In Example 10-6, you can see that links are wrapped in a navigation landmark, providing shortcuts for screen reader users. It's labeled to allow for distinction from other navigations that are likely also present on the page. `aria-current="page"` highlights the currently active page semantically and provides a hook for styling in CSS, as shown in Example 10-8. The label of each link is just the page number, which should be clear enough from context. If you want to be more explicit with the accessible names of your links, you can overwrite them using `aria-label`, as shown in Example 10-7 and Figure 10-2.

If you show many results per page and those contain interactive elements, you can add a skip link at the beginning of the results region. That makes it easier for keyboard and screen reader users to reach the pagination. Alternatively, you can replicate the pagination and put it before the results. When you do that, test the entire page thoroughly with the keyboard to ensure a decent user experience.

Results

1. **Bambule (1998)**
   Absolute Beginner
2. **Very proud of ya (1996)**
   AFI
3. **All Hollows EP (1999)**
   AFI
4. **Die große Palmöllüge (2020)**
   Akne Kid Joe
5. **Suffer (1988)**
   Bad Religion
6. **Age of Unreason (2019)**
   Bad Religion
7. **Licensed to Ill (1986)**
   Beastie Boys
8. **Paul's Boutique (1989)**
   Beastie Boys
9. **Check Your Head (1992)**
   Beastie Boys
10. **Ill Communication (1994)**
    Beastie Boys

Figure 10-2. Results and pagination

10.4 Sort and Display Results

Problem

When you give users control over how to display data, you must inform them about changes and the current state of each option. If you don't, they might not understand why the interface looks or behaves differently.

Solution

Inform users about settings when they access options and when they change them. Example 10-9 provides users with two sorting options using radio buttons, and the code in Example 10-10 sorts the results and informs users about the changes.

Example 10-9. A list of sorting options

```
<fieldset id="sorting">
  <legend>Sort by</legend>


  <div>
```

```
    <input type="radio" id="sorting_artist" name=
    <label for="sorting_artist">Artist</label>
    <input type="radio" id="sorting_date" name="
    <label for="sorting_date">Date</label>
  </div>
</fieldset>
<div id="live-region-sorting" hidden>Sorted by [
```

Example 10-10. Sorting results

```
const sorting = form.querySelector("#sorting");
const sortingMessage = form.querySelector("#live

function sortRecords(type) { ❶
  function compare(a, b) {
    let fieldA = a[type];
    let fieldB = b[type];

    if (typeof fieldA === "string") {
      fieldA = fieldA.toLowerCase();
      fieldB = fieldB.toLowerCase();
    }

    if (fieldA < fieldB) {
      return -1;

    }
    if (fieldA > fieldB) {
```

```
        return 1;
      }
      return 0;
    }

    filtered.sort(compare);
  }

  sorting.addEventListener("change", (e) => {
    const type = e.target.value;
    sortRecords(type);
    showResults();
    liveRegion.textContent = sortingMessage.textCo
  });
```

1. Simple sorting logic.
2. Informs users that the list has changed.

Example 10-11 offers display options using the button element, and the code in Example 10-12 switches the display type and informs users about the changes.

Example 10-11. Offering display options

```
  <fieldset id="display">
```

```
    <legend>Display</legend>
    <button type="button" aria-pressed="true" aria
      <svg>…</svg>
    </button>
    <button type="button" aria-pressed="false" ari
      <svg>…</svg>
    </button>
  </fieldset>
  <div id="live-region-display" hidden>Showing res
```

Example 10-12. Switching between list and grid view

```
  const display = form.querySelector("#display");
  const displayMessage = form.querySelector("#live

  display.addEventListener('click', e => {
    const button = e.target.closest('button'); ❶
    const previous = display.querySelector('[aria-

    if (button) {
      button.setAttribute('aria-pressed', true); ❸
      list.classList.replace(previous.dataset.disp
      previous.removeAttribute('aria-pressed');
      const label = button.getAttribute('aria-labe
      liveRegion.textContent = displayMessage.text

    }
  })
```

```
```

❶ The selected display option

❷ The previously selected option

❸ Selects the current option

❹ Informs users about the update

Discussion

As mentioned in Recipe 10.1, there are different ways of presenting options to the user. Which type of element you use depends on how you submit the form. With checkboxes and radio buttons, users usually expect to make a selection and confirm it when they press the submit button. With buttons, they expect their choice to have an immediate effect.

Whether you pick client-side or server-side rendering, you must ensure that users can always tell which option is selected. For `<select>` elements, you can use the `selected` attribute, which highlights the selected option visually and programmatically, as shown in Example 10-9. For radio buttons and checkboxes, use the `checked` attribute. For toggle buttons,

use `aria-pressed` and set it to `true` or `false` (see Example 10-11).

If you choose to apply changes immediately, you have to inform users. In Example 10-10, you can see how the items in the list reorder in response to a change in the `<select>` element. You can use a live region to inform screen reader users about the changes. The same is true for changing the presentation of the list, as shown in Example 10-12. Focus management is not a great alternative here because you want to let users finish filling out the form, not move them away from it because they changed the sorting or display.

10.5 Group Filters

Problem

Complex forms can be complicated for everyone, especially those with cognitive disabilities. A filter form quickly becomes overwhelming when you present users with too many options.

Solution

Prioritize popular options and collapse or hide additional settings, as shown in Examples 10-13, 10-14, and 10-15.

Example 10-13. A button wrapped in a legend

```
<fieldset>
  <legend>
    <button type="button" class="toggle" aria-ex
  </legend>

  <div hidden>
    <input type="checkbox" name="country" id="co
    <label for="country_at">Austria</label>

    <input type="checkbox" name="country" id="co
    <label for="country_ca">Canada</label>

    <input type="checkbox" name="country" id="co
    <label for="country_us">USA</label>

    <input type="checkbox" name="country" id="co
    <label for="country_de">Germany</label>

    <input type="checkbox" name="country" id="co
    <label for="country_uk">United Kingdom</labe


  </div>
</fieldset>
```

Example 10-14. Toggling the `aria-expanded` attribute of the button

```
document.querySelector(".toggle").addEventListen
  const button = e.target.closest("[aria-expande
  const isOpen = button.getAttribute("aria-expan

  if (button) {
    button.setAttribute("aria-expanded", isOpen)
  }
});
```

Example 10-15. Showing and hiding the group of options based on the value of the `aria-expanded` attribute

```
fieldset:has([aria-expanded="true"]) > div {
  display:block;
}
```

Discussion

When dealing with data that has many attributes users can filter, consider prioritizing the options. Instead of showing all possibilities all the time, you could show only popular filters

and collapse those that users are less likely to use. In Example 10-13, you can turn a simple fieldset into a disclosure widget (see Chapter 8) by wrapping the text content of the legend in a button and wrapping the form items in a `div`.

NOTE

The `legend` element must be a direct child of the `fieldset` element. Otherwise, the semantic connection between legend and fieldset may not work.

On click, you toggle the `aria-expanded` attribute, indicating whether the group is collapsed (see Example 10-14). The attribute also serves as a hook for styling the group based on its value, as shown in Example 10-15.

There are also other ways of reducing the interface. Sales site *gumtree.com* doesn't hide all options; it shows the first few and hides the rest in a disclosure widget, as shown in Figure 10-3.

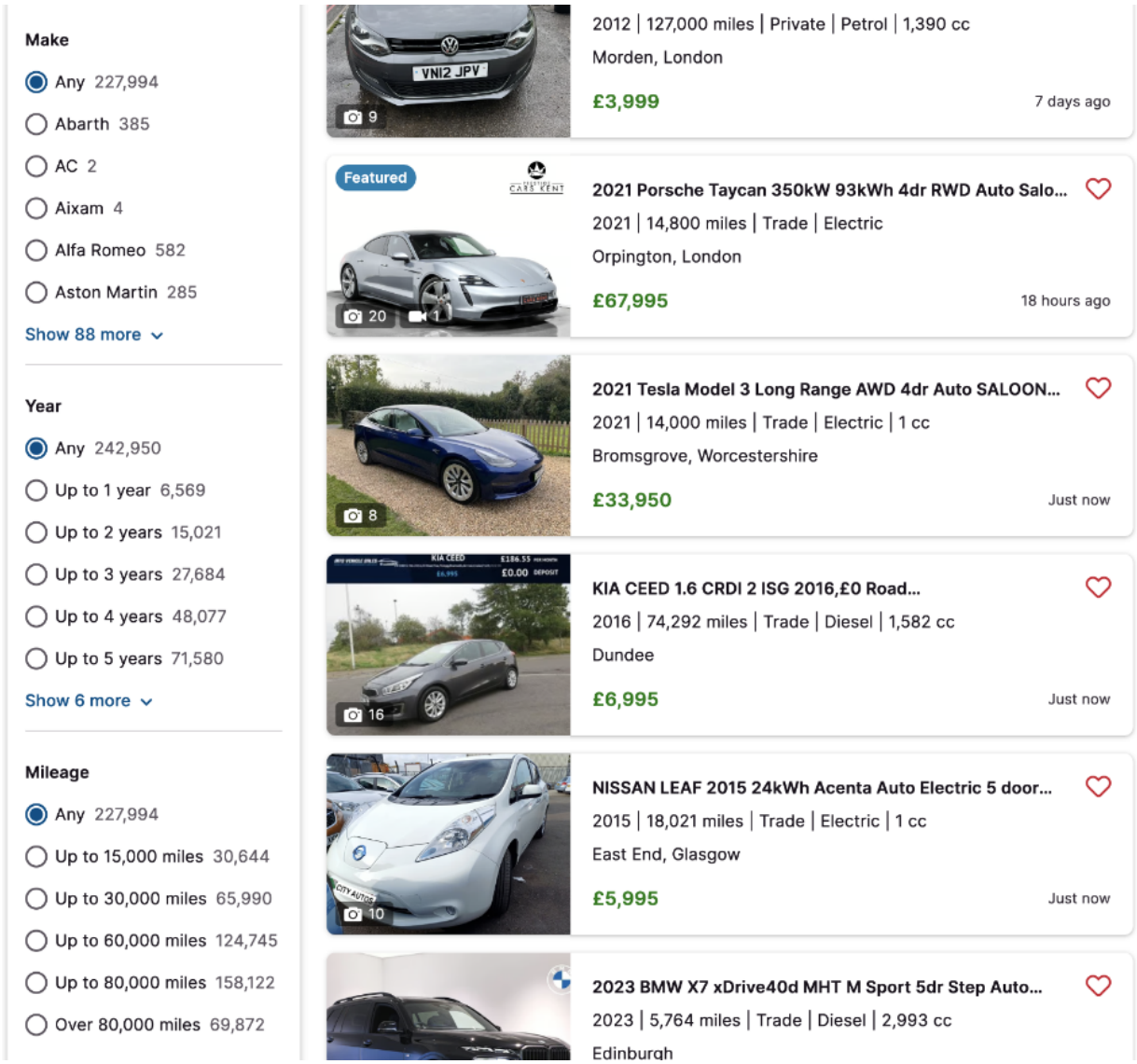


Figure 10-3. gumtree.com showing the first eight options per filter and hiding the rest in a disclosure widget

On *ikea.com*, all filters are collapsed. How many you see depends on the type of products you view and the viewport's width. If there are more options, you can access them by clicking an "All filters" button, which opens a fly-out menu (Figure 10-4).

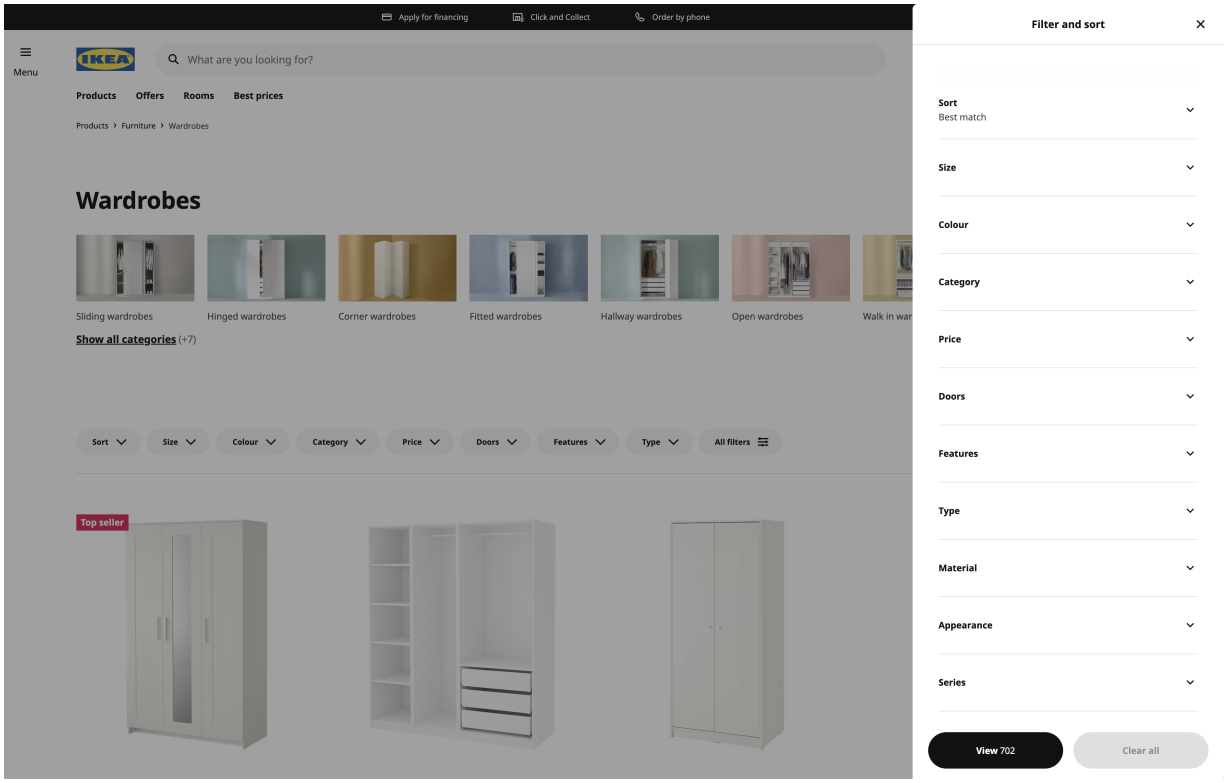


Figure 10-4. ikea.com shows up to eight filter options and an “All filters” button, which toggles a fly-out menu with more options

Hiding filters or options can be a good way of decluttering your interface, but treat it as a last resort. Before you hide something, evaluate whether you need it in the first place.

See Also

- “UX: Infinite Scrolling vs. Pagination” by Nick Babich

Chapter 11. Presenting Tabular Data

Developers have misused tables so much in the past that today, they're almost afraid of using them in fear of harming accessibility. The truth is that tables have their place and can even improve accessibility. Understanding when and how to use them and when another solution might be better is essential.

11.1 Pick the Right Elements

Problem

Like most semantic elements, tables can be helpful to your users, but only if you use them correctly and if they're the right tool for the job. If not, they can make the experience worse. They can provide useful information for screen reader users or make it impossible to consume the content.

Solution

Use tables only when you have data with more than one dimension, and a table-like structure is the best way to present

it and convey its meaning (see Example 11-1 for an example). Don't use tables for layout.

Example 11-1. A table listing names and scores of players in a fictional online game

```
<table>
  <caption>Scores Group A</caption>
  <thead>
    <tr>
      <th>Name</th>
      <th>Score</th>
      <th>Country</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Michael</td>
      <td>27</td>
      <td>Austria</td>
    </tr>
    <tr>
      <td>Robert</td>
      <td>7</td>
      <td>Croatia</td>
    </tr>
    <!-- … -->
```

```
    </tbody>
  </table>
```

Discussion

The `<table>` element is great; please use it! It can improve the user experience significantly. When you decide which element to use to represent data, ask yourself the following questions:

- Is the data multidimensional?
- Would it be helpful to make comparing data easier?
- Can I provide a meaningful label for each column?

If the answer to all three questions is "Yes," use a `<table>`.

You have several options if you want to list related content. You can use `<ul>` and `<ol>` for simple lists, unordered or ordered. You can use `<dl>` for lists of key-value pairs. For example, if you need to list names, you probably want to go for an unordered list, as shown in Example 11-2.

Example 11-2. A list of names

```
<ul>
  <li>Michael</li>
  <li>Robert</li>
```

```
    <li>Andreas</li>
    <!-- … -->
  </ul>
```

You can also list additional data with each name, as shown in Example 11-3.

Example 11-3. A list of names with scores

```
<ul>
  <li>
    Paul<br>
    Score: 34
  </li>
  <li>
    Nicolas<br>
    Score: 8
  </li>
</ul>
```

That's fine, but users may want to compare scores, which would be easier if they were listed closely together. Also, if you add more data, like each player's country of origin, scanning, comparing, and relating data gets harder. When you have data with more than one dimension, it makes sense to present it across two axes, not just vertically but also horizontally. In

these cases, a table might be a better choice, as shown in Example 11-1 and illustrated in Figure 11-1.

Scores Group A

Name	Score	Country
Michael	27	Austria
Robert	7	Croatia
Andreas	53	Austria
Dominik	19	Croatia
David	21	Austria
Heinrich	12	Austria
Markus	14	Austria
Paul	42	Austria
Nicolas	3	Spain

Figure 11-1. A table listing players' names, scores, and countries of origin

Before we had powerful layout tools like CSS Flexbox and Grid, developers often misused tables because the tools made it easy to create multicolumn layouts. Today, you should strictly avoid that. Tables don't play well with responsive web design. They

also make it harder for screen reader users to navigate, if you use them to structure the entire page.

Screen readers provide users with many keyboard shortcuts and helpful information that makes it easier to navigate in a table and understand, compare, and locate data. They vary among different software brands, but usually you get the following features:

- When you enter a table, the screen reader announces the number of rows and columns.
- If a caption is present, it announces it.
- You can navigate between cells using custom screen reader navigation keys.
- If you switch from one column to another, it announces the column header alongside the cell's content.
- If you switch between rows and a row header is present, it announces the row header.
- You can jump from one table to another.

However, if the table has been misused, these features make the experience unpleasant, confusing, or even impossible.

If you're unsure whether to use a table, ask yourself the three questions at the beginning of this section. If that still doesn't

help, you can try this question instead: “If this wasn’t a website, would I pick a text document or a spreadsheet to display the data?”

11.2 Structure Tables

Problem

There are several elements and attributes you can use to create tables. How you combine them depends on your content’s size and complexity. Too little or too much structure can make using and navigating tables harder.

Solution

There are several measures you can take:

Label your tables. Depending on what you’re trying to achieve, you can pick different techniques to label tables. Example 11-4 adds a visible caption to the table.

Example 11-4. Using the `<caption>` element to label a table

```
<table>
  <caption>Scores Group A</caption>
```

```
    <!-- … -->
  </table>
```

In Example 11-5, there's no additional label to avoid verbosity.

Example 11-5. A table inside a `<figure>`, labeled by the `<figcaption>`

```
  <figure>
    <table>
        <!-- … -->
    </table>
    <figcaption>Scores Group A</figcaption>
  </figure>
```

The table in Example 11-6 is labeled by the existing heading so that it gets announced when screen reader users navigate using quick navigation commands.

Example 11-6. A heading that precedes the table

```
  <h2 id="table_heading">Scores Group A</h2>
  <table aria-labelledby="table_heading">
    <!-- … -->
  </table>
```

Use header cells to label your columns or rows, as shown in Examples 11-7 and 11-8.

Example 11-7. Header cells labeling columns

```
<table>
  <caption>Scores Group A</caption>
  <thead>
    <tr>
      <th>Name</th>
      <th>Score</th>
    </tr>
  </thead>
  <tbody>
    <!-- … -->
  </tbody>
</table>
```

Example 11-8. Header cells labeling columns and specific data cells labeling rows

```
<table>
  <caption>Total scores Group A</caption>
  <thead>
    <tr>
      <th>Name</th> ❶
      <th>Q1</th>
```

```
        <th>Q2</th>
        <th>Q3</th>
        <th>Q4</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <th>Michael</th> ❷
        <td>47</td>
        <td>28</td>
        <td>91</td>
        <td>65</td>
      </tr>
      <tr>
        <th scope="row">Robert</th> ❸
        <td>97</td>
        <td>13</td>
        <td>42</td>
        <td>61</td>
      </tr>
      <!-- … -->
    </tbody>
  </table>
```

- ❶ `th` within the `thead` labels the respective column.
- ❷ `th` within the `tbody` labels the respective row implicitly.

❸ `th` with a row scope within the `tbody` labels the respective row explicitly.

Ensure that the table is usable on smaller viewports and doesn't cause any overflow, as shown in Example 11-9.

Example 11-9. A table nested in a scrollable and focusable region

```
<div role="region" aria-labelledby="scores_capti
  <table>
    <caption id="scores_caption">Total scores Gr
    <!-- … -->
  </table>
</div>

<style>
  [role="region"][tabindex="0"][aria-labelledby]
    overflow: auto;
  }
</style>
```

Discussion

The same rule that applies to most elements also applies to tables: avoid complexity and provide only as much information

and structure as necessary.

Labeling

In Recipe 11.1, I explained what makes tables special for screen reader users. Labeling plays a vital role in that.

In some screen readers, you can jump from table to table using the `T` key without interacting with the rest of the page. When you enter a table, the software usually announces the number of columns and rows. Without context, you can't tell what data the table contains. That's why it's essential to label your tables.

The best way to label a table and provide an accessible name is the `<caption>` element, as shown in Example 11-4. It adds a visual label to the table, and screen readers announce it alongside the number of rows and columns. Alternatively, you can use a `<figcaption>` if the table is contained in a `<figure>`, as shown in Example 11-5. If a heading precedes a table, it can also serve as kind of a label, as shown in Example 11-6. The advantage of the caption element is that a screen reader reads it out if the user accesses the table directly.

Users can use shortcuts to navigate between cells. They can also go up and down in a column, or switch from one column to another. Screen readers usually announce the column's name

when users do that, if you've provided one. Use header cells (`<th>`) to label your columns (see Example 11-7). If you have a table where the first (or any other cell) in each row labels the row, use the `<th>` element, as shown in Example 11-8 and Figure 11-2. That provides additional clarity and context, because the screen reader knows that this header cell labels the row and announces it when you switch rows.

Total scores Group A

Name	Q1	Q2	Q3	Q4
Michael	47	28	91	65
Robert	97	13	42	61
Dominik	29	28	19	57
David	61	11	29	59
Markus	72	70	73	44
Paul	122	100	87	69

Figure 11-2. Cells in the name column labeling their row

According to Steve Faulkner, it's not necessary to use the `scope="row"` attribute and value on the `th`, but you can (in case a screen reader doesn't recognize the table header correctly).

Responsiveness

One of the pitfalls of tables is their inflexible styling. Optimizing them for small viewport widths can be challenging. If you decide to change the layout of a table completely, ensure that it still functions in all relevant browsers and screen readers. In Recipe 5.4, I explain how the `display` property on tables and table rows can make them completely unusable.

An alternative to changing the table layout is to make it scrollable. In Example 11-9, you can see how I wrap the table in a scrollable `div`. Scrollable elements aren't keyboard accessible in all browsers, so you need to put a `tabindex="0"` on the `div`, which makes it focusable. That requires it to have an accessible name. That's why you should use `aria-labelledby` with a reference to the caption element inside the table. Alternatively, you can also use `aria-label`. Finally, since it's not valid to label generic elements, add a `role="region"` to the `div`. As a bonus, this also makes it a landmark.

Other

The examples in this recipe use the `thead`, `tbody`, and `tfoot` elements. Those don't affect accessibility, but they can simplify scanning the code and understanding its structure and styling. Just ensure that the thead containing the table headers is the first element in the table (or the second, if there's a caption).

You may have noticed that none of the examples in this recipe uses the `rowspan` and `colspan` attributes. Adrian Roselli advises avoiding spanning table headers, because screen readers do not support them well.

11.3 Add Interactive Elements

Problem

Tables sometimes contain interactive elements. Since users can access them not only by interacting with cells and rows directly but also by tabbing, the same rules apply to them as to any other interactive element: they must be focusable, labeled, and clearly identifiable. If that's not the case, keyboard users may be unable to access them and screen reader users unable to identify them.

Solution

Add buttons with unique labels, as shown in Example 11-10.

Example 11-10. Each row containing a details button

```
<table>
  <caption>Scores Group A</caption>
  <thead>
    <tr>
      <th>Name</th>
      <th>Score</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th id="name1">Michael</th>
      <td>27</td>
      <td>
        <button id="details1" aria-labelledby="d
      </td>
    </tr>
    <!-- … -->
  </tbody>
</table>
```

- A reference to itself and the column header

Label input fields even if they don't have visible labels, as shown in Example 11-11.

Example 11-11. Each row containing an input field

```
<table>
  <caption>Scores Group A</caption>
  <thead>
    <tr>
      <th>Name</th>
      <th id="score">Score</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td id="name1">Michael</td>
      <td><input type="text" aria-labelledby="na
    </tr>
    <!-- … -->
  </tbody>
</table>
```

❶ A reference to itself and the column header

Use a link or a button and CSS to create clickable rows, as shown in Example 11-12.

Example 11-12. Clickable rows

```
<table>
  <caption>Total scores Group A</caption>
  <thead>
    <tr>
      <th>Name</th>
      <th>Q1</th>
      <th>Q2</th>
      <th>Q3</th>
      <th>Q4</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row"><a href="#">Michael</a></t
      <td>47</td>
      <td>28</td>
      <td>91</td>
      <td>65</td>
    </tr>
    <!-- … -->
  </tbody>
</table>

<style>
  table {
    overflow: hidden; ❶
```

```
  }

  tr:has(a):is(:hover, :focus-within) {  ❷
    background: #eee;
  }

  td {
    position: relative; ❸
  }

  table a::before {
    content: "";
    cursor: pointer;
    position: absolute; ❹
    inset: 0;
    width: 100vw; ❺
    z-index: 1;
  }
</style>
```

❶ Cut off overflowing clickable areas.

❷ Highlight tables containing a link visually on hover or when the link is focused.

❸ Create a new stacking context for every `<td>`; necessary for Safari because it doesn't support `position:`

`relative` on `<tr>`.

4. Put the `pseudo` element on top of the containing cell.
5. Make the `pseudo` element span the entire row (and more). The `overflow:hidden` declaration on the `<table>` makes sure there's no horizontal scrolling.

Discussion

Tables sometimes contain interactive elements like buttons that allow users to show details, delete rows, or perform other actions. Using these doesn't necessarily imply that the complexity of the table itself has to increase. Sara Higley explains that you don't necessarily have to turn your table into a complex grid. You can stick to the native table element, but you must ensure these elements are also accessible and identifiable *outside* the table context.

Let's say you have a table with 50 rows, with an option to show more details for each row. There are thus 50 buttons labeled "Details." If you use table navigation keys with a screen reader to navigate inside the table, it's easy to understand to which row each button refers from context. But, if you use the `Tab` key instead and jump from button to button, you'll hear

“Details” 50 times without any reference to the corresponding row. In Example 11-10, you see one way of addressing that issue. Each button is labeled by itself and a cell that identifies the row clearly—in this case, the player’s name. The element must reference itself to maintain its original label because `aria-labelledby` overrides the accessible name from the button’s content. Instead of just “Details,” the screen reader announces, “Details Michael.”

You can use the same technique if your cells contain input fields without labels. The input elements in Example 11-11 refer to an identifier in the same row and the column’s label. The accessible name of each input is “[name] score.”

Another common issue comes from trying to link entire rows. Putting a click event on each table row works for pointer and touch users. However, keyboard and screen reader users won’t be able to identify and activate the row because it’s not an interactive element. You can use the block-link technique introduced in Recipe 3.8, as shown in Example 11-12. This solution is almost perfect, but due to a bug in Safari, it’s impossible to solve that issue entirely without a little help. Safari doesn’t support `position: relative;` on table rows. That’s why you must position the pseudoelement relative to the containing cell and make it span the entire screen. To avoid

horizontal scrolling, the `<table>` needs `overflow: hidden;`.

11.4 Sort Columns

Problem

When you allow users to sort columns in tables, you have to ensure several things to enable the feature for everyone:

- The sort button is an actual `<button>` element.
- Columns communicate how they're sorted in machine- and human-readable ways.
- Sorting provides semantic *and* visual feedback.

If you don't, you're making this valuable feature harder or even impossible for keyboard and screen reader users.

Solution

Add buttons for sorting to your column headers, as shown in Example 11-13.

Example 11-13. A table with buttons for sorting

```
  <div role="status" class="visually-hidden"></div>
```

```
<table>
  <caption>Scores Group A</caption>
  <thead>
    <tr>
      <th>
        <button class="sort"> ❶
          Name
          <svg width="13" viewBox="0 0 126 171" a
            <path d="M62.7 3.9 6 70l114-.5z"/>
            <path d="M63 166.5 6 100.6h114z"/>
          </svg> ❷
        </button>
      </th>
      <th>
        <button class="sort">
          Score
          <svg width="13" viewBox="0 0 126 171" a
            <path d="M62.7 3.9 6 70l114-.5z"/>
            <path d="M63 166.5 6 100.6h114z"/>
          </svg>
        </button>
      </th>
    </tr>
  </thead>
  <tbody>
    <tr>

      <td>Michael</td>
      <td>27</td>
```

```
    </tr>
    <!-- … -->
  </tbody>
</table>

<style>
.sort {
  all: unset; ❸
  display: flex;
  gap: 0.4rem;
  align-items: center;
}

.sort path {
  fill: transparent;
  stroke: currentColor;
  stroke-width: 12;
}

[aria-sort="ascending"] path:first-child { ❹
  fill: currentColor;
}

[aria-sort="descending"] path:last-child { ❺
  fill: currentColor;


}
</style>
```

```
```

❶ Button for sorting

❷ An SVG file containing triangles pointing up and down

❸ Removes default button styles

❹ Fills the up-pointing triangle

❺ Fills the down-pointing triangle

Inform users how tables are sorted by using a live region and ARIA attributes, as shown in Example 11-14.

Example 11-14. Logic for sorting tables

```
const table = document.querySelector("table");
const liveRegion = document.querySelector("[role=
let toSort;
let direction = "ascending";

table.addEventListener("click", (e) => {
  const button = e.target.closest("thead button"

  if (button) {

    const cell = button.parentNode;
    const tbody = table.querySelector("tbody");
```

```
    const rows = tbody.querySelectorAll("tr");

    toSort = [];
    getRows(cell, rows); ❶
    updateButton(cell); ❷
    sortRows(rows); ❸
    updateLiveRegion(); ❹
  }
});

const getRows = (cell, rows) => {
  const index = [...cell.parentNode.children].in

  for (let i = 0; i < rows.length; i++) {
    const row = rows[i];
    const cells = row.querySelectorAll("td");

    toSort.push([cells[index].innerText, row.clo
  }
};

const sortRows = (rows) => {
  toSort.sort(function (a, b) {
    const comp = a[0].localeCompare(b[0], "en",
    return comp;
  });


  if (direction === "descending") {
```

```
    toSort.reverse();
  }

  for (let i = 0; i < rows.length; i++) {
    const row = rows[i];
    row.parentNode.replaceChild(toSort[i][1], ro
  }
};

const updateButton = (cell) => {
  const sortedColumn = table.querySelector("[ari
  if (sortedColumn && sortedColumn !== cell) {
    sortedColumn.removeAttribute("aria-sort");
  }

  direction =
    cell.getAttribute("aria-sort") === "ascendin
  cell.setAttribute("aria-sort", direction);
};

const updateLiveRegion = () => {
  liveRegion.textContent = `Sorted ${direction}`

  setTimeout(() => {
    liveRegion.textContent = ``;


  }, 1000);
};
```

```
```

1. Gets all values of the current column and saves them in an array
2. Puts the `aria-sort` attribute on the table header of the sorted column and removes the attribute if it's present in another column
3. Sorts and reorders the rows
4. Updates the live region, then clears it after one second

Discussion

This recipe contains a simple solution for sorting and reordering rows in a table. You can use it as is or replace it with more sophisticated logic—that doesn't matter. What's important is what happens before and after sorting.

To inform users that a column is sortable, you need a button and a visual indicator like an icon. Don't put the click event on the table header directly because that would make sorting inaccessible to keyboard users. You need an interactive element like a button. The button doesn't just make it accessible to everyone—it also indicates to blind users that the column can

be sorted. A simple label like “Name” or “Score” that describes the column is sufficient; you don’t need additional text to indicate that the columns are sortable. For sighted users, you can use an icon. A well-recognized symbol for sorting is arrows or triangles that point up and/or down, as shown in Figure 11-3.

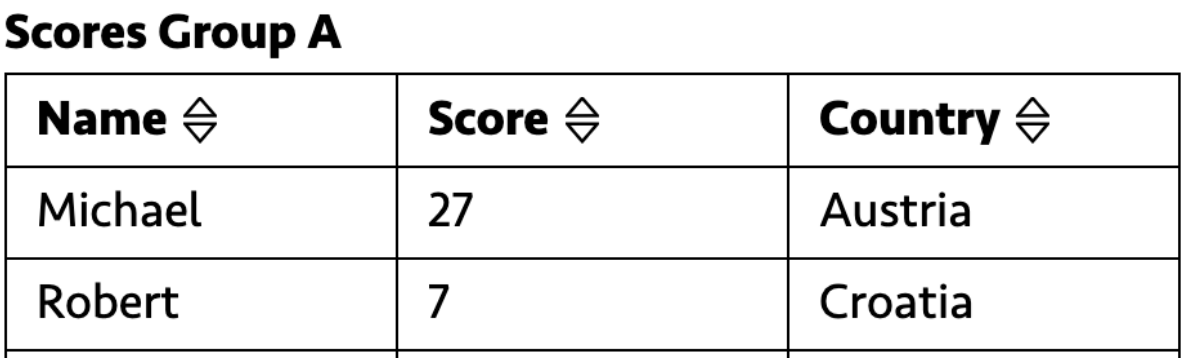


Figure 11-3. Triangles indicate that these columns can be sorted

When the user clicks the button, you must inform them in a human- and machine-readable way that the rows have been sorted. You can change the icon’s styling, to indicate ascending or descending order, as shown in Figure 11-4. For screen readers, put the `aria-sort` attribute on the `<th>` and set the value either to *ascending* or *descending*. To avoid confusion, remove the attribute when the user clicks another column. Avoid verbosity by not changing the button’s label to something like “Score (sorted descending),” because the `aria-sort` attribute already provides that information.

Scores Group A

Name ⇳	Score ⇳	Country ⇳
Andreas	53	Austria
Paul	42	Austria

Figure 11-4. A filled triangle indicates a column sorted in descending order

Setting the attribute doesn't trigger any announcement in most screen readers. It informs users about the sorted column only when they encounter it. Using a live region is one way of telling screen reader users that sorting was successful. With each click, you change the value of the live region to *sorted ascending* or *sorted descending*. Clear the content of the live region again after about a second because there's no announcement if the user sorts another column in the same direction, and the region's content doesn't change.

See Also

- "Uniquely Labeling Fields in a Table" by Adrian Roselli
- "Sortable Table Columns" by Adrian Roselli
- "Inclusive Components—Data Tables" by Heydon Pickering

Chapter 12. Creating Custom Elements

Web components are a set of web platform APIs that allow you to build your own fully featured DOM elements.

Being able to create custom elements to build interactive websites natively is exciting, but it also introduces new accessibility issues. You must be aware of the limitations and opportunities of custom elements and their related APIs. With the right architecture and enough planning, web components can encourage an accessibility-first development mindset and create great experiences. They can also break accessibility inherently, if used without caution.

12.1 Working with IDs

Problem

It's impossible to reference an element from Light DOM in Shadow DOM, or vice versa, using the `id` attribute. If you're not aware of this limitation and try to create these references anyway, the broken relation can affect users:

- Skip links may not work, making navigation harder.
- Form elements may not have proper labels, making it harder for screen reader users to distinguish them.
- ARIA references may break, resulting in missing information or feedback for screen reader users.

Solution

In a form, put both the label and the form field in Light DOM or both in Shadow DOM, but don't mix the two contexts, as shown in Examples 12-1 and 12-2.

Example 12-1. The label and input field are in Light DOM

```
<label for="email">E-Mail</label>

<the-input>
  <input type="email" id="email" />
</the-input>

<script>
  class TheInput extends HTMLElement {
    constructor() {
      super();

      this.attachShadow({ mode: "open" });
      this.shadowRoot.innerHTML = `<slot></slot>
```

```
    }
  }

  customElements.define("the-input", TheInput);
</script>
```

Example 12-2. The label and input field are in Shadow DOM

```
<the-input></the-input>

<script>
  class TheInput extends HTMLElement {
    constructor() {
      super();

      this.attachShadow({ mode: "open" });
      this.shadowRoot.innerHTML = `
        <label for="email">E-Mail</label>
        <input type="email" id="email" />
      `;
    }
  }

  customElements.define("the-input", TheInput);
</script>
```

Discussion

The DOM is a tree-like representation of the HTML on your web pages. It's built with all the elements on the page, and its branches reflect the hierarchy and relationships between them. It lets you interact with the elements in your HTML document using JavaScript.

In the context of web components, this DOM tree is called the Light DOM. Its counterpart, the Shadow DOM, defines additional smaller DOM trees that you can attach to custom elements. Elements in those smaller trees are *encapsulated*, which means that from the outside (the document), you don't have direct access to elements on the inside (called the *shadow root* of the web component). That can be useful, but also limiting.

One of the limitations is that IDs are also scoped within a shadow root. It's important to know that, because your web components usually live inside a document, where they interact with content from the Light DOM. In Example 12-3, you can see a simple illustration of the issue. In the document (Light DOM) is an anchor link that points to an element inside a custom element (Shadow DOM). When you click the link, the hash in the address bar of your browser changes, but the link doesn't

take you anywhere. It's looking for the ID *content*, but the element with that ID doesn't exist in the document, only in the shadow root of the component—and that isn't accessible from the outside.

Example 12-3. An anchor link in Light DOM trying to point to an ID in Shadow DOM

```
<a href="#content">Skip to content</a>
<the-component></the-component>


<script>
class TheComponent extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });

    this.shadowRoot.innerHTML = `
      <div id="content"></div>
    `;
  }
}

customElements.define("the-component", TheCompon
</script>
```

This limitation is especially problematic when you need ID references to add semantic information to elements. The input in Example 12-4 is not properly labeled, because the ID the label is trying to reference exists only in the shadow root of the component.

Example 12-4. A broken label to form control reference

```
<label for="email">E-mail</label>
<the-input></the-input>

<script>
class TheInput extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: "open" });
    this.shadowRoot.innerHTML = `<input type="em
  }
}

customElements.define("the-input", TheInput);
</script>
```

Implicit labeling (see Example 12-5) doesn't solve the issue either. The label element works only on labelable element

descendants. These include `button`, `input` (if the type attribute is not in the hidden state), `meter`, `output`, `progress`, `select`, `textarea`, and form-associated custom elements.

Example 12-5. A broken implicit label to form relation

```
<label>
  E-mail
  <the-input></the-input>
</label>
```

The `ElementInternals` API solves the labeling issue partially because it allows custom elements to participate in form submissions and validations. The `formAssociated = true` property associates the element with a form, and `this.internals` gives you access to internal information like associated labels, as shown in Example 12-6. You could use the API to provide an accessible name for the input field, but it doesn't properly connect the input with the label since clicking the label doesn't put focus on the input.

Example 12-6. Getting access to associated labels using the `ElementInternals` API

```
class TheInput extends HTMLElement {
  static formAssociated = true;

  constructor() {
    super();

    this.internals = this.attachInternals();

    this.attachShadow({ mode: "open" });
    this.shadowRoot.innerHTML = `<input type="em
  }

  connectedCallback() {
    console.log(this.internals.labels[0].textCon
  }
}

customElements.define("the-input", TheInput);
```

1. Returns the text content of the first associated label element

Encapsulation also affects ID references with ARIA attributes, which I discuss in detail in Recipe 12.2.

In Examples 12-1 and 12-2, you can see two solutions that work because they establish the relation between two elements only, either in the context of the Light DOM or in the context of the Shadow DOM. That may change in the future—but for now, that's the only practical advice I can offer you.

12.2 Creating ARIA References

Problem

When working with web components, DOM encapsulation can become an accessibility issue. As discussed in Recipe 12.1, one of the reasons is that element IDs are scoped within a shadow root. That affects ARIA references, which means that users may miss important information about state, accessible names, and descriptions if you don't create those references correctly.

Solution

Create ARIA references in Light DOM or in Shadow DOM only, but don't mix contexts, as shown in Examples 12-7 and 12-8.

Example 12-7. The input and the referenced paragraph are in Light DOM

```
<label for="date">Birthday</label>
<input type="date" id="date" aria-describedby="h
<the-hint>
  <p id="hint">
    Format: DD.MM.YYYY
  </p>
</the-hint>
```

Example 12-8. The input and the paragraph are both in Shadow DOM

```
class TheInput extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: "open" });
    this.shadowRoot.innerHTML = `
      <label for="date">Birthday</label>
      <input type="date" id="date" aria-describe
      <the-hint>
        <p id="hint">
          Format: DD.MM.YYYY
        </p>
      </the-hint>

    `;
  }
}
```

```
customElements.define("the-input", TheInput);
```

Discussion

The issues with ID references described in Recipe 12.1 also apply to ARIA references. Attributes like `aria-labelledby`, `aria-describedby`, `aria-controls`, `aria-owns`, and `aria-activedescendant` use IDs to reference one or multiple other elements. A reference breaks if the referenced element doesn't exist in the same context (Light DOM or Shadow DOM), as shown in Example 12-9.

Example 12-9. A broken `aria-describedby` reference

```
<label for="date">Birthday</label>
<input type="date" id="date" aria-describedby="h
<the-hint>
  #shadowRoot
  | <p id="hint">
  | Format: DD.MM.YYYY
  | </p>
  #shadowRoot
</the-hint>
```

IDREF attribute reflection and ARIA Mixins can solve this issue, but there are constraints and browser support is currently insufficient.

ARIA Mixins

Every ARIA content attribute that refers to other elements by their IDs, such as `aria-labelledby`, `aria-describedby`, and `aria-controls`, has a corresponding property called IDL attribute on DOM elements, which you can set or get via JavaScript. You can see a list of example properties and their corresponding reflected content attributes in Table 12-1.

Table 12-1. Examples of reflected ARIA content attributes

IDL	Reflected ARIA
ariaActiveDescendantElement	aria-activedescendant
ariaControlsElements	aria-controls
ariaDescribedByElements	aria-describedby
ariaLabelledByElements	aria-labelledby
ariaOwnsElements	aria-owns

You can find the full list in the ARIA specification.

Instead of setting the content attribute by referencing an ID (`input.setAttribute('aria-describedby', 'hint')`), you can set the IDL attribute by referencing an array of elements (`input.ariaDescribedByElements = [hint.shadowRoot.querySelector('#hint')]`).

The obvious advantage is that you no longer have to connect elements via their IDs, as shown in Example 12-10.

Example 12-10. Using IDL attribute reflection to set ARIA properties

```
<label for="date">Birthday</label>
<input type="date" id="date">
<the-hint>
  #shadowRoot
  | <p id="hint">
  | Format: DD.MM.YYYY
  | </p>
  #shadowRoot
</the-hint>

<script>
  const input = document.querySelector('#date')

  const hint = document.querySelector('the-hint'
  input.ariaDescribedByElements = [hint.shadowRo
</script>
```

ARIA Mixins work as a solution only if the referenced element is in the same shadow root as the target element, or if the referenced element is in a parent, grandparent, or ancestor shadow root of the target element. Currently, only Safari and Chrome Canary support ARIA Mixins.

ARIA Mixins can solve some problems when support improves. However, they're not a universal solution due to the previously mentioned constraints. Unfortunately, they're the only real option we have now.

Two proposals could solve this issue for good: cross-root ARIA delegation and cross-root ARIA reflection.

Cross-root ARIA delegation

The idea behind cross-root ARIA delegation is that you can set a new option to `attachShadow()` called `delegatesAriaAttributes` (similar to `delegatesFocus`). This option enables ARIA attributes set on a custom element to be forwarded to elements inside of that element's shadow root. Example 12-11 demonstrates how cross-root ARIA delegation could work.

Example 12-11. `delegatesAriaAttributes` delegates `aria-describedby` from the host to elements inside its shadow tree

```
<p id="hint">
  Format: DD.MM.YYYY
</p>

<the-input aria-describedby="hint"></the-input>

<script>
class TheInput extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({
      mode: "open",
      delegatesAriaAttributes: "aria-describedby
    });

    this.shadowRoot.innerHTML = `
      <label for="date">Birthday</label>
      <input type="date" id="date" delegatedaria
    `
  }
}
```

```
  customElements.define("the-input", TheInput);
  </script>
```

Cross-root ARIA reflection

The idea behind cross-root ARIA reflection is that you can set new options to `attachShadow()` for ARIA attributes (`reflects*`), similar to `delegatesFocus`. This lets you make elements inside a shadow root available as targets for relationship attributes. Example 12-12 demonstrates how cross-root ARIA reflection could work.

Example 12-12. `reflectariadescribedby` reflects `aria-describedby` from elements inside a shadow tree to its host

```
  <label for="date">Birthday</label>
  <input type="date" id="date" aria-describedby="h
  <the-hint id="hint">
    #shadowRoot
    | <p reflectariadescribedby>
    | Format: DD.MM.YYYY
    | </p>


    #shadowRoot
  </the-hint>
```

Cross-root ARIA delegation and reflection could solve the issues described in this recipe, but they're still only proposals and have not been implemented in any browser yet.

If all your relationships for an element happen exclusively in Light DOM or Shadow DOM and you don't try to cross boundaries, working with ARIA won't be a problem. That's not always possible, though. Without a doubt, this problem needs a solution. Alice Boxhall describes it well:

> *The contents of the shadow root are private to its light tree, but not to users. If a user can perceive a relationship between elements in the light tree and the shadow tree, but the author can't express that relationship in code, then the encapsulation provided by Shadow DOM is at odds with the semantics of the page, and so at odds with accessibility. This is a conundrum for Shadow DOM.*

Many experts, like Simon MacDonald, recommend carefully considering your use case before reaching for the Shadow DOM by default because eschewing Shadow DOM prevents the issues described in this recipe and in Recipe 12.1.

12.3 Focus Elements in Shadow DOM

Problem

Nodes within the Shadow DOM of a component are not directly accessible from the document. Not being able to query them also means that developers can't focus them programmatically, which if you manage focus can break keyboard accessibility.

Solution

You have two options: Access the shadow root (Example 12-13) or delegate focus (Example 12-14).

Example 12-13. Accessing a node within the `shadowRoot` of a web component with an open Shadow DOM

```
class TheButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });

    const button = document.createElement("butto
    button.textContent = "Click me";
    button.addEventListener("click", () => alert


    this.shadowRoot.append(button);
  }
}
```

```
customElements.define("the-button", TheButton);

const theButton = document.querySelector("the-bu
const button = theButton.shadowRoot.querySelecto
button.focus(); ❸
```

❶ Query the component.

❷ Access the shadow root of the component and query the element you want to focus.

❸ Focus the element within the shadow root.

Example 12-14. Enabling focus delegation from the host to its first child

```
class TheButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({
      mode: "open",
      delegatesFocus: true ❶
    });


    const button = document.createElement("butto
    button.textContent = "Click me";
```

```
      button.addEventListener("click", (e) => aler

      this.shadowRoot.append(button);
    }
  }

  customElements.define("the-button", TheButton);

  const theButton = document.querySelector("the-bu
  theButton.focus(); ❸
```

❶ Enable focus delegation.

❷ Query the component.

❸ Focus the component directly.

Discussion

It's not impossible to focus an element in Shadow DOM, but if and how you can do it depends on the mode of the shadow root you're attaching and which element you want to focus.

Accessing the shadow root

You can create an *open* or *closed* Shadow DOM when you attach a shadow tree to a node.

Open means that JavaScript from the outside has access to the nodes inside the Shadow DOM. That's usually the default. In Example 12-15, you can see how the `shadowRoot` property gives you access to the Shadow DOM tree of the component and returns 1 for the length of the button node list.

Example 12-15. A component with an open `shadowRoot`

```
class TheButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({
      mode: "open" ❶
    });

    const button = document.createElement("butto
    button.textContent = "Hello World";
    this.shadowRoot.append(button);
  }
}

customElements.define("the-button", TheButton);


const theButton = document.querySelector('the-bu
```

```
console.log(theButton.shadowRoot.querySelectorAl
// returns 1
```

❶ Open shadow root.

In a *closed* Shadow DOM, the same query throws an error, because a closed Shadow DOM denies access to the nodes from the outside, as illustrated in Example 12-16.

Example 12-16. A component with a closed `shadowRoot`

```
class TheButton extends HTMLElement {
  constructor() {
    super();
    this._shadow = this.attachShadow({
      mode: "closed" ❶
    });

    const button = document.createElement("butto
    button.textContent = "Hello World";
    this._shadow.append(button);
  }
}

customElements.define("the-button", TheButton);


const theButton = document.querySelector('the-bu
```

```
console.log(theButton.shadowRoot.querySelectorAl
// returns “Cannot read properties of null (read
```

1. Closed shadow root.

The solution in Example 12-13, which uses the `shadowRoot` property to access elements inside the Shadow DOM of a component, works only with an *open* shadow root. If that's the case, then finding shadow nodes is pretty straightforward. Instead of querying elements directly on the host component (`theButton.querySelectorAll('button')`), you first access the shadow root, then you query `theButton.shadowRoot.querySelectorAll('button')`.

Delegating focus

Another solution that works both with an open and closed Shadow DOM is focus delegation. When you attach the Shadow DOM, you can pass the delegatesFocus option in addition to the mode. When it's true, you can call `focus()` on the host itself without accessing the shadow root, and the first focusable element in the host's Shadow DOM receives focus. You can pick this option only if you strictly want to focus the first focusable element, as shown in Example 12-14.

If your components have an open shadow root, focus management is not an issue. If the shadow root is closed, your only option is to focus the component itself and delegate focus to the first focusable element.

12.4 Debugging and Testing

Problem

When testing and debugging web components, know the limitations and peculiarities of your testing tools and techniques. Some tools don't support Shadow DOM, which means that you might miss accessibility bugs when running automated tests with the wrong tool.

Solution

Before you pick and rely on an automated testing tool, ensure that it supports web components and Shadow DOM by running it on an encapsulated test component that includes intentional bugs.

Manual testing should be the same as you're used to, but debugging keyboard accessibility is a bit different when dealing

with Shadow DOM. Use `document.activeElement.shadowRoot.activeElement` to return the actively focused element inside the shadow root of a component, as shown in Example 12-17.

Example 12-17. Accessing the currently focused element inside a shadow tree

```
console.log(document.activeElement)
// returns the component itself

console.log(document.activeElement.shadowRoot.ac
// returns the element focused inside the compon
```

Discussion

Many automated testing tools are a collection of JavaScript functions you run on a page. Most of those rely on querying the DOM. If a tool doesn't consider shadow trees, it catches only accessibility errors in the Light DOM, which may give you a wrong sense of safety and could affect users. That's one more reason not to rely on automated testing only (more on that in Recipe 13.1).

That doesn't mean you shouldn't use automated testing if your site contains web components. You just have to ensure that the tool you're using supports those components. A good way of doing that is to create a simple component that contains some accessibility bugs caused by nodes attached to the component's Shadow DOM, as illustrated in Example 12-18.

Example 12-18. A component with accessibility bugs for evaluating automated testing tools

```
<main>
  <h1 id="h1">Testing a11y bugs in web component

  <a href="#el">Jump to elem in shadow</a>

  <some-bugs></some-bugs>
   <h4>Heading</h4>
</main>

<script>
class SomeBugs extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({mode: 'open'});


    this.shadowRoot.innerHTML = `
    <style>:host { color: #efefef }</style> ❶
```

```
      <h4 id="el">Heading</h4> ❷
      <img src="#test"> ❸

      <button aria-labelledby="h1"></button> ❹
      <input type="text" id="input"> ❺
        `
    }
  }

  customElements.define('some-bugs', SomeBugs);
  </script>
```

❶ Low contrast

❷ Skipped heading

❸ Missing alt

❹ Broken ARIA reference/missing accessible name

❺ Missing label

If you add this component to a page, your testing tools should find at least five issues. I picked five popular tools: axe DevTools, Google Lighthouse, ARC Toolkit, WAVE, and IBM Equal Access Accessibility Checker (IBM EAAC), and ran them

against the component in Example 12-18. I've summarized the results in Table 12-2.

Two tools, axe DevTools and Lighthouse, both based on axe-core, found all five issues. WAVE found no problems in Shadow DOM, but it reported a broken skip link in Light DOM. IBM EAAC found all issues except the skipped heading, which can be considered a bad practice and not a violation. IBM EAAC doesn't report that anyway, regardless of the type of DOM. ARC Toolkit found all issues except the low contrast, and it reported the broken skip link.

Table 12-2. Comparison of issues reported in different testing tools

Bug	axe/Lighthouse	WAVE	ARC	I
Detected issues	5	1	5	4
Broken label/input	yes	no	yes	y
Broken ARIA	yes	no	yes	y
Missing alt	yes	no	yes	y
Skipped heading	yes	no	yes	n
Low contrast	yes	no	no	y
Broken anchor link	no	yes	yes	n

I’m not aware if Shadow DOM is on WAVE’s roadmap, but in the meantime, I’d recommend not using it with sites that contain web components, or using another tool to double-check.

The same applies to HTML validators. The official W3C validator, for example, doesn't support Shadow DOM.

There are no manual testing constraints on using screen readers, the keyboard, or other assistive technology. The only particularity with testing focus on elements nested in shadow roots is that `document.activeElement` doesn't return the currently focused element, as it does in Light DOM. It always returns the host component. To get the focused element instead, you must access it through its parent's shadow root, as shown in Example 12-17.

12.5 Enforce Best Practices

Problem

If you create customizable components as part of a design system that others will use, you can ensure a certain baseline level of accessibility. However, if others implementing your components don't know how to use them in an accessible manner, some of your efforts may become worthless.

Solution

Use web components to enforce best practices, compensate for issues that developers may cause, and report errors. Here are some examples:

Progressive enhancement

Design your web components in a way that facilitates progressive enhancement, as shown in Example 12-19.

Example 12-19. A progressively enhanced disclosure widget

```
<the-disclosure>
  <p>Here's more content…</p>
</the-disclosure>

<template id="disclosure">
  <button aria-expanded="false">Details</button>
  <div class="content" hidden>
    <slot></slot>
  </div>
</template>

<script>
class TheDisclosure extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: "open" });
```

```
    let template = document.getElementById("disc
    let templateContent = template.content;
    this.shadowRoot.appendChild(templateContent.

    this._attachStyles(); ❷

    const button = this.shadowRoot.querySelector
    this._expanded = button.getAttribute("aria-e
    button.addEventListener("click", this._toggl
  }

  _attachStyles() {
    const style = document.createElement("style"
    style.textContent = `
      [aria-expanded="true"] + .content {
        display: block;
      }
    `;

    this.shadowRoot.appendChild(style);
  }

  _toggle() {
    this._expanded = !this._expanded;
    this.setAttribute("aria-expanded", this._exp

  }
}
```

```
customElements.define("the-disclosure", TheDiscl
</script>
```

1. Clone the template for the component and attach it to the Shadow DOM.
2. Add basic styling for the disclosure widget.
3. Retrieve the button's expanded state (`true` or `false`).

Compensate for issues

Use named slots to enforce a specific DOM order, as shown in Examples 12-20 and 12-21.

Example 12-20. A card component with three named slots

```
class TheCard extends HTMLElement {
  constructor() {
    super();
    this.shadow = this.attachShadow({ mode: "ope


    this.shadow.innerHTML=`
      <slot name="heading"></slot>
      <slot name="media"></slot>
```

```
        <slot name="content"></slot>
      `;
    }
  }

  customElements.define("the-card", TheCard);
```

Example 12-21. An image, heading, and content passed to the card component in that exact order

```
<the-card>
  <img slot="media" src="image.jpg" width="436" h
  <h2 slot="heading">Heading</h2>
  <div slot="content">
    <p>Content….</p>
  </div>
</the-card>
```

Error reporting

If mandatory attributes are missing, you can display an error instead of rendering the component (see Example 12-22).

Example 12-22. This map component shows an error instead of a map if you don't provide a label for it

```
class TheMap extends HTMLElement {
  static observedAttributes = ["title"];

  constructor() {
    super();

    this.attachShadow({ mode: "open" });

    let template = `
      <iframe title="${this.title}"
              width="300" height="200"
              src="https://www.openstreetmap.org/
    `

    if (!this.title) {
      template = 'Please provide a title for this
    }

    this.shadowRoot.innerHTML = template
  }
}

customElements.define("the-map", TheMap);
```

Discussion

After reading this chapter's previous recipes, you may think I'm against the Shadow DOM, but that's not entirely true. Contrary to many other opinions you'll read in discussions about web components, I don't believe Shadow DOM is the unique selling point of web components. It's just one part of it that can be useful. How I use web components depends on the specific problem I'm trying to solve. Sometimes I use Shadow DOM exclusively. Sometimes, I use it for progressive enhancement. Sometimes I don't use it at all.

I don't believe that web components have been designed with accessibility in mind, but some of their core concepts encourage an accessibility-first development mindset.

Progressive enhancement

When you work with web components, you have two contexts: the *page context* (Light DOM) and the *components context* (Shadow DOM). Content can exist on the page or in the shadow tree of a component, but you can also pass content from the page to the component using the *slot element*, allowing content to exist in both contexts. The big advantage of slotting content is that the browser renders it even when JavaScript is disabled. Web components that use slots have progressive enhancement at their very core.

The markup of the basic disclosure widget in Example 12-19 consists of two parts: content you pass between the components' start and end tags, and a template that takes the content and mixes it with more HTML from the components' Shadow DOM. With JavaScript disabled, users only see the slotted content. With JavaScript enabled, which should be the default for most users, they see a button they can click to show and hide the content. An architecture like that makes the component easy for the developers to use without bothering them with the complexity that sometimes comes with progressive enhancement.

Compensating for issues

Another advantage of working with slots is that you can create multiple named slots and arrange them as you like. Example 12-20 shows a card component with three slots: heading, content, and media. In Example 12-21, you see how someone might use that component. The order of the slotted elements seems inspired by the card's typical visual presentation: image first, followed by heading and text. Starting with the heading would be better because headings introduce users to new sections or subsections. Semantically, it doesn't make sense for the image to come before the heading that introduces it.

As the component author, you don't have to worry about the developer who uses the component picking the wrong order. What counts is the order in which you arranged the slots inside the components' shadow tree. The user gets the order *you* decide is best. You can then use CSS to arrange it visually, as shown in Example 12-23:

Example 12-23. Using CSS to display the image above the heading and the content

```
the-card {
  max-width: 25rem;
  display: grid;
  border: 1px solid;
  grid-template-columns: 1rem 1fr 1rem;
}

the-card [slot="media"] {
  grid-column: 1 / -1;
  order: -1; ❶
}

the-card :is([slot="heading"],[slot="content"])


  grid-column: 2 / -2;
}
```

❶ Moves the image visually to the first row within the card.

Error reporting

Attributes provide developers with options to customize web components. Some attributes may be mandatory for accessibility. You can add validation to your components to ensure they're present, as shown in Example 12-22. The component renders the interactive map only if there's a title attribute, providing an accessible name for the embedded iframe.

How web components are designed can greatly help create inclusive experiences, but be aware of the issues Shadow DOM may cause.

See Also

- "Shadow DOM and accessibility: the trouble with ARIA" by Nolan Lawson
- "Web Components Accessibility FAQ" by Manuel Matuzović
- "How Shadow DOM and accessibility are in conflict" by Alice Boxhall

Chapter 13. Debugging Barriers

Understanding what makes an accessible website and knowing what to look for when implementing it is essential, but it doesn't protect you from making mistakes anyway.

Furthermore, you often work with others on a website. The more people are involved, the easier it is to introduce bugs. More people from different disciplines means more potential sources of error.

Automated testing software doesn't replace manual testing, but it can help catch low-hanging fruits and find sloppy mistakes. Paired with debugging tools, they allow you to prepare your websites for manual testing using keyboards, screen readers, and other assistive technology.

13.1 Find Accessibility Issues Automatically

Problem

Looking for accessibility issues by scanning the code manually is time-consuming, challenging, and error-prone.

Solution

Use automated testing tools to find easily detectable accessibility issues before you proceed to manual testing. The code in Example 13-1 includes several accessibility issues for the purpose of testing.

Example 13-1. Testing demo: Inaccessible sample code in HTML

```
<h2 aria-label="Registration">Sign Up</h2>

<form aria-labelledby="h2">
  <input type="text" name="username">

  <button>
    <img src="send.jpg">
  </button>

  <p>
    <a href="#" aria-hidden="true">Disclaimer</a
  </p>
</form>
```

You can test the code in Example 13-1 using different browser extensions. Five popular testing tools are axe DevTools (see

Figure 13-1), Google Lighthouse (see Figure 13-2), WAVE (see Figure 13-3), ARC Toolkit (see Figure 13-4), and IBM Equal Access Accessibility Checker (see Figure 13-5).

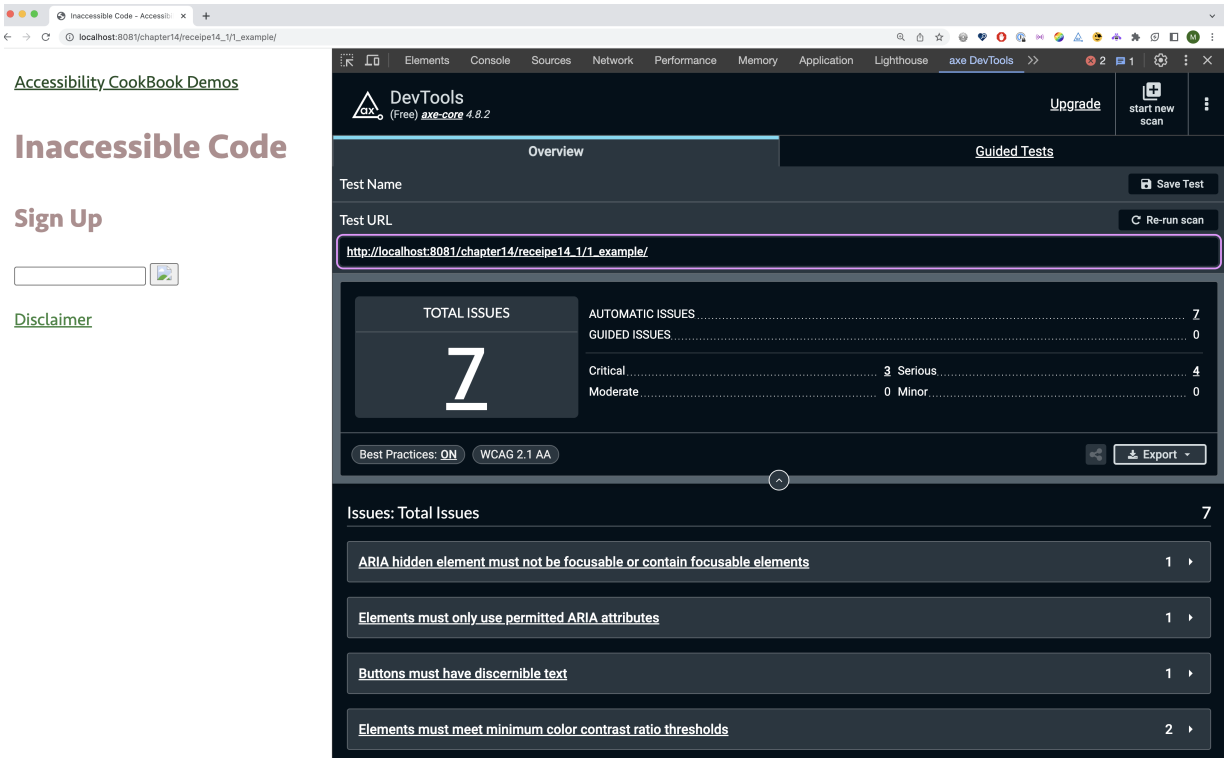


Figure 13-1. axe DevTools shows the total number of issues

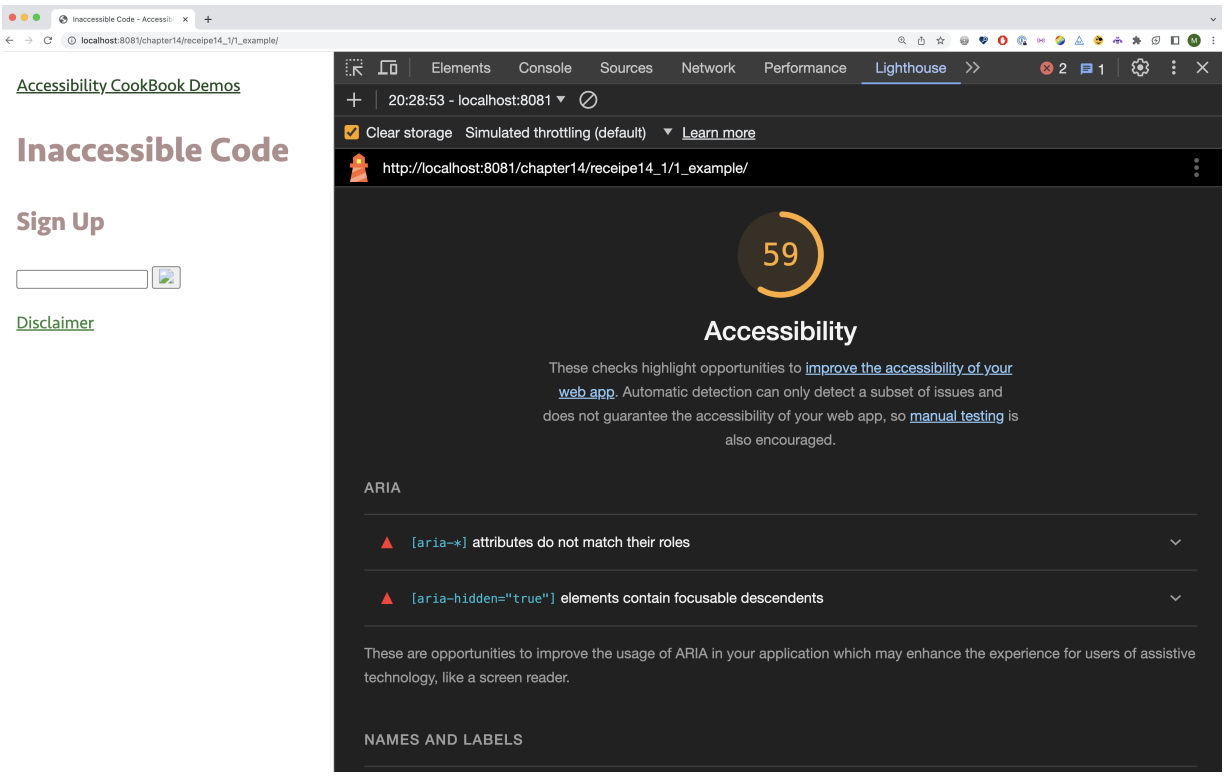


Figure 13-2. Google Lighthouse gives you a score

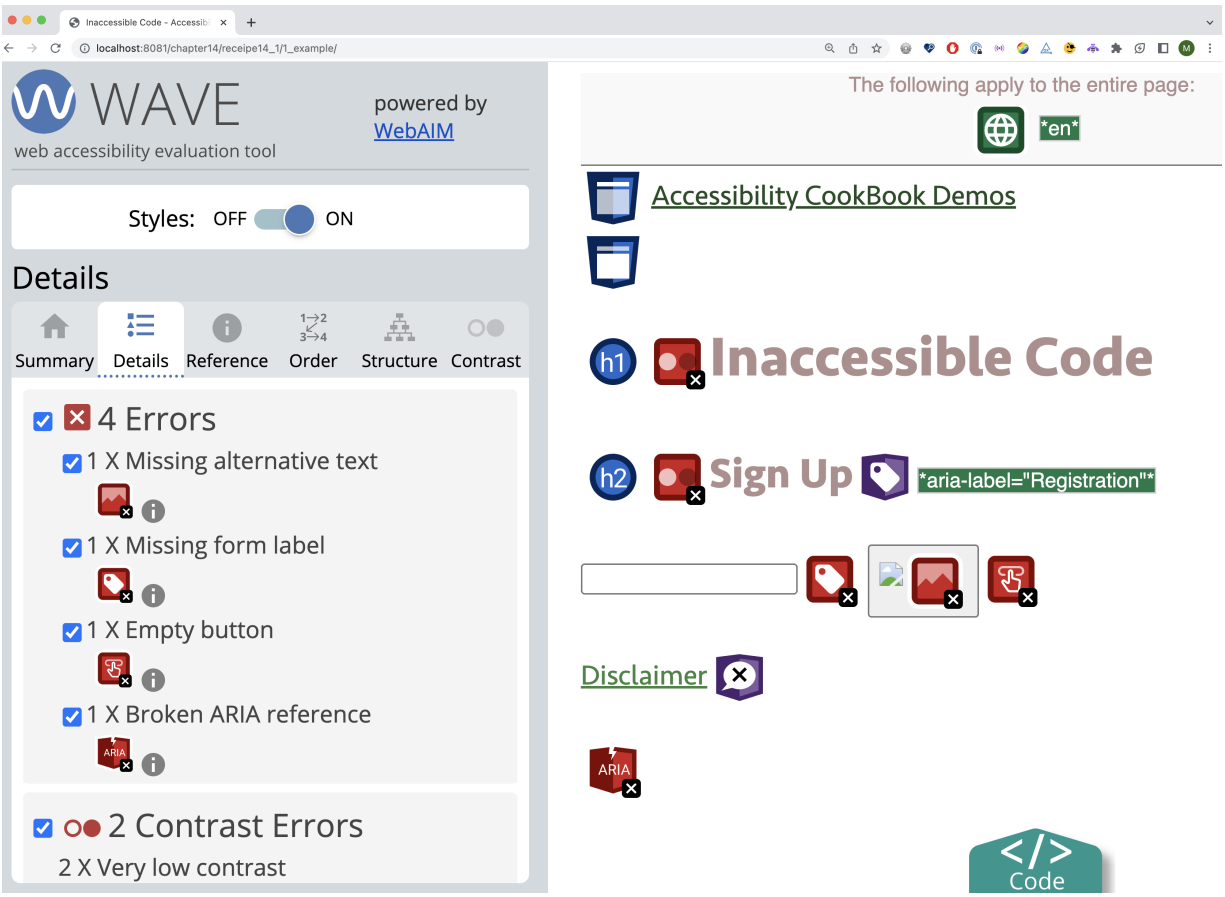


Figure 13-3. WAVE shows the number of errors and annotates affected elements

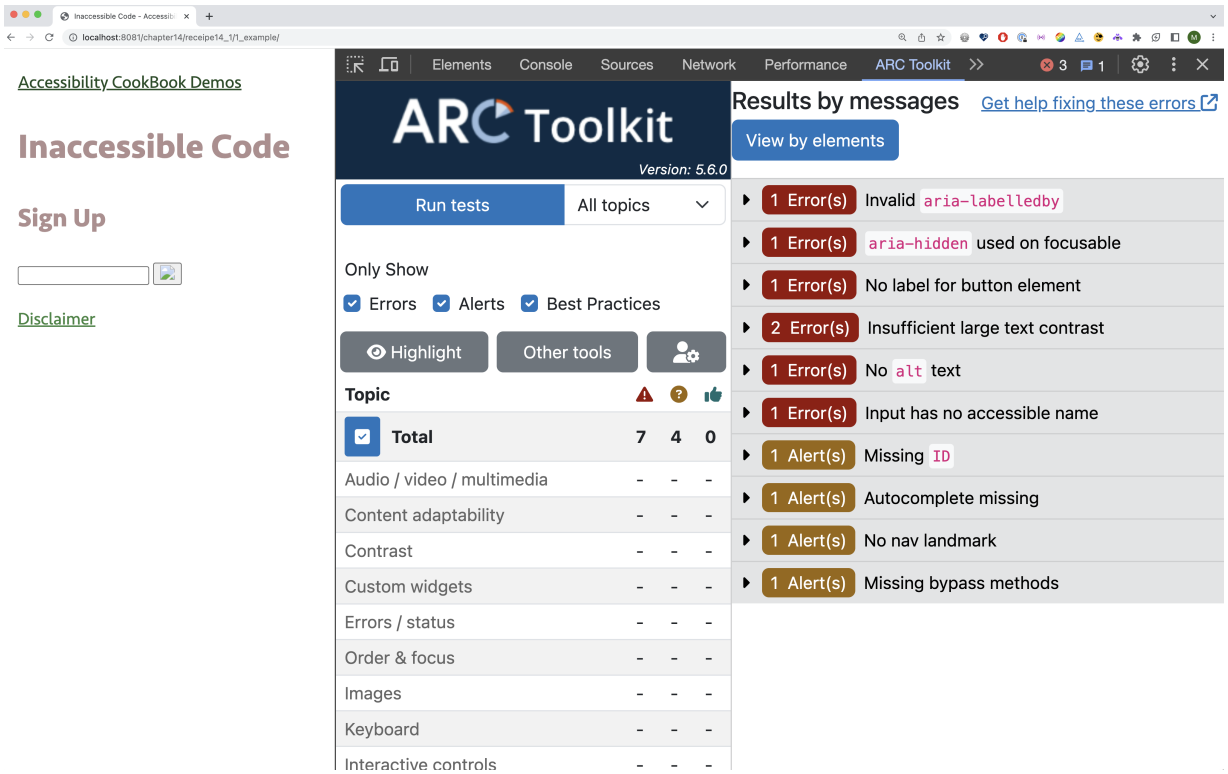


Figure 13-4. ARC Toolkit shows the total number of issues

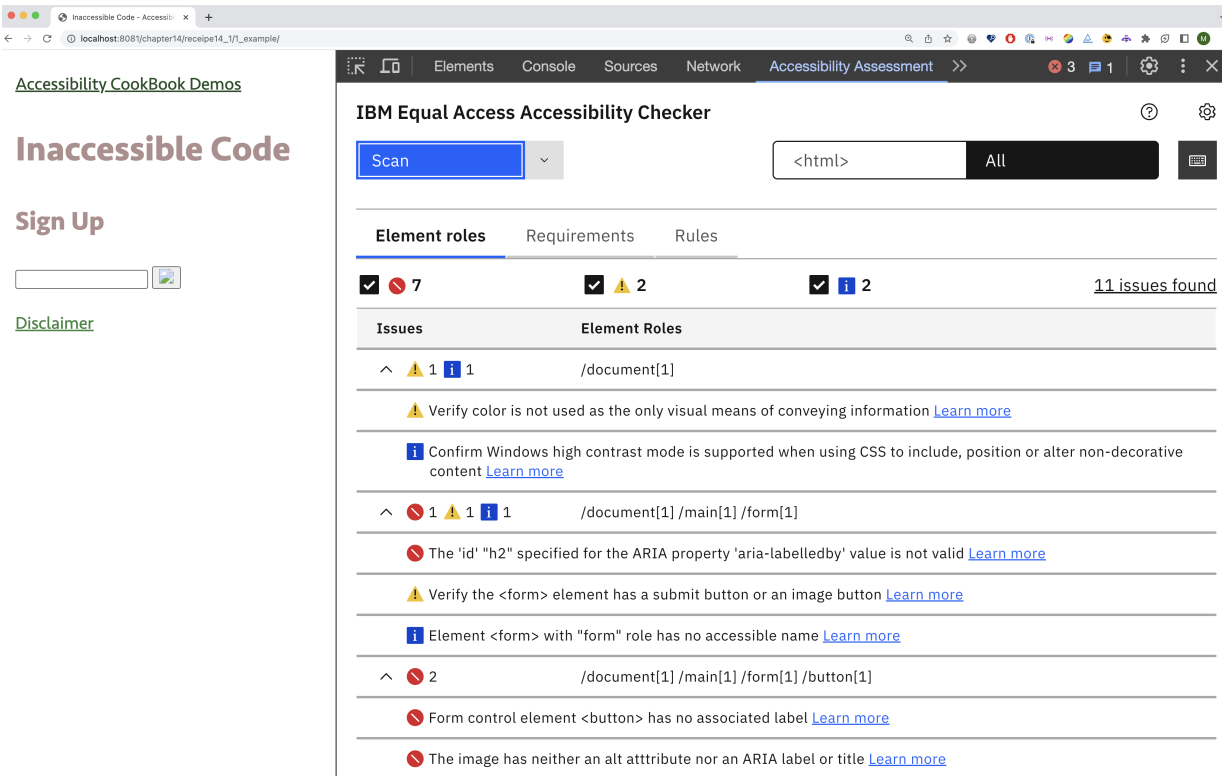


Figure 13-5. IBM Equal Access Accessibility Checker shows the total number of issues

You can run automated tests in Node.js, as shown in Example 13-2.

Example 13-2. Sample test for pa11y

```
pa11y("http://127.0.0.1:8080", {
  runners: ["axe"],
  level: "warning",
  viewport: {
    width: 320,
    height: 480,

    deviceScaleFactor: 2,
    isMobile: true,
```

```
    },
    actions: ["click element #show", "screen captu
  }).then((results) => {
    console.log(results.documentTitle);
    console.log(`Number of issues: ${results.issue
    console.log(results.issues);
  });
```

Discussion

If you look at the code in Example 13-1, how many accessibility-related issues can you spot? You may find all six bugs, but this task is much more challenging in a larger and more complex code base. There are also tests you can perform only with software, like checking color contrast. That's where automated accessibility testing tools can come in handy. They run scripts that test all elements on the page against certain rules. For example, the testing tool *axe* has a rule called "Buttons must have discernible text" that checks whether elements with the `button` role have an accessible name.

Some tools use the same rules as axe, while others have their own, which are usually very similar but different in certain aspects. That's one of many differences you'll notice comparing software.

Automated testing software

I've highlighted five tools I've used in the past and still use to give you an idea of how they differ.

axe DevTools

axe DevTools by Deque Systems uses the axe core engine, which is open source. The extension is available for Chrome, Edge, and Firefox and adds a new axe DevTools panel to your browser developer tools. Within that panel, axe reports the total number of issues, split into critical, serious, moderate, and minor, as shown in Figure 13-1. Each issue includes details, suggestions on how to fix the bug, additional learning resources, and options to highlight and inspect affected elements.

The free version should provide you with everything you need, but exporting to formats other than JSON, saving, and sharing are possible only in the paid version.

Lighthouse

Lighthouse by Google is a quality assessment tool built into Chromium-based browsers. It has audits for performance, accessibility, progressive web apps, SEO, and more. It uses the axe core engine, just like axe DevTools, but not all of the rules.

The UI is different, and instead of reporting the total number of items, it gives you a score between 0 and 100. You can export results to JSON, HTML, or PDF and share them via a URL if you connect Lighthouse to a GitHub account.

WAVE

WAVE by WebAIM is a testing tool for Chrome, Firefox, and Edge that uses its own engine. Unlike other tools, it's not integrated into the browser's developer tools, but adds an icon to the extensions toolbar and a "WAVE this page" option to the context menu in your browser viewport. Instead of simply listing errors, it also annotates affected elements using different icons, as you can see in Figure 13-3. Besides errors, it also lists and annotates features like ARIA attributes, and it has a contrast checking and tweaking tool built-in.

ARC Toolkit

ARC Toolkit by TPGi is a Chrome extension that adds an *ARC Toolkit* panel to your browser developer tools. It lists errors, warnings, and best practices, as shown in Figure 13-4. In addition to testing, you can highlight certain elements, like links, images, or buttons, and send the page directly to an HTML and ARIA validator.

IBM Equal Access Accessibility Checker

IBM Equal Access Accessibility Checker by IBM is a testing tool for Chrome and Firefox that uses its own engine and adds a new “Accessibility Assessment” panel to your browser developer tools. It lists errors, warnings, and recommendations you can store and download as HTML and XLSX, as shown in Figure 13-5.

Continuous testing

You don’t necessarily have to open the browser to run automated tests on your websites. Pa11y is particularly interesting because it allows you to automate things like resizing the browsers, performing actions on the site before you run tests, and taking screenshots. In Example 13-2, you can see an example config that uses axe for testing, resizes the viewport, clicks an element with the ID *show*, takes a screenshot, runs tests, and then returns an object with the results.

You can run tests on the command line:

- Lighthouse CLI
- axe CLI
- pa11y CLI

You can use some testing scripts in Node.js:

- Lighthouse
- axe-core
- pa11y

There are also tools you integrate into your build pipeline:

- Lighthouse CI
- Lighthouse GitHub Action
- Pa11y CI

Features

There are hundreds of tools you can pick from. Which tools you decide to use depends on several factors:

Guidelines

Most tools support the latest version of the WCAG, but depending on the organization or government you're working for, you may need to test against other rules.

Software environment

Some tools support only specific browsers and operating systems.

Who's testing?

Depending on who's performing the test (developers, designers, QA, etc.), the UI and feedback the software gives should be more or less technical.

Sharing and comparing

Do you need to compare test results with previous tests? Do you need an easy way to share results with stakeholders?

Error reporting

Tools handle feedback and error reporting differently. Some list only errors; others also annotate affected elements.

Report format

Some tools allow you to export results as HTML, CSV, JSON, etc.

Issue filing

Do you need the ability to file issues in a ticketing system directly from within the tool?

Personal preference

Some details in the UI and UX may work better than others for you.

The WAI provides more factors you may want to consider when picking testing tools in their guide "Selecting Web Accessibility Evaluation Tools".

Try the extensions listed in this recipe and see what works best for you.

Strengths and limitations

Automated accessibility testing tools are perfect for quickly catching low-hanging fruits, those types of errors that are detectable programmatically. However, accessibility is about humans and human perception, context, and nuance—things you can't test automatically.

Automated testing is a significant first step before you proceed to manual testing. It's also helpful in repeating tests at different stages in a site's lifecycle. Another strength is that you don't necessarily need accessibility or development knowledge to run and understand tests. You must interpret the results correctly. Explanations and additional learning resources linked in most tools can help you.

Automated accessibility testing software doesn't catch all of the accessibility errors, which means that 0 errors reported or a perfect score don't mean your site is accessible. It suggests only that you have a good baseline for testing with the keyboard, screen readers, and other assistive technology.

13.2 Explore the Accessibility Tree

Problem

When you test a page for accessibility issues with automated or manual testing tools, it's not always clear what the source of a particular bug or behavior is. Sometimes, you must dig deeper and explore the data the browser sends to assistive technology to find the root cause of an issue.

Solution

Analyze the accessibility tree using built-in browser tools. Use the simple code snippet in Example 13-3 to analyze its representation in the accessibility tree.

Example 13-3. Random sample code in HTML

```
<article>
```

```
    <h2>About</h2>
    <button aria-expanded="true">Details</button>
    <a href="#" aria-current="page">Website</a>
  </article>
```

You can view the representation of the code in Example 13-3 in the page's accessibility tree using your browser's developer tools.

In Chromium-based browsers like Chrome, Edge, or Polypane, you can select a node in the *Elements* panel and analyze it by navigating to the *Accessibility* pane, as shown in Figure 13-6.

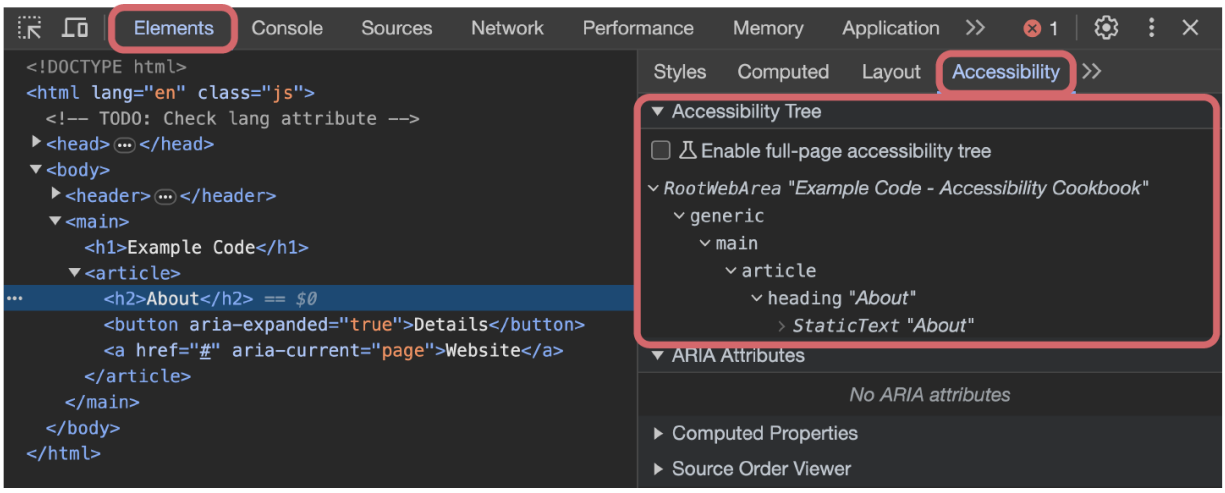


Figure 13-6. A heading selected in the Elements panel and its representation in the accessibility shown in the Accessibility pane

By checking the *Enable full-page accessibility tree* option in the *Accessibility* pane, you can enable the full accessibility tree. A

new button in the Elements panel lets you switch between the DOM and the accessibility tree, as shown in Figure 13-7.

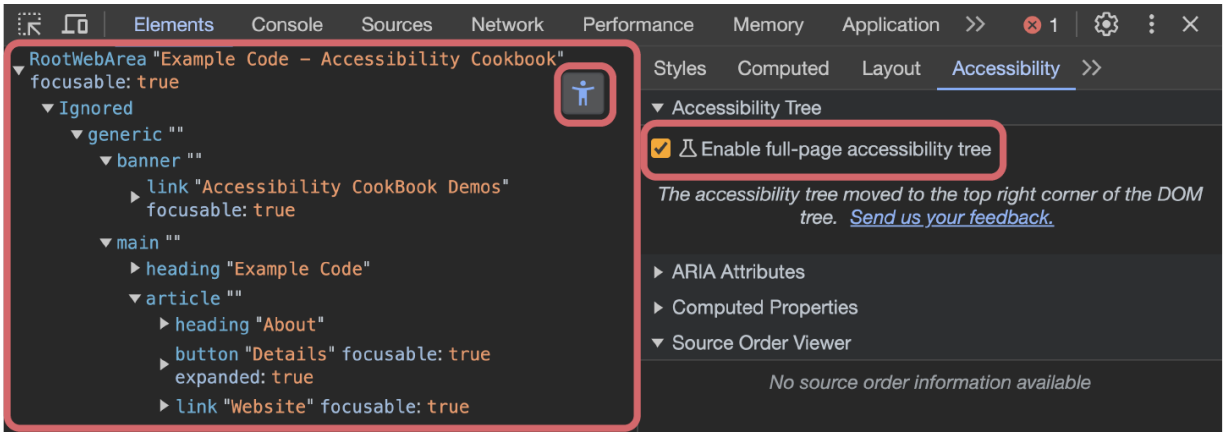


Figure 13-7. DevTools showing the accessibility tree instead of the DOM

Firefox shows the full accessibility tree by default in its *Accessibility* panel, as shown in Figure 13-8.

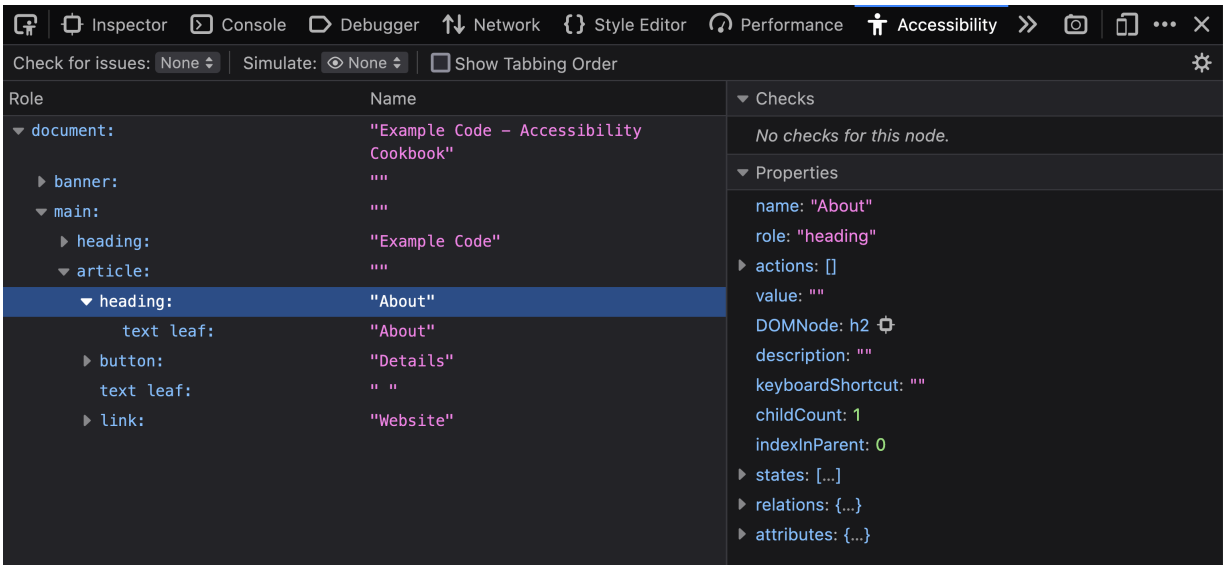


Figure 13-8. Firefox DevTools showing the full accessibility tree

Discussion

When you create an `.html` file and write HTML, you "only" create a file composed of text and special characters. However, when you open it in the browser, the HTML parser takes your text file, analyzes it, and constructs a tree-like representation of it, the DOM. The DOM is a programming interface for web documents that represents the document as nodes and objects. It contains all the information that makes up the page and provides an API to manipulate it.

Derived from the DOM, there is another tree—the accessibility tree. It's similar to the DOM but simpler, because it contains only semantic information in the form of accessible objects that express the structure of the UI. There are different accessibility APIs on each operating system that take that information and pass it to assistive technology, like screen readers.

An object within this tree typically contains its role, name, and state. As you can see in all figures in the Solution section, especially in Figure 13-7, there are several details the accessibility API gets from the accessibility tree about the `<h2>` and passes on to screen readers.

- The `role` of the `<h2>` is *heading*.

- Its accessible name is *About*.
- It has a parent `article` with no accessible name.
- It has two siblings: a focusable element with the `button` role, *Details* as its accessible name, and the `expanded` state set to true, and another focusable element with the `link` role and *Website* as its accessible name.

Access to that information before it's passed to assistive technology via accessibility APIs proves helpful for debugging.

In Chromium-based browsers, you can select a node in the DOM and analyze the DOM tree in the accessibility pane, as shown in Figure 13-6. That's fine, but you get a better developer experience by switching to the full accessibility tree, as shown in Figure 13-7. Instead of showing a part of the tree, you can analyze the tree as a whole and switch back and forth between it and the DOM. You also get nicer syntax highlighting. In Firefox, you always get the full accessibility tree by default (see Figure 13-8).

13.3 Debug Roles, Names, Properties, and States

Problem

Debugging HTML is often a matter of manually inspecting code using developer tools and analyzing the DOM. That works well when the code is simple, but with an increased amount of attributes and relations to other elements, it gets more challenging. That's especially true when you work with ARIA attributes that overwrite native semantics and add properties or states to elements. Complex markup makes it often impossible to tell why a given solution doesn't work as expected.

Solution

Learn about an element's accessibility features by highlighting it in your browser's developer tools, as shown in Figures 13-9 and 13-10.

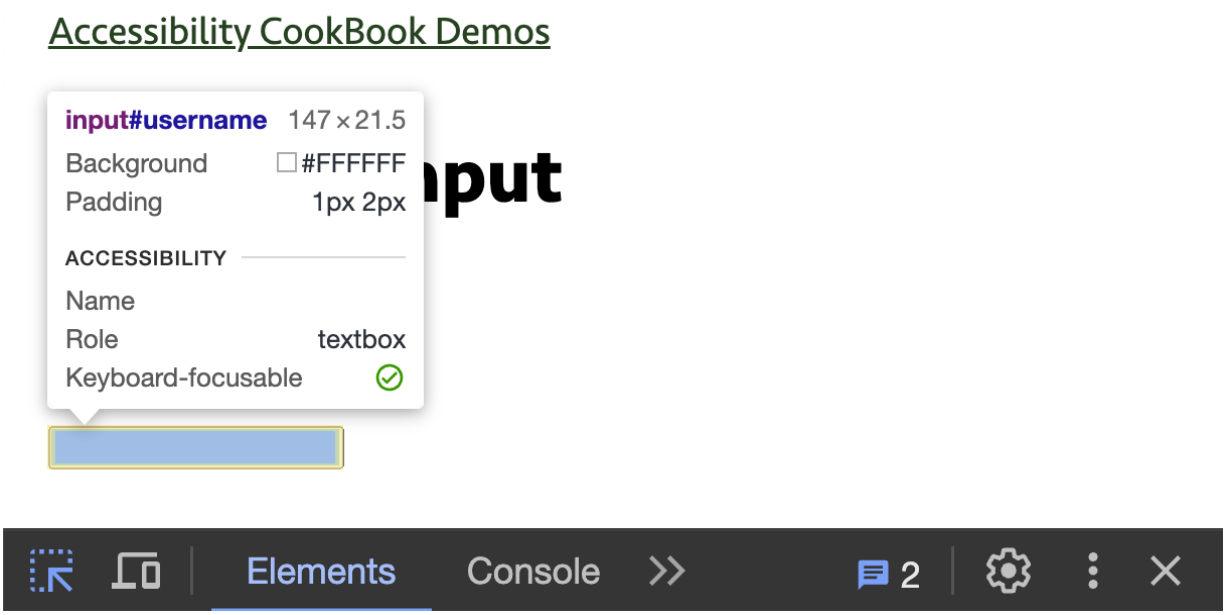


Figure 13-9. A tooltip in Chrome showing the element's role and an empty name

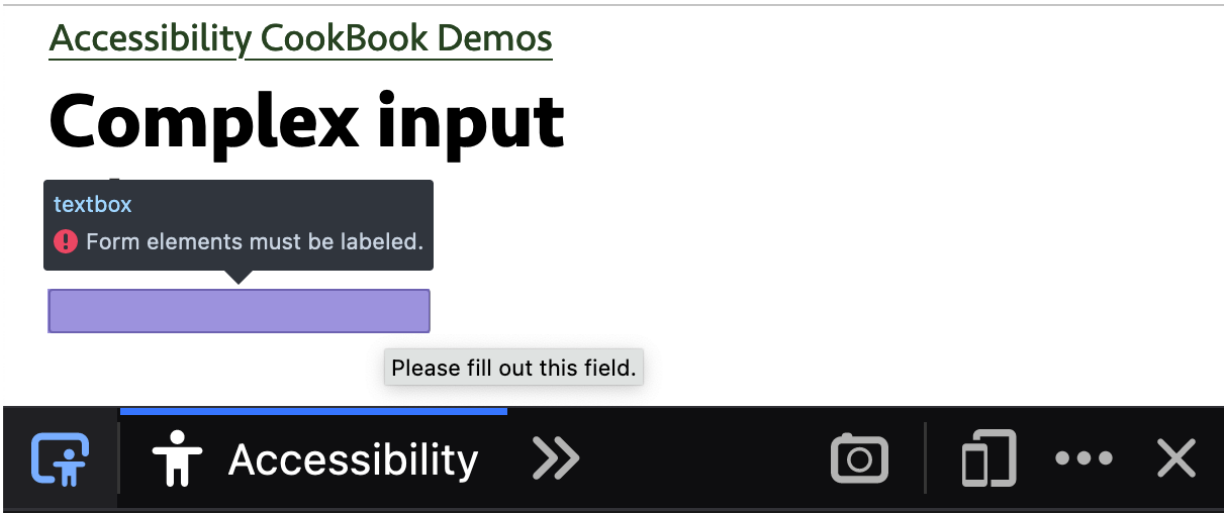


Figure 13-10. A tooltip in Firefox showing that element's role is "textbox" and that it must be labeled

Find information about roles, names, properties, and states in developer tools accessibility panels, as shown in Figure 13-11.

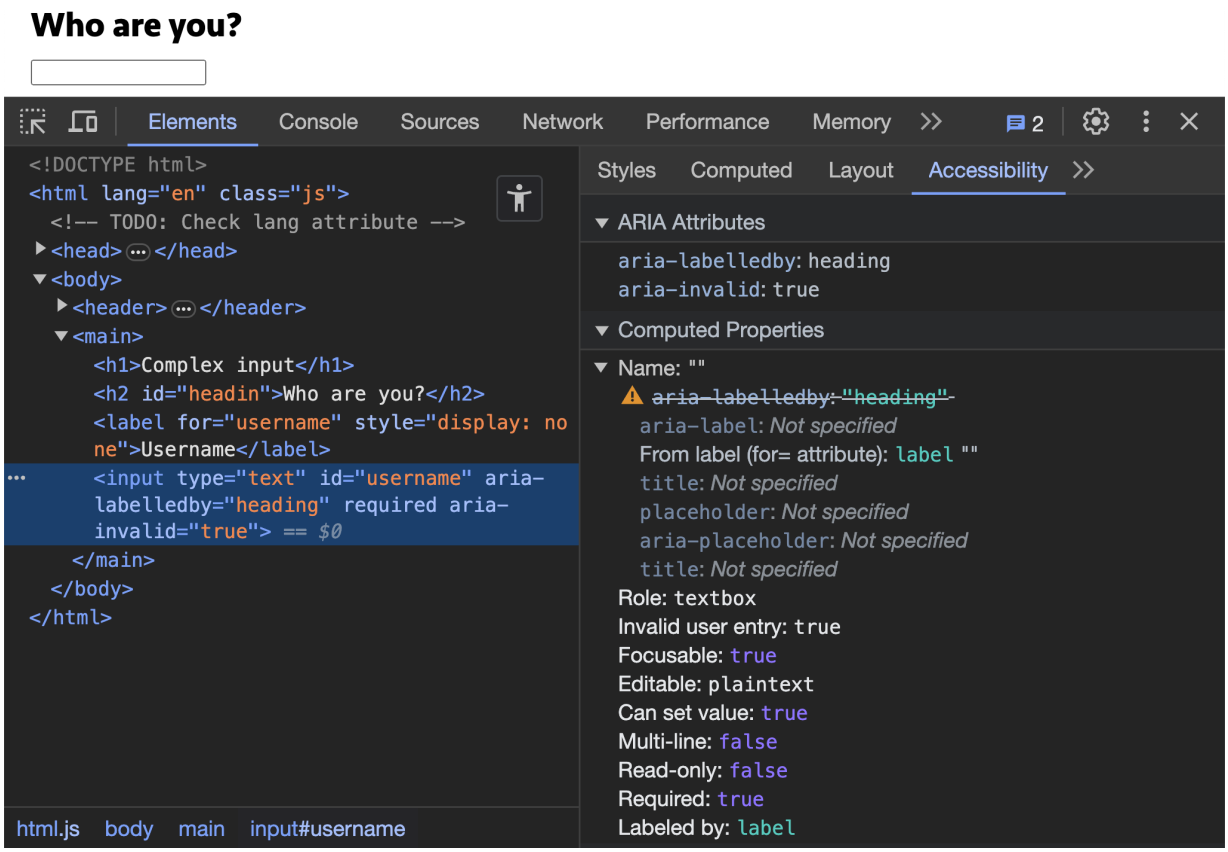


Figure 13-11. The Accessibility panel in Chrome listing ARIA attributes and computed properties

Use the code in Examples 13-4 and 13-5 for debugging in this recipe.

Example 13-4. Demo debugging code: Complex labeling on an input field

```
<h2 id="headin">Who are you?</h2> ❶
<label for="username" style="display: none">User

<input type="text"
       id="username"
       aria-labelledby="heading" ❸
```

```
        aria-invalid="true">
```

1. Typo in the ID, a *g* is missing
2. Hidden label
3. Reference to the heading

Example 13-5. Demo debugging code: Visually identical buttons, but only one is a *real* button

```
<style>
  .btn {
    display: inline-block;
    border: none;
    line-height: 1;
    font-size: 1rem;
    font-family: inherit;
    text-decoration: none;
    background: #123456;
    padding: 1em;
  }

  .btn, .btn:is(:link, :visited) {

    color: #fff;
  }
```

```
    :focus-visible {
      outline: 4px solid #123456;
      outline-offset: 2px;
    }
  </style>

  <button class="btn"> ❶
    Button 1
  </button>

  <a class="btn" href="#"> ❷
    Button 2
  </a>

  <div class="btn"> ❸
    Button 3
  </div>

  <div class="btn" tabindex="0"> ❹
    Button 4
  </div>

  <div class="btn" role="button" tabindex="0"> ❺
    Button 5
  </div>
```

❶ A real button

2. A link that looks like a button
3. A `div` that looks like a button
4. A focusable `div`
5. A focusable `div` with a button role

Discussion

The first rule of ARIA use says that *"If you can use a native HTML element or attribute with the semantics and behavior you require already built in, instead of re-purposing an element and adding an ARIA role, state, or property to make it accessible, then do so."* In other words: avoid using ARIA, unless you can't achieve the desired result with HTML only. One of the reasons this rule exists is that your code gets significantly more complex once you add ARIA roles, properties, and states. However, you often need ARIA to make elements semantically richer or to compensate for the lack of support for native elements or attributes.

Tooltips

To simplify debugging, browsers offer several ways of accessing semantic information. In Chrome, you can use the inspection

tool to show a tooltip when you hover over an element. Besides other information, the tooltip lists the element's role and the accessible name, as shown in Figure 13-9. Firefox shows the role and name or an error message if there's no name, as you can see in Figure 13-10. To get this information, you must select the *Accessibility* panel first.

Detailed access to roles, names, properties, and states

If you want to dig deeper, switch to the Accessibility panel in Firefox or the Accessibility pane in Chrome. In Figure 13-11, you can see how Chrome lists all ARIA attributes associated with the element. The computed properties section summarizes the element's role, name, properties, and states. There's a lot of helpful information you can extract about the input:

- There's an ARIA reference, but there's something wrong with it.
- The name's coming from a label element, and it's empty.
- It's a required field.
- It contains invalid data.

Here's another example: Figure 13-12 shows five identical buttons. They look the same, but they behave differently and

communicate different information. In addition to testing with a keyboard or screen reader, you can use the Accessibility panel to learn whether they meet the essential accessibility criteria for a button listed in Recipe 4.1.

Figure 13-12. The buttons look the same, but they don't behave the same

If you debug the buttons (for the code, see Example 13-5), you get the following results:

- **Button 1**: Name: "Button 1," Role: button, Focusable: true
- **Button 2**: Name: "Button 2," Role: link, Focusable: true
- **Button 3**: Name: "Button 3", Role: generic
- **Button 4**: Name: "Button 4," Role: generic, Focusable: true
- **Button 5**: Name: "Button 5," Role: button, Focusable: true

With Button 3, you can see that debugging in the Accessibility pane is not always about the information displayed but also about information missing. It doesn't say, "Focusable: false," it omits the property entirely, which means that this button is not focusable.

13.4 Visualize Tabbing Order

Problem

Visual and DOM order are sometimes so heavily out of sync, that debugging using a keyboard or a similar device becomes daunting. Also, when your job is to communicate tabbing order bugs to others, verbalizing the underlying problem with only text can be challenging.

Solution

Use tools to visualize tabbing order, as shown in Figure 13-13.

Figure 13-13. The Polypane browser visualizing tab stops

Discussion

The code in Example 13-6 looks harmless: it creates a grid and explicitly places some links in it. When you tab through the page, you'll notice that the order in which items receive focus isn't predictable.

NOTE

Recipe 6.5 describes why that can be problematic, and it explains that DOM order should match the visual order as well as possible.

Example 13-6. A grid of linked images

```
<style>
  .grid {
    display: grid;
    grid-template-columns: repeat(3, 12rem);
    grid-auto-rows: 100px;
    grid-gap: 1rem;

    grid-auto-flow: dense;
  }

  .link1, .link3 {
    grid-row-end: span 3;
  }


  .link4 {
```

```
    grid-column: span 2;
  }

  .link5, .link9 {
    grid-row: span 2;
  }

  .link7, .link8 {
    grid-column: 2 / span 2;
  }

  .grid img {
    object-fit: cover;
    width: 100%;
    height: 100%;
  }

</style>

<div class="grid">
  <a href="#" class="link1">
    <img src="/image_1.jpg" alt="Train on a trac
  </a>
  <a href="#" class="link2">
    <img src="/image_2.jpg" alt="Cold juicy beve
  </a>

  <a href="#" class="link3">
    <img src="/image_3.jpg" alt="Yellow floating
```

```
    </a>
    <a href="#" class="link4">
      <img src="/image_4.jpg" alt="Yellow leaves o
    </a>
    <a href="#" class="link5">
      <img src="/image_5.jpg" alt="Stacked boxes o
    </a>
    <a href="#" class="link6">
      <img src="/image_6.jpg" alt="Close-up of a h
    </a>
    <a href="#" class="link7">
      <img src="/image_7.jpg" alt="Sunsetvibes in
    </a>
    <a href="#" class="link8">
      <img src="/image_8.jpg" alt="Slices of apple
    </a>
    <a href="#" class="link9">
      <img src="/image_9.jpg" alt="A gecko climbin
    </a>
  </div>
```

You can show the tab order and sometimes even automate tabbing by using a tabbing order visualization tool. Figure 13-13 shows the focus outline option in the Polypane browser. It annotates each interactive element with a number when you activate it and connects them in the order of the tab sequence.

Other browsers or extensions offer similar functionality:

- “Show source order” in Chrome’s *Accessibility* pane.
- “Show tabbing order” in Firefox’s *Accessibility* panel.
- *Tab stops* option in Accessibility Insights’ *Ad hoc tools*.
- *Tab order* option in ARC Toolkit’s *Highlight* section.
- *Keyboard Checker Mode* in IBM’s Equal Access Accessibility Checker.
- The reading order bookmarklet by Adrian Roselli for all browsers that support bookmarklets, including Safari.

This feature can be convenient, especially when describing the problem to others.

13.5 Emulate Color Deficiencies, Reduced Motion, and More

Problem

Operating systems provide users with more and more options to tweak certain parts of their setup according to their needs. With CSS, HTML, and JS, you can query some of those user preferences and adapt your UIs accordingly. To do that and test

your changes, you must apply these settings to your operating system.

Changing how the whole operating system looks and behaves to test a web UI can be annoying and disturbing because some settings are pretty invasive regarding the user experience. Also, not all operating systems support all settings, so testing your adaptions can be impossible.

Solution

Emulate reduced motion and other settings in your browser developer tools, as shown in Figures 13-14, 13-15, and 13-16.

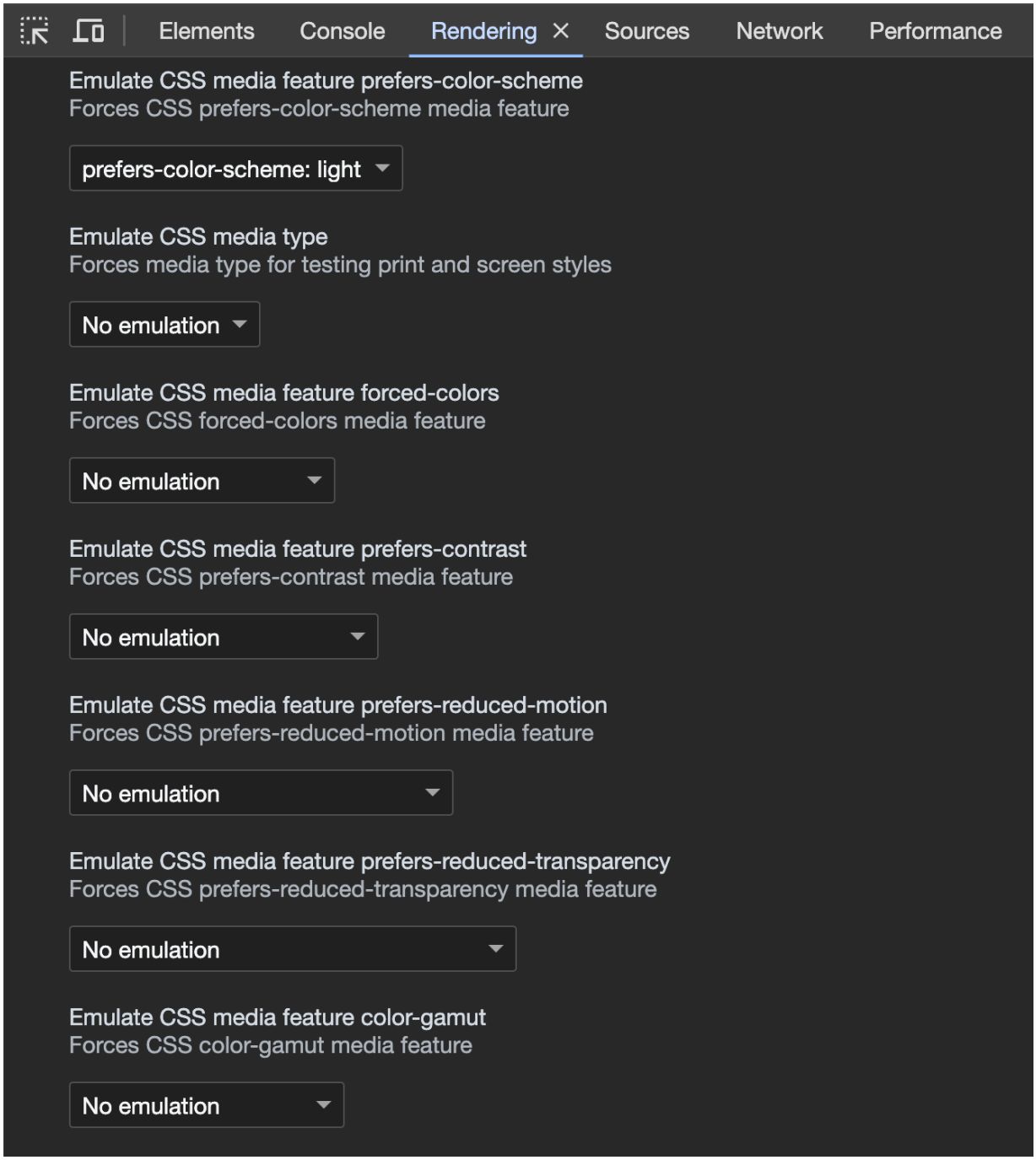


Figure 13-14. Rendering tab in Chrome DevTools

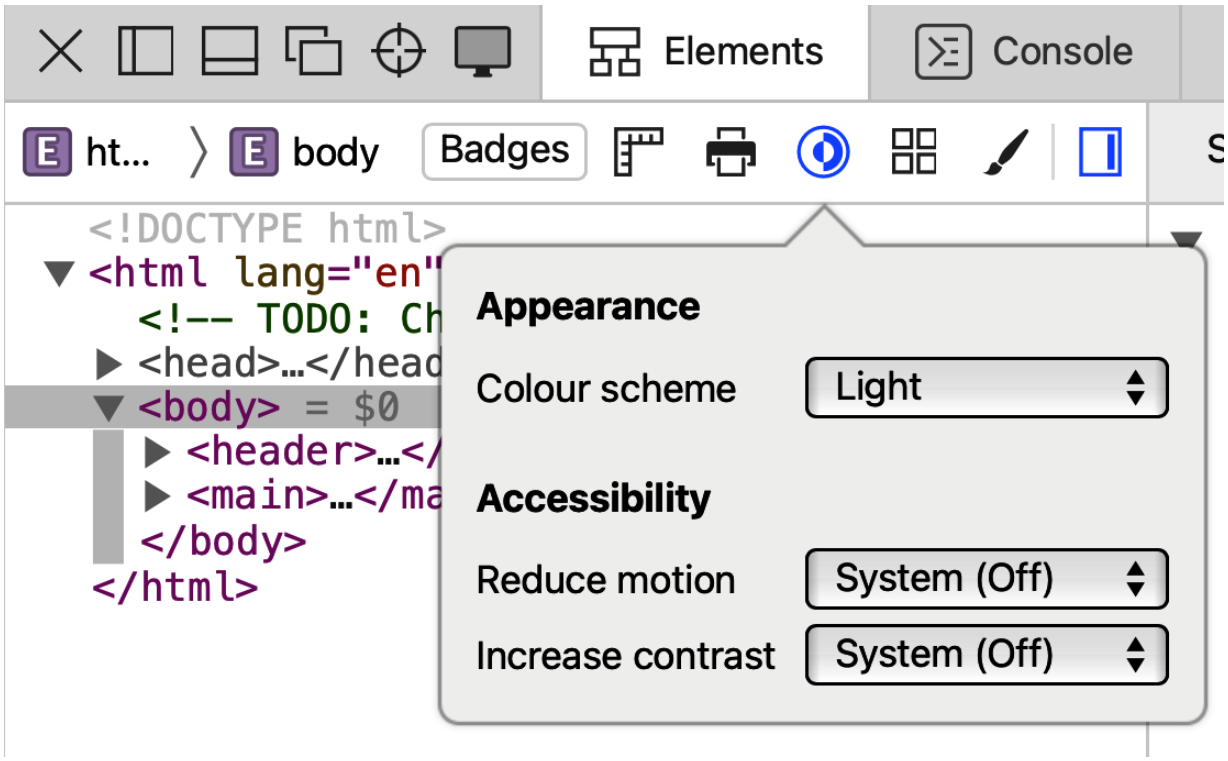


Figure 13-15. “Override user preferences” popover in Safari’s Web Inspector

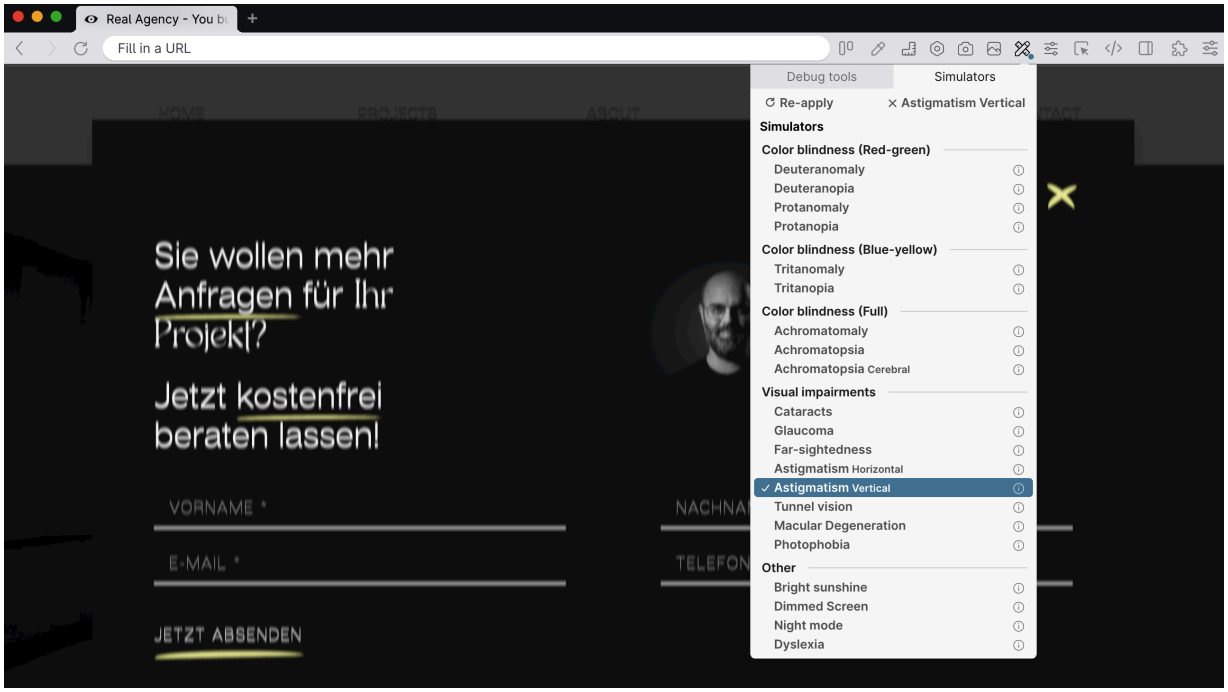


Figure 13-16. Simulators in the Polypane browser

Discussion

In Recipe 5.2, you've learned that respecting user preferences is one of the most critical aspects of designing and building inclusive UIs. Operating systems offer different options to tweak UIs and behavior according to their needs. For example, users can reduce the motion in UIs on Windows, macOS, iOS, and Android. In your CSS, you can query those settings and remove or minimize onscreen movement accordingly. To test your changes, you can find the appropriate settings in your OS and activate them. It's wise to test your code using the actual native feature in the OS and not to rely only on emulation, but having to constantly switch between the browser and OS settings during development can be annoying.

Luckily, some browsers offer options to emulate specific user preferences. Chromium-based browsers provide the most options, as you can see in Figure 13-14. When you select "prefers-reduced-motion: reduce," the browser acts as if the OS setting is active as long as your developer tools are open. That enables you to switch between different modes quickly. In Safari, you can press the "Override user preferences" button in the Elements panel to emulate the preference, as shown in Figure 13-15.

Aside from reduced motion, you can also emulate other settings:

Chrome:

- Reduced motion
- Forced colors
- Contrast
- Color scheme
- Reduced transparency

On top of that, you can also emulate print stylesheets and simulate different vision deficiencies in Chrome's *Rendering* panel.

Safari:

- Reduced motion
- Contrast
- Color scheme
- Print styles (*Elements* panel)

Firefox:

Firefox has no option for emulating reduced motion. Still, you can simulate several color deficiencies in the *Accessibility* panel

and the print media type and color scheme options in the *Inspector* panel.

In the *Colors* section of the browser settings, you can define a custom forced-colors theme under *Manage colors*.

Polypane:

Polypane, the browser for web developers, offers the most accessibility emulation and simulation options. In addition to everything all the other browsers can do, you can simulate things like bright sunshine, a dimmed screen, or dyslexia, as shown in Figure 13-16.

Please note that all simulations and emulations can only give you a vague idea of how people perceive your content. You shouldn't rely solely on them, but test with real users.

All these settings are a great time-saver and helpful for development purposes, but they don't replace testing. You should also test your conditional code in different operating systems with the actual settings.

13.6 Write Custom Debugging Rules

Problem

There are situations where you can't use any of the tools mentioned in the previous recipes:

- You can't access developer tools in the browsers and devices you're testing with.
- You are not allowed to install third-party tools in your browser.
- Extensions may not meet your company's security guidelines.
- Features and settings in your browser are limited due to corporate regulations.
- Specific extensions are not available for your OS or browser.

Solution

Use or create tests and simulations written in CSS or JavaScript, as shown in Examples 13-7 through 13-12.

Example 13-7. Using axe core directly in your website

```
<script src="https://unpkg.com/axe-core@4.8.2/ax
<script>
const errors = []

  axe
  .run()
```

```
    .then(results => {
      if (results.violations.length) {
        results.violations.forEach(error => {
          errors.push({
            id: error.id,
            description: error.description,
            help: error.help,
            nodes: error.nodes
          })
        });

        console.error('Accessibility issues found'
        console.table(errors)
      }
    })
    .catch(err => {
      console.error('Something bad happened:', err
    });
  </script>
```

You can also write tests in CSS.

Example 13-8. Highlighting images without `alt` attribute in CSS

```
  img:not([alt]) {
    border: 4px solid red;
```

```
}
```

Example 13-9. Testing whether a natural language is set on the root element

```
html:not([lang]),
html[lang*=" "],
html[lang=""] {
  border: 10px solid red;
}
```

Example 13-10. A class in CSS that removes all color from the page

```
.a11y-tests-grayscale {
  filter: grayscale(100%) !important;
}
```

Example 13-11. A class that blurs the whole page

```
:root {
  --a11y-blur: 2px;
}

.a11y-tests-blur {
  filter: blur(var(--a11y-blur)) !important;
}
```

You can also write your own tests in JavaScript.

Example 13-12. A script that checks the presence of elements that match certain selectors

```
const check = [
  "a:not([href])",
  'a[href="#"]',
  "a[tabindex]",
  '[role="menuitem"]',
  '[role="button"]',
  '[role="link"]',
  "label a",
  "button a",
  "a button",
  "section > section",
];

let results = 0;

console.log(`%cStuff that probably shouldn't be

for (el of check) {
  if (document.querySelector(el)) {
    results++;
    console.warn(`${results}. Found "${el}". Ple
    console.log(document.querySelector(el));
```

```
    }
  }

  if (!results) {
    console.info("Nothing found, looks good!");
  }
```

Discussion

Browsers have many debugging and testing features built-in, but third-party tools bring a lot of additional functionality that the browser may not provide. They make my professional life much easier, but some developers are in a situation where they can't choose the browser they develop with or they don't have the privileges to install extensions.

Using axe locally

Luckily, some testing tools come in different flavors. The axe testing tool, for example, is available as a browser extension, a command-line interface, a GitHub action, and a standalone JavaScript file that you can include in your website.

WARNING

Don't include axe in production websites; use it only in local development environments because it can impact performance negatively.

In Example 13-7, you can see that after embedding the file, you can access the axe object, which contains comprehensive information about the rules it runs. You can list all errors, if any, by accessing the violations field.

Testing with CSS

Another exciting way of testing accessibility without a dependency on third parties is writing testing rules in CSS. You could create a CSS file called accessibility-tests.css and include it in your local development environment. In this file, you use selectors that match elements that violate accessibility rules. In Example 13-8, every image that doesn't have an `alt` attribute, and thus no accessible name, shows a red border. In Example 13-9, the `html` element shows a red border if it has no or an empty `lang` attribute. Those are just examples; there's a lot more you can do with CSS. The a11y.css project by Gaël Poupard illustrates that well. It's an extensive set of CSS testing rules available in nine languages.

Simulating with CSS

CSS is great for finding bugs, but you can also use it for basic simulations. The rule in Example 13-10 uses the filter property in CSS to simulate the absence of color. In Example 13-11, the `blur()` function simulates blurred vision. You can adjust the amount of blurriness by changing the `--a11y-blur` custom property.

Testing with JS

Testing tools usually only report accessibility violations according to specific guidelines. Some also have additional rules that test UIs for best practices, but these rules are general and don't know anything about your project and setup. You can use custom JavaScript to test for additional code patterns or rules tailor-made for your projects. The code in Example 13-12 looks for certain elements on the page that aren't necessarily breaking any WCAG rules but still look suspicious. For example, a link without an `href` or just a simple `#` as the `href` value is usually an indicator the button element should be used instead.

In Chrome, you can save those snippets and run them by using shortcuts. Go to the *Sources* panel in DevTools and find the *Snippets* pane. Click "new snippet" and name it. Add your JS

code to the snippet and run it by pressing Cmd/Ctrl + Enter (see Figure 13-17). You can call the script anywhere in DevTools by pressing Cmd/Ctrl + P and typing `![name]`, e.g., `!mytests`.

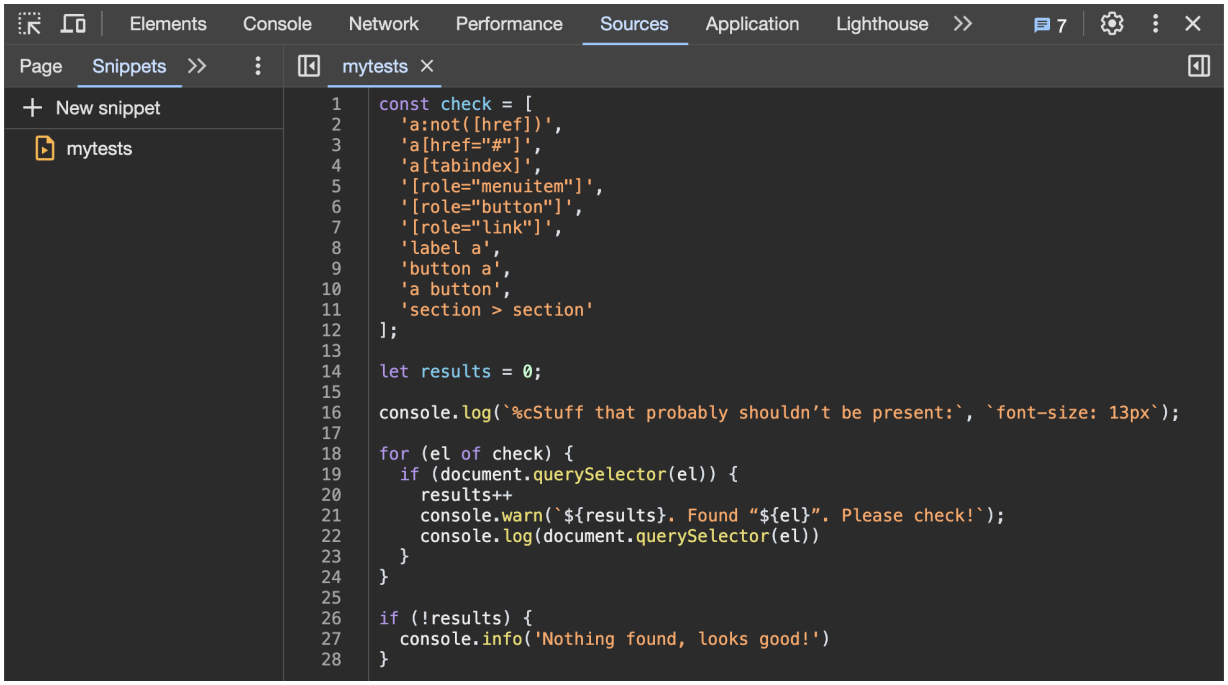


Figure 13-17. The Snippets pane in Chrome DevTools

You can check all kinds of things. For example, I have a snippet named doc info, that gives me general information about a document (see Figure 13-18).

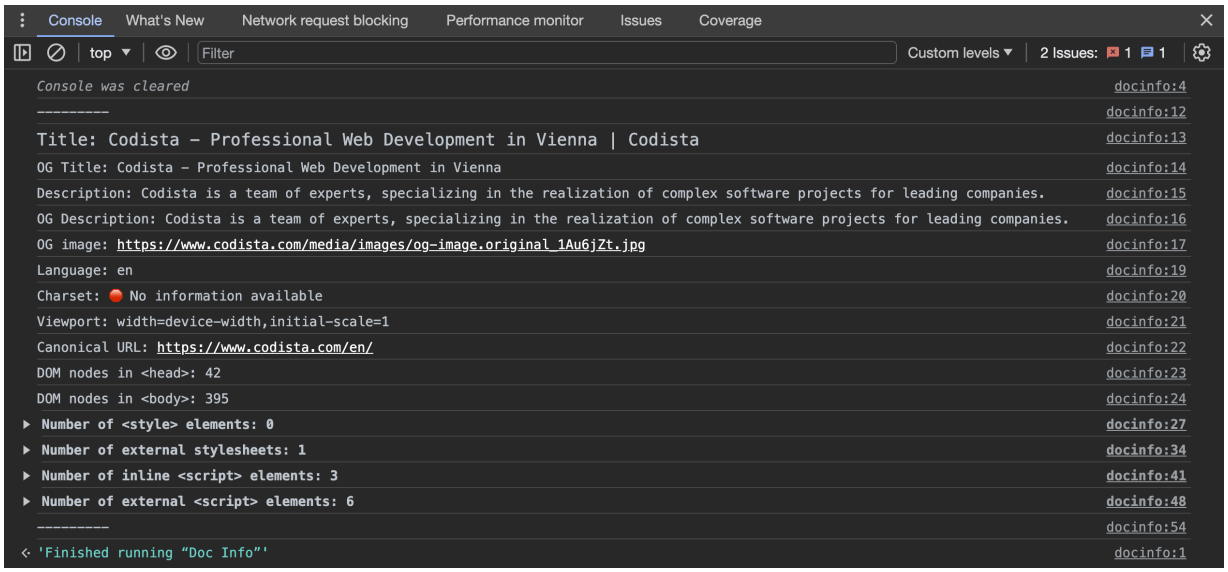


Figure 13-18. The console in Chrome DevTools after I ran the doc info script

CSS and JS can be powerful little helpers, but custom rules in CSS or JS don't replace algorithmic simulation or real user testing.

See Also

- "Full accessibility tree in Chrome DevTools" by Johan Bay
- "Accessibility APIs: A Key To Web Accessibility" by Léonie Watson and Chaals McCathie Nevile
- Elements Tab in Safari
- Accessibility overview in Polypane
- "52 Accessibility Bookmarklets You Can Use For A11Y Testing" by Raghavendra Satish Peri

- “Generating Accessibility Test Results for a Whole Website With Pa11y CI” by Matthias Ott

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

pinch-zoom and media queries, Media Queries

About the Author

Manuel Matuzović is a frontend developer, consultant, accessibility auditor, and teacher with over 15 years of experience creating websites. He has helped cities, universities, retail stores, ecommerce sites, and small and large businesses create accessible products.

Manuel is passionate about HTML and CSS, and enjoys sharing his knowledge and experience in blog posts and at meetups and conferences worldwide.

Colophon

The animal on the cover of *Web Accessibility Cookbook* is a Nova Scotia Duck Tolling Retriever. This breed of medium-sized gundog is the smallest retriever, and it was developed in the early nineteenth century by the Acadian community of Little River Harbour in Yarmouth County, Nova Scotia. It was originally referred to as the Little River Duck Dog, which later became its current name. They are often referred to simply as Tollers.

While Tollers are often mistaken for small golden retrievers, they are more physically and mentally active than goldens and are slightly different in build. Tollers are medium to heavy boned, muscular, compact, athletic, powerful, and balanced. Their heads are clear cut and slightly wedge shaped, and their ears sit high on their head and are triangular with a rounded tip. The underside of their tail, legs, and body have some feathering. Typical coat colors range from golden red to crimson.

Tollers were bred as hunting dogs that lure waterfowl within hunting range of their owners and then spook the birds to make them fly so they can shoot them. After the bird is shot, the dog is

sent to retrieve it, which often involves retrieving the bird from icy waters. Fortunately, Tollers have a water-repellent double coat of fur to help them remain warm while doing so. Tollers love to hunt, swim, hike, and play and are happiest when working and using up their boundless energy. It is said that their expressions turn sad when they don't have a task to work on. Tollers are also highly intelligent, alert, affectionate, and have a desire to please their owners. They are good family dogs that get along well with children and other dogs.

While Tollers are not at risk of being endangered, many of the animals on O'Reilly covers are; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from a loose plate, source unknown. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.