Assistive Technology testing

- Screen reader testing basics
  - Browser compatibility
  - Screen reader commands
    - Key commands for desktop screen readers
    - Key commands for mobile screen readers
- Screen reader testing demo
  - Step 1
  - Step 2
    - Issue 1: Content structure {: #content-structure}
- Issue 2: Link context {: #link-context}
- Issue 3: Decorative image
  - Issue 4: Bullet decoration {: #bullet-decoration}
  - Issue 5: Form field
- Wrap up

{% Aside %} This module is a continuation of the previous two testing modules, automated accessibility testing and manual accessibility testing. If you have not gone through the exercises in those modules yet, we encourage you to do so, as this module starts where they left off. {% endAside %}

This module focuses on using assistive technology (AT) for accessibility testing. A person with disabilities can use AT to help increase, maintain, or improve the capabilities of performing a task.

In the digital space, ATs can be:

- No/Low-tech: head/mouth sticks, hand-held magnifiers, devices with large buttons
- High-tech: voice-activated devices, eye-tracking devices, adaptive keyboards/mice
- Hardware: switch buttons, ergonomic keyboards, auto-refreshing Braille device
- Software: text-to-speech programs, live captions, screen readers

We encourage you to use multiple types of ATs in your overall testing workflow.

Screen reader testing basics

In this module, we focus on one of the most popular digital ATs, screen readers. A screen reader is a piece of software that reads the underlying code of a website or app. It then converts that information into speech or Braille output for the user.

Screen readers are essential for people who are blind and deafblind, but they also could benefit people with low vision, reading disorders, or cognitive disabilities.

Browser compatibility

There are multiple screen reader options available. The most popular screen readers today are JAWS, NVDA, and VoiceOver for desktop computers and VoiceOver and Talkback for mobile devices.

Depending on your operating system (OS), favorite browser, and the device that you use, one screen reader may stand out as the best option. Most screen readers are built with specific hardware and web browsers in mind. When you use a screen reader with a browser it was not calibrated for, you may encounter more "bugs" or unexpected behavior. Screen readers work best when used in the following combinations.

Screen reader	OS	Browser compatibility
Job Access With Speech (JAWS)	Windows	Chrome, Firefox, Edge
Non-Visual Desktop Access (NVDA)	Windows	Chrome and Firefox
Narrator	Windows	Edge
VoiceOver	macOS	Safari
Orca	Linux	Firefox
TalkBack	Android	Chrome and Firefox
VoiceOver (for mobile)	iOS	Safari
ChromeVox	ChromeOS	Chrome

Screen reader commands

Once you have the proper set-up for your screen reader software for your desktop or mobile device, you should look at the screen reader documentation (linked in the preceding table) and run through some

essential screen reader commands to familiarize yourself with the technology. If you have used a screen reader before, consider trying out a new one!

When using a screen reader for accessibility testing, your goal is to detect problems in your code that interfere with the usage of your website or app, not to emulate the experience of a screen reader user. As such, there is a lot you can do with some foundational knowledge, a few screen reader commands, and a bit—or a lot—of practice.

If you need to further understand the user experience of people using screen readers and other ATs, you can engage with many organizations and individuals to gain this valuable insight. Remember that using an AT to test code against a set of rules and asking users about their experience often yields different results. Both are important aspects to create fully inclusive products.

Key commands for desktop screen readers

Element	NVDA (Windows)	VoiceOver (macOS)
Command	Insert (NVDA key)	Control + Option (VO key)
Stop audio	Control	Control
Read next/prev	↓ or ↑	VO + → or ←
Start reading	NDVA + ↓	VO + A
Element List/Rotor	NVDA + F7	VO + U
Landmarks	D	VO + U
Headings	H	VO + Command + H
Links	K	VO + Command + L
Form controls	F	VO + Command + J
Tables	T	VO + Command + T
Within Tables	NDVA + Alt + ↓ ↑ ← →	VO + ↓ ↑ ← →

Key commands for mobile screen readers

Element	TalkBack (Android)	VoiceOver (iOS)
Explore	Drag one finger around the screen	Drag one finger around the screen
Select or activate	Double tap	Double tap
Move up/down	Swipe up or down with two fingers	Swipe up or down with three fingers
Change pages	Swipe left or right with two fingers	Swipe left/right with three fingers
Next/previous	Swipe left/right with one finger	Swipe left/right with one finger

Screen reader testing demo

To test our demo, we used a Safari on a laptop running MacOS and capture sound. You can walk through these steps using any screen reader, but the way you encounter some errors may be different from how its described in this module.

Step 1

Visit the updated CodePen, which has all the automated and manual accessibility updates applied.

View it in debug mode to proceed with the next tests. This is important, as it removes the `<iframe>` which surrounds the demo webpage, which may interfere with some testing tools. Learn more about CodePen's debug mode.

Step 2

Activate the screen reader of your choice and go to the demo page. You may consider navigating through the entire page from top to bottom before focusing on specific issues.

We've recorded the our screen reader for each issue, before and after the fixes are applied to the demo. We encourage you to run through the demo with your own screen reader.

Issue 1: Content structure {: #content-structure}

Headings and landmarks are one of the primary ways people navigate using screen readers. If these are not present, a screen reader user has to read the entire page to understand the context. This can take a lot of time and cause frustration. If you try to navigate by either element in the demo, you will quickly discover that they do not exist.

- Landmark example: `<div class="main">...</div>`
- Heading example: `<p class="h1">Join the Club</p>`

If you have updated everything correctly, there should not be any visual changes, but your screen reader experience will have dramatically improved.

> {% YouTube "o8gWVi97cMg" %}
> Listen to the screen reader navigate through this issue.
>
> {% Img src="image/VbsHyyQopiec0718rMq2kTE1hke2/dNzbda0Lx1XUeCadVLMH.svg", alt="", width="28", height="28"%}

Let's fix it.

Some inaccessible elements can't be observed by just looking at the site. You may remember the importance of heading levels and semantic HTML from the Content structure module. A piece of content may look like a heading, but the content is actually wrapped in a stylized `<div>`.

To fix the issue with headings and landmarks, you must first identify each element that should be marked up as such and update the related HTML. Be sure to update the related CSS as well.

Landmark example: `<main>...</main>`

Heading example: `<h1>Join the Club</h1>`

If you have updated everything correctly, there should not be any visual changes, but your screen reader experience will have dramatically improved.

> {% YouTube "FfM3qvEWHjk" %}
> Now that we've fixed the content structure, listen to the screen reader navigate through the demo again.

Issue 2: Link context {: #link-context}

It's important to give content to screen reader users about the purpose of a link and if the link is redirecting them to a new location outside of the website or app.

In our demo, we fixed most of the links when we updated the active image alternative text, but there are a few additional links about the various rare diseases that could benefit from additional context—especially since they redirect to a new location.

```
<a href="https://rarediseases.org/rare-diseases/maple-syrup-urine-disease">
  Maple syrup urine disease (MSUD)
</a>
```

> {% YouTube "kk7LNdtfYMM" %}
> Listen to the screen reader navigate through this issue.

> {% Img src="image/VbsHyyQopiec0718rMq2kTE1hke2/dNzbda0Lx1XUeCadVLMH.svg", alt="", width="28", height="28"%}

Let's fix it.

To fix this issue for screen reader users, we update the code to add more information, without affecting the visuals element. Or, to help even more people such as those with reading and

cognitive disorders, we may choose to add additional visual text instead.

There are many different patterns we may consider to add additional link information. Based on our simple environment that supports just one language, an ARIA label is a straightforward option in this situation. You may notice that the ARIA label overrides the original link text, so make sure to include that information in your update.

```
<a href="https://rarediseases.org/rare-diseases/maple-syrup-urine-disease"
  aria-label="Learn more about Maple syrup urine disease on the Rare Diseases website.">
  Maple syrup urine disease (MSUD)
</a>
```

{% YouTube "Ezr7cMdCQlE" %}
Now that we've fixed the link context, listen to the screen reader navigate through the demo again.

Issue 3: Decorative image

In our automated testing module, Lighthouse was unable to pick up on the inline SVG that acts as the main splash image on our demo page—but the screen reader finds it and announces it as "image" without additional information. This is true, even without explicitly adding the `role="img"` attribute to the SVG.

```
<div class="section-right">
  <svg>...</svg>
</div>
```

{% YouTube "TKHHTGghrHs" %}
Listen to the screen reader navigate through this issue.

{% Img src="image/VbsHyyQopiec0718rMq2kTE1hke2/dNzbda0Lx1XUeCadVLMH.svg", alt="", width="28", height="28"%}

Let's fix it.

To fix this issue, we first need to decide if the image is informative or decorative. Based on that decision, we need to add the appropriate image alternative text (informative image) or hide the image from screen reader users (decorative).

We weighed the pros and cons of how best to categorize the image and decided it was decorative, which means we want to add or modify the code to hide the image. A quick method is to add a `role="presentation"` to the SVG image directly. This sends a signal to the screen reader to skip over this image and not list it in the images group.

```
<div class="section-right">
  <svg role="presentation">...</svg>
</div>
```

{% YouTube "KqTf8Pl2lMU" %}
Now that we've fixed the decorative image, listen to the screen reader navigate through the demo.

Issue 4: Bullet decoration {: #bullet-decoration}

You may have noticed that the screen reader reads the CSS bullet image under the rare diseases sections. While not the traditional type of image we discussed in the Images module, the image still must be modified as it disrupts the flow of the content and could distract or confuse a screen reader user.

```
<p class="bullet">...</p>
```

{% YouTube "sDR2w-HGHOo" %}
Listen to the screen reader navigate through this issue.

{% Img src="image/VbsHyyQopiec0718rMq2kTE1hke2/dNzbda0Lx1XUeCadVLMH.svg", alt="", width="28", height="28"%}

Let's fix it.

Much like the decorative image example discussed earlier, you can add a `role="presentation"` to the HTML with the bullet class to hide it from the screen reader. Similarly, a `role="none"` would work. Just be sure not to use `aria-hidden: true` or you will hide all of the paragraph information from screen reader users.

```
<p class="bullet" role="none">...</p>
```

Issue 5: Form field

In the Forms module, we learned that all form fields must also have a visual and programmatic label. This label must remain visible at all times.

In our demo, we're missing both a visual and programmatic label on our newsletter sign-up email field. There is a text placeholder element, but this does not replace the label as it's not visually persistent and is not fully compatible with all screen readers.

```
<form>
  <div class="form-group">
    <input type="email" placeholder="Enter your e-mail address" required>
    <button type="submit">Subscribe</button>
  </div>
</form>
```

{% YouTube "7hncAhi4UUk" %}
Listen to the screen reader navigate through this issue.

{% Img src="image/VbsHyyQopiec0718rMq2kTE1hke2/dNzbda0Lx1XUeCadVLMH.svg", alt="", width="28", height="28"%}

Let's fix it.

To fix this issue, replace the text placeholder with a look-alike label element. That label element is programmatically connected to the form field and movement was added with JavaScript to keep the label visible even when content is added to the field.

```
<form>
  <div class="form-group">
    <input type="email" required id="youremail" name="youremail" type="text">
    <label for="youremail">Enter your e-mail address</label>
    <button type="submit" aria-label="Subscribe to our newsletter">Subscribe</button>
  </div>
</form>
```

{% YouTube "hNbDfcmdi_A" %}
Now that we've fixed the form, listen to the screen reader navigate through the demo.

Wrap up

Congratulations! You have completed all of the testing for this demo. You can look at all of these changes in the updated Codepen for this demo.

Now, you can use what you've learned to review the accessibility of your own websites and apps.

The goal of all of this accessibility testing is to address as many possible issues that a user may potentially encounter. However, this does not mean that your website or app will be perfectly accessible when you're finished. You'll find the most success by designing your website or app with accessibility throughout the process, and incorporating these tests with your other pre-launch testing.

{% Assessment 'at' %}

□□□ #2
□ □□□ □□□ 2026-10-31 22:44:41 UTC
□ □□□ □□□ 2026-10-31 22:44:43 UTC