Accessible Accordions

Using data definitions, Velocity, and a little JS to get there

Hey everyone. Welcome to Accessible Accordions using data definitions, velocity, and a little JS to get there

About your presenter(s)

- Winston Churchill-Joell, Director of Digital Services at Sarah Lawrence College since 2001
- Cascade Server for nearly 15 years
- Responsible for all things Cascade
- Passionate about digital accessibility, semantic markup, CSS, vanilla JS, and dogs
- Leo is the cute one with the slightly bigger ears; he's passionate about anything that squeaks

I'm Winston Churchill-Joell, Director of Digital Services in Marketing & Communications at Sarah Lawrence College. I'm responsible for Cascade Server administration, development and user support. We've been in Cascade for nearly 15 years. I love working on the web and I care about making experiences that are accessible to everyone. The cute guy perched on my shoulder there is Leo. He made me promise not to go into the weeds on this talk. You can see he's all ears, but if I start droning on he may start barking to make this talk more interesting.

Content warning: this presentation uses small gif animations on most slides as illustrations. These can be distracting or triggering for folks who struggle with motion on screen.

Project Files and Resources

- Github repo with all HTML, CSS, and JS files
- Download Cascade Server site archive
- Reference materials for this talk
- Authors and content creators I follow

slc.edu/csuc2023

We'll cover a lot of ground in this talk so we'll be moving fast. The QR code and URL here link through to a Github repo with all the files used for the topics we'll cover. This includes the Cascade Server site archive that has the data definitions and velocity formats used to build the demo pages in this talk. The read me also has a few links to original sources for this talk along with links to some of the authors and other resources I frequently follow and read.

The Accessibility Audit

- Partnered with an accessibility firm to audit our public website for WCAG 2.1 AA compliance
- A chance to work with web accessibility domain experts
- User testing by people with disabilities who rely on assistive technologies
- An opportunity to learn best practices
- Humbling experience that taught me a lot about web development

We knew we wanted to do a better job of making our site accessible, but we didn't know where to begin. We decided to partner with a firm and get an honest assessment of where our current site stood regard to the Web Content Accessibility Guidelines, or WCAG for short. These guidelines are not a legal threshold, but they are often used as the standard to determine accessibility compliance. That assessment would eventually become a scope of work and a roadmap for our remediation efforts. I'm grateful we had the opportunity to not only work with domain experts but with people with disabilities who rely on assistive technologies every day. I learned a lot about best practices and frankly, I developed a whole new appreciation for the fundamentals of web development.

Accessibility Remediation: A Chance to do Better

The Problem

- **Non-semantic controls** and brittle interaction build out
- **Poor color contrast** and missing visual cues that excluded some users
- **Absent landmarks** and confusing heading hierarchy for screen reader users
- **Over-dependence on libraries** and frameworks for simple problems

The Opportunity

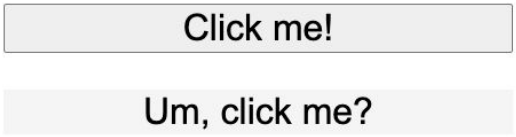
- **Model interactive components** from the ground up after best practices
- **Adjust our color palette** for better contrast and restore interaction styles for clarity
- **Use semantic regions** and set up content management for better heading control
- **Lean into modern standards**, vanilla JavaScript and CSS

The audit findings were a lot to digest, but it was liberating to have a concrete list of things to fix.

I'm listing a handful themes here that came up often in our remediation work. Tackling these issues ranged from attainable, low-hanging fruit to more substantial rewrites. It's been a rewarding process that created opportunities to improve our practices in Cascade Server through better data definition design and in our front-end code base by taking advantage of modern standards and features.

Accessibility is a *Requirement*, Not a Feature

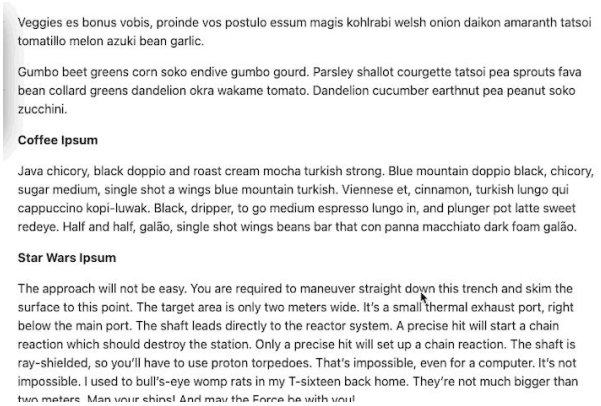
- Accessibility also happens to be the default in most cases
- The requirement is that we don't break stuff that works out of the box
- Use progressive enhancement on top of functioning default behavior (which may be no behavior at all)
- Override defaults with care and intention; the result should be something equivalent or better



You may have heard this saying, "accessibility is a requirement, not a feature." I couldn't agree more, but I would add that accessibility is also the default in most cases and the requirement is that we don't mess it up. Interactive components like accordions are a gray area where we need to step in and add functionality, but as we'll see when we dig in here, that functionality should be a layer on top of something that works by default. In the case of content, the default should be… content. Ok, let's dig in.

Collapsible Sections: What they are

- **Semantic headings** that control the visibility of their content sections
- **Content by default**: no JS? No problem!
- **Keyboard accessible**: controls can be reached by tab key and activated by space and enter keys
- **Screen reader accessible**: semantic content, landmarks, and controls reveal structure and announce interaction state
- **Visually accessible**: unambiguous controls clearly show focus and interaction state



I chose to talk about the accordion component, because it touches upon most of the key issues that came up during our remediation work. We have interaction, behavior state, semantic controls, landmarks, and heading hierarchy. Getting this thing right was the key to solving a number of related accessibility issues.

Accordions, or collapsible sections are essentially headings that can control the visibility of the content they are related to. One way to think of them is that accordions are an affordance for sighted users that actually offers an equivalent experience to what screen reader users enjoy. That's not something you hear very often, and yes you heard that correctly. Screen readers can easily navigate the headings of a properly marked up page and accordions create a similar experience for sighted users.

Collapsible Sections: What they are

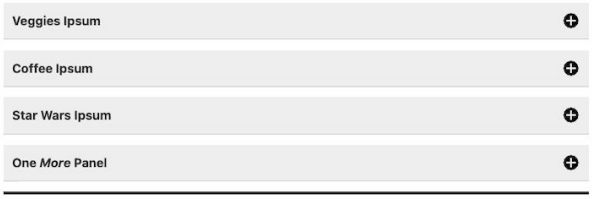
- **Semantic headings** that control the visibility of their content sections
- **Content by default**: no JS? No problem!
- **Keyboard accessible**: controls can be reached by tab key and activated by space and enter keys
- **Screen reader accessible**: semantic content, landmarks, and controls reveal structure and announce interaction state
- **Visually accessible**: unambiguous controls clearly show focus and interaction state

Take away the interaction and accordions should be content by default. Network connections can fail, end users can have external CSS or JS disabled for any reason. Your content should be available no matter what.

In the illustration I've commented out the script tag to simulate JavaScript disabled or failed to load. The content remains in the state it was published from Cascade Server. Completely available and usable.

Collapsible Sections: What they are

- **Semantic headings** that control the visibility of their content sections
- **Content by default**: no JS? No problem!
- **Keyboard accessible**: controls can be reached by tab key and activated by space and enter keys
- **Screen reader accessible**: semantic content, landmarks, and controls reveal structure and announce interaction state
- **Visually accessible**: unambiguous controls clearly show focus and interaction state

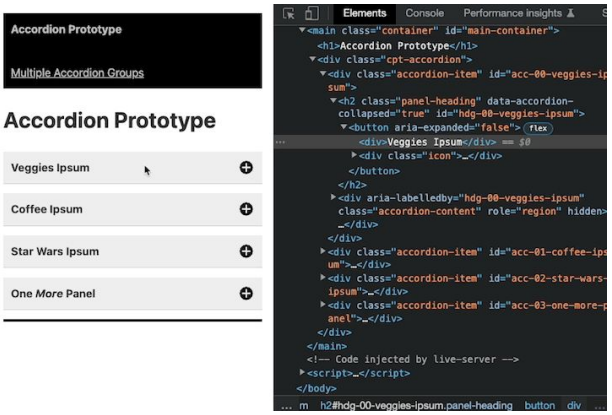


Keyboard access is a convenience for many users and it's a must for users with fine motor control disabilities who struggle to using a pointing device.

In this illustration, I'm tabbing through the button headings and hitting the space key to expand and collapse the content. No pointing device required.

Collapsible Sections: What they are

- **Semantic headings** that control the visibility of their content sections
- **Content by default**: no JS? No problem!
- **Keyboard accessible**: controls can be reached by tab key and activated by space and enter keys
- **Screen reader accessible**: semantic content, landmarks, and controls reveal relationships and announce state
- **Visually accessible**: unambiguous controls clearly show focus and interaction state

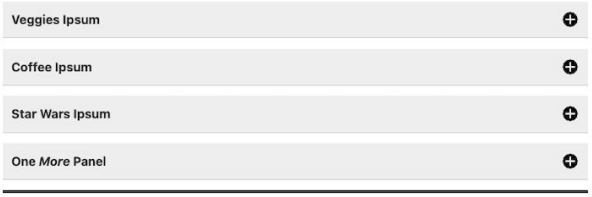


Screen readers rely on semantic markup and a layer of web standards called Accessible Rich Internet Applications, or ARIA for short. Used sparingly and I mean sparingly—no ARIA is better than bad ARIA. But used carefully, these features are a necessary bridge to help people with disabilities use interactive components such as accordions.

The developer tools window in the illustration shows updates to the button's aria-expanded attribute, and the hidden property on the accordion content panel. The unique ID of the heading is referenced by the aria-labelledby attribute on the content panel which is designated a region landmark by the role attribute. You might be thinking, if there's a time for his dog to start barking, this has to be it. Bear with me. I know this seems like a lot to take in but don't worry, this is why we're setting up a data definition and Velocity format to do the heavy lifting.

Collapsible Sections: What they are

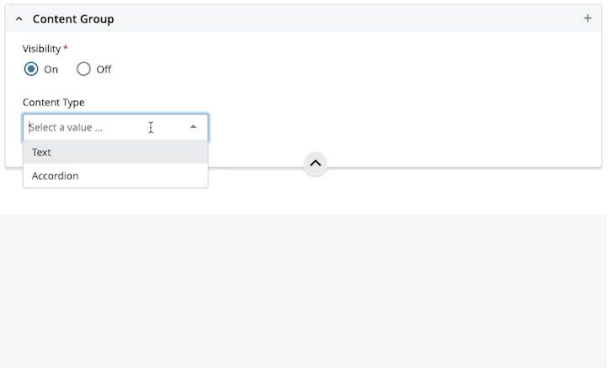
- **Semantic headings** that control the visibility of their content sections
- **Content by default**: no JS? No problem!
- **Keyboard accessible**: controls can be reached by tab key and activated by space and enter keys
- **Screen reader accessible**: semantic content, landmarks, and controls reveal structure and announce interaction state
- **Visually accessible**: unambiguous controls clearly show focus and interaction state



Accessibility isn't just for blind users. Users come to the web with a range of cognitive and physical abilities and visual comprehension. Our job is to make interaction experiences that are intuitive and easy to understand to everyone. This example is probably a bit extreme for most brand guidelines, but the key is to make things obvious and use adequate text size and color contrast.

Collapsible Sections: Data Definition

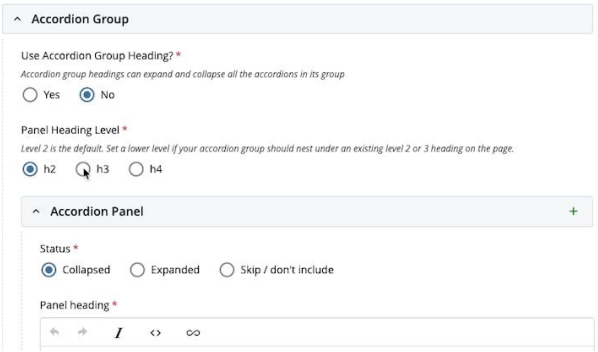
- **Consistency**: structured data to provide an easy, repeatable way to create content
- **Heading hierarchy**: settings to easily manage heading levels for accessibility
- **Rich formatted headings**: a streamlined inline-only WYSIWYG
- **Keeping it simple with schema**: limit the options for panel content and avoid inaccessible content scenarios
- **Use shared fields** for portability



Because accordions have specific requirements around attributes for ARIA, classes, and data attributes for JavaScript, accordions need to be consistently marked up. Obviously this is where a CMS can shine. Cascade's data definitions with branch logic, shared fields, and granular WYSIWYG controls take it much further to create a good experience for content managers and remove the burden of creating a complex component such as this.

Collapsible Sections: Data Definition

- **Consistency**: structured data to provide an easy, repeatable way to create content
- **Heading hierarchy**: settings to easily manage heading levels for accessibility
- **Rich formatted headings**: a streamlined inline-only WYSIWYG
- **Keeping it simple with schema**: limit the options for panel content and avoid inaccessible content scenarios
- **Use shared fields** for portability



Since screen readers can navigate by headings, managing heading level hierarchy is very important to avoid jarring or confusing moments for people with disabilities. A bonus here is that good heading management is also key to good SEO, so it's a win win. In this example, we're giving content managers explicit control over heading levels and explaining why it matters right in the data definition.

Radio buttons that control the visibility of required fields create a better experience for Cascade developers with controlled vocabularies for settings and fewer scenarios where missing required content has to be handled with conditional logic

In the illustration, and we'll see this later in the demo, I have an option for setting the panel heading level for h2 through h4, with h2 being the default. If I choose yes to enable the Group Heading option, the data definition swaps out the required panel heading level radio buttons for a required Group Heading. For my purposes, I've set a requirement that the group heading is always a level 2. By turning this on, we *do* need the heading text, but we don't need manual panel heading levels. Those are now required to be level 3 to follow the group heading. Planning for these scenarios and guiding the content manager with the data definition can spare content managers and developers alike, a lot of grief down the road

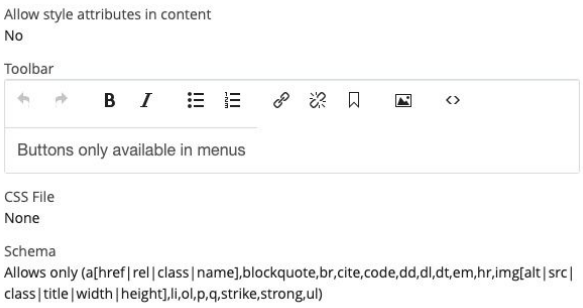
Collapsible Sections: Data Definition

- **Consistency**: structured data to provide an easy, repeatable way to create content
- **Heading hierarchy**: settings to easily manage heading levels for accessibility
- **Rich formatted headings**: a streamlined inline-only WYSIWYG
- **Keeping it simple with schema**: limit the options for panel content and avoid inaccessible content scenarios
- **Use shared fields** for portability

Speaking of required headings, Cascade's WYSIWYG definitions are an excellent alternative to plain text fields for headings. You can enable a limited set of inline elements and strip out everything else that might end up in the markup from a copy/past operation. Since we're designating the heading level in Velocity, we want to make sure no one introduces their own headings, or tables, or cat videos. Basically anything that might break the functionality of the heading as a control button.

Collapsible Sections: Data Definition

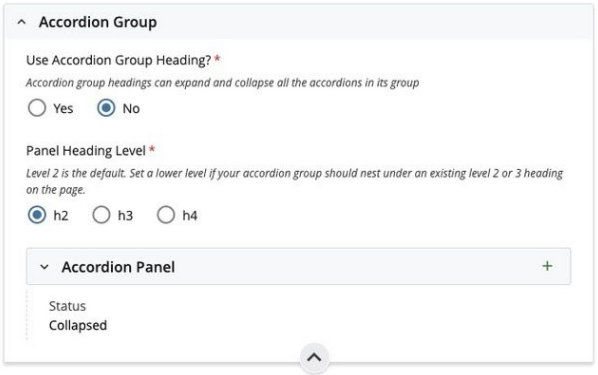
- **Consistency**: structured data to provide an easy, repeatable way to create content
- **Heading hierarchy**: settings to easily manage heading levels for accessibility
- **Rich formatted headings**: a streamlined inline-only WYSIWYG
- **Keeping it simple with schema**: limit the options for panel content and avoid inaccessible content scenarios
- **Use shared fields** for portability



Similarly I've set up a schema that limits the content that can come through for panel content. If you're tired of spans and divs coming along for the ride in copy/paste, schema is your friend. In this example we're allowing a generous list of elements, but some notable exceptions again are headings 1-6. If you decide to use Schema, you have to explicitly allow not only the tags you want, but any allowed attributes those tags should have. For example you could add a link or an image, but if you forget to allow the href or source attributes, they'll get stripped out by the schema. It's a powerful tool, but it needs some testing to make sure you've got it set up properly for your needs

Collapsible Sections: Data Definition

- **Consistency**: structured data to provide an easy, repeatable way to create content
- **Heading hierarchy**: settings to easily manage heading levels for accessibility
- **Rich formatted headings**: a streamlined inline-only WYSIWYG
- **Keeping it simple with schema**: limit the options for panel content and avoid inaccessible content scenarios
- **Use shared fields** for portability



Now that you've got this awesome component defined, you might realize you have a lot of places you need to add it before you can use it. Let's say you want to use the accordion pattern in a block or in several data definitions. Externalizing it to a shared field makes it easier for future enhancements and fixes to make their way to all instances of that pattern.

Collapsible Sections: Velocity

- **String methods** to make easy work of generating the required ARIA attributes and element IDs to set up accordions
- **Using `$foreach.index`** and `$foreach.parent` to ensure unique IDs
- **Helper macros** as utilities to handle complex string operations
- **Render semantic output** and nothing more; use JS to add interactive elements and icons

```
#end
##
#macro ( stringToID $string )
${_EscapeTool.xml($_DisplayTool.stripTags($string)).toLowerCase
    ().replaceAll(' ?(?:&(?:#\d+|\w+);|[^A-Za-z0-9]) ?', ' ').trim
    ().replaceAll(' ','-').replaceAll('^\d+','')}##
#end
##
```

Now that we have have our data definition set up, it's time to transform that structure into our final output. Remember from our earlier illustrations that there are IDs and ARIA references needed to make accordions screen reader accessible. While this can be done with XSLT, Velocity's Java string methods and regular expression support make this job much easier. We'll talk more in a minute about macros and how they can help tame string operations like the one illustrated here from overwhelming your formats.

Collapsible Sections: Velocity

- **String methods** to make easy work of generating the required ARIA attributes and element IDs to set up accordions
- **Using `$foreach.index`** and `$foreach.parent` to ensure unique IDs
- **Helper macros** as utilities to handle complex string operations
- **Render semantic output** and nothing more; use JS to add interactive elements and icons

```
('heading').textValue)">
<${panelHeadingLevel} class="panel-heading" data
    -accordion-collapsed="${item.getChild('status'
    ).textValue}" id="hdg-${foreach.parent
    .index}${foreach.index}-#stringToID($item.getChild
    ('heading').textValue)"><span>$item.getChild
    ('heading').textValue</span></${panelHeadingLevel}>
<div aria-labelledby="hdg-${foreach.parent
    .index}${foreach.index}-#stringToID($item.getChild
    ('heading').textValue)" class="accordion-content"
    role="region">
    $item.getChild('panel-body').textValue
</div>
```

You may have a scenario where the same heading is reused in several accordion groups on the same page. For example you could outline the same steps of a process on the same page for several different audiences. Since we're using the heading text to generate a unique ID, this presents a problem. IDs must be unique to avoid confusion for screen readers and later on we'll run into problems with deep linking to an ID if it's not unique.

Velocity's foreach object can come in handy here. It gives you access to the iteration count and index not only of the loop you're currently in but any parent loops you're iterating through as well. Combining these indices makes it possible to generate unique IDs when the headings you're processing are repeated for whatever reason.

Collapsible Sections: Velocity

- **String methods** to make easy work of generating the required ARIA attributes and element IDs to set up accordions
- **Using `$foreach.index`** and `$foreach.parent` to ensure unique IDs
- **Helper macros** as utilities to handle complex string operations
- **Render semantic output** and nothing more; use JS to add interactive elements and icons

```
ndex}-#stringToID($item.getChild('heading').textValue)">
{panelHeadingLevel} class="panel-heading" data-accordion
  -collapsed="${item.getChild('status').textValue}" id="hdg
  -${foreach.parent.index}${foreach.index}-#stringToID($item
  .getChild('heading').textValue)"><span>$item.getChild('heading
  ).textValue</span></${panelHeadingLevel}>
iv aria-labelledby="hdg-${foreach.parent.index}${foreach.index}
  -#stringToID($item.getChild('heading').textValue)" class
  ="accordion-content" role="region">

nd

acro ( stringToID $string )
_EscapeTool.xml($_DisplayTool.stripTags($string)).toLowerCase
  ().replaceAll(' ?(?:&(?:#\d+|\w+);|[^A-Za-z0-9]) ?', ' ').trim
  ().replaceAll(' ','-').replaceAll('^\d+','')}##
nd
```

As we discussed, Velocity has some great options for string manipulation, but the code can get hard to ready very quickly. Cleaning up strings can involve several chained operations that can make your code hard to troubleshoot, especially inside of an attribute on a tag. Moving those operations to a clearly named macro can give you a handy utility and a single place to maintain for future development. In this example we're stripping out tags, XML escaping the result, then lower casing the entire string before removing special characters, replacing spaces with hyphens and removing any leading integers. In other words making a string viable as an ID.

Collapsible Sections: Velocity

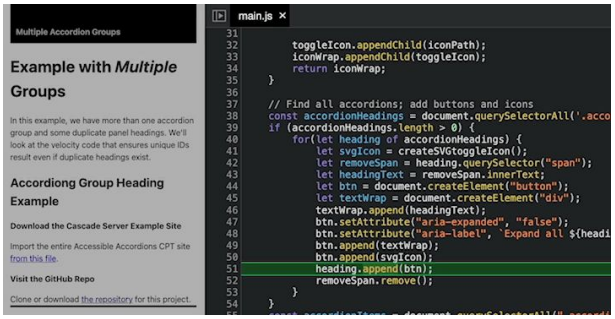
- **String methods** to make easy work of generating the required ARIA attributes and element IDs to set up accordions
- **Using `$foreach.index`** and `$foreach.parent` to ensure unique IDs
- **Helper macros** as utilities to handle complex string operations
- **Render semantic output** and nothing more; use JS to add interactive elements and icons

```
<div class="cpt-accordion">
    <div class="accordion-item" id="acc-00-veggies-ipsum">
        <h2 class="panel-heading" data-accordion-collapsed="true" id="hdg-00
            -veggies-ipsum">
            <span>Veggies Ipsum</span>
        </h2>
        <div aria-labelledby="hdg-00-veggies-ipsum" class="accordion-content"
            role="region">
            <p>Veggies es bonus vobis, proinde vos postulo essum magis
                kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon
                azuki bean garlic.</p>
            <p>Gumbo beet greens corn soko endive gumbo gourd. Parsley shallot
                courgette tatsoi pea sprouts fava bean collard greens
                dandelion okra wakame tomato. Dandelion cucumber earthnut pea
                peanut soko zucchini.</p>
        </div>
    </div>
```

So here's our output. We're introducing a heading with a unique id followed by a div container for the panel content that is given a semantic role of region. We're telling screen readers that this region is labeled by the preceding heading. We're passing along some information about the state of the accordion in the heading. What's notably absent here is a button or an icon. Whether or not this ends up as an accordion, we've provided an accessible foundation to work with and excluded anything that might not be used or needed.

Collapsible Sections: Progressive Enhancement

- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)

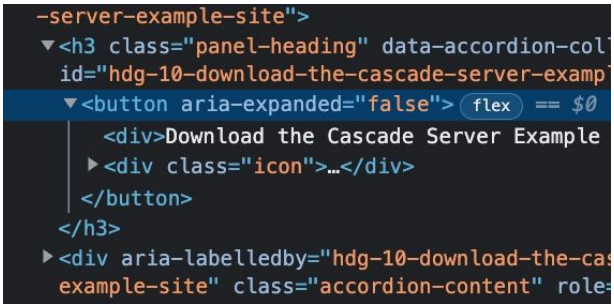


The idea of progressive enhancement is that features are added on top of a working foundation that can still function without those features. The ability to collapse sections of content by clicking on their headings is an enhancement. If for whatever reason that feature didn't work or wasn't supported, there's nothing broken about a bunch of headings and their contents fully visible on the page. A good analogy might be an escalator. If for some reason the escalator breaks down or there's no power, the escalator is still a flight of stairs.

A common misperception about JavaScript is that it is a barrier to accessibility. While you can definitely build inaccessible experiences with JavaScript, interactive components like accordions need JavaScript to be fully accessible as well as operable. This makes the case for using JavaScript to inject the interactive button and visual SVG elements that would otherwise be confusing and inoperable if JavaScript is unavailable for any reason.

Collapsible Sections: Progressive Enhancement

- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)



Now that we're ready to progressively enhance these headings as controls, it's time to add something to the heading that users can tab to, that registers a click event when the spacebar or enter key are pressed, and that screen readers automatically recognize as a button. The HTML button element is purpose built for this use case, it already has the semantics and the interactivity we need built in. And it's important that we don't replace the heading with a button. We need the role of heading and the role of button so we need both elements. We're using ARIA here to communicate the state of the button, which may seem counterintuitive since were collapsing the content, but the screen reader will announce state of the button when its encountered. We'll get a chance to hear this in action during the demo.

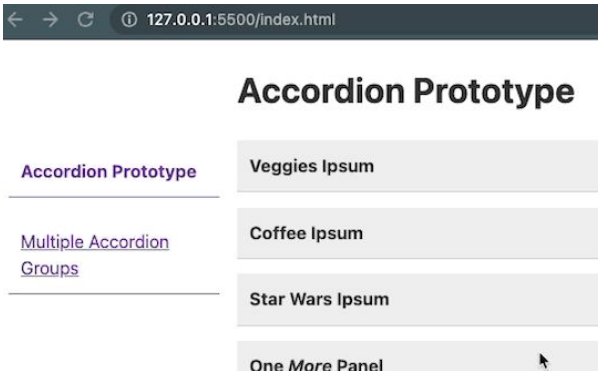
Collapsible Sections: Progressive Enhancement

- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)

I like using data attributes for storing state rather than classes. They're easier to work with in JavaScript and their intent is clearer. Same goes for properties like hidden. The bonus with hidden is that you don't need to even use CSS classes. When the time comes to talk about page search and printing, we'll talk about overriding the default behavior of hidden

Collapsible Sections: Progressive Enhancement

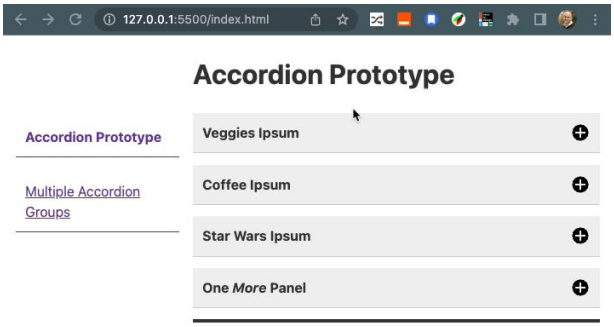
- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)



One of the many useful features of having unique IDs on each accordion heading is that they're accessible from the address bar as hashes at the end of the URL. The hash is an instruction to the browser to look for an element with that ID and scroll to that location. In our example we're using the history API to update the URL with the hash or ID of the heading that was last clicked. This makes it easy to copy the URL as a deep link down to the scroll position of the content and we'll use JavaScript to expand the accordion that matches the hash/ID from the URL

Collapsible Sections: Progressive Enhancement

- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)



Page search or command/control F is another common pattern for users to access the content of the page and I would argue it's fairly universal. Collapsed content needs to be searchable in the page, so we'll use JavaScript to listen for those key combinations and expand all the accordions on the page

Collapsible Sections: Progressive Enhancement

- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)

In the JavaScript for this project, there are looping operations to insert buttons and icons in all the accordion group headings and panel headings found in the DOM. While inserting each button It's tempting to add a click event handler right on the button. For small examples you can get away this, but it doesn't scale and you could end up with dozens or maybe even hundreds of event handlers sitting in memory that may never get used. The better option is to use event delegation. Add a click event handler to the document and use event.target.closest to isolate the button. This will capture clicks on the SVG icon or button text as well, which is critical since any of the button's descendants may capture the click event.

Collapsible Sections: Progressive Enhancement

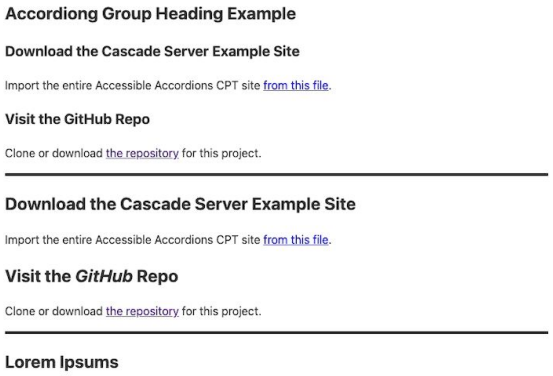
- **Add semantic controls** with JavaScript
- **Use buttons**, not DIVs; they ship with tab focus and keyboard interactivity for free
- **Use properties and attributes** instead of classes to keep track of state
- **Use the history API** to create URLs for deep linking to sections
- **Add support for search** on page (cmd/ctrl F)
- **Use event delegation** for performance
- **Execute first** in the call stack as soon as the DOM is *interactive* (`defer` or `readyState === 'interactive'`)

```
<head>
    <title>
        Accordion Prototype
    </title>
    <script src="js/main.js" defer></script>
    <link rel="stylesheet" href="css/main.css">
</head>
```

This project example is obviously a lot simpler than what you're likely to have set up in production. You'll have several external scripts loading from your own domain and potentially third-party scripts. Conventional wisdom is to avoid loading JavaScript before CSS to prevent blocking the page render, but this is one instance where you want the JS to update the DOM as soon as it is available or interactive. The defer property on the script tag is equivalent to a readyState of interactive on the readystatechange event listener. You want to wait until the DOM is loaded, but you want to modify the DOM before layout begins to render. Depending on your setup, you may opt to inline your JS on the page and wrap it in the readystatechange event listener to avoid a flash of expanded content that will cause layout shift.

Collapsible Sections: Minimal CSS

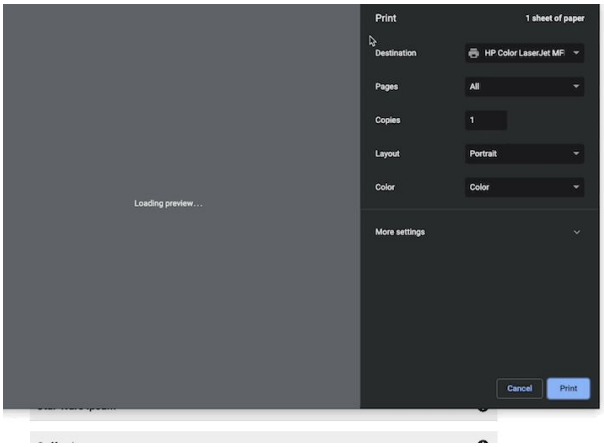
- **Preserve intuitive heading appearance** when JS is not present
- **Normalize accordion and panel heading** appearance regardless of level
- **Add print support** with media queries
- **Focus rings** can be *beautiful* and they are *essential*
  - **Bonus**: let the user choose the color
- **Use `:hover` and `background-color`** for clear visual cues
- **Use `:target` and `scroll-behavior: smooth`** (responsibly) for deep links



Your CSS should accommodate a number of scenarios and user preferences. We use JS to update the class list of the HTML element and give us a class of es6 or JS to manage layout of the component whether JS is present or not. If JS isn't present, we let the heading appearance reflect the heading level. If JS is present we want to normalize the appearance of the panel heading, regardless of what semantic heading level is used.

Collapsible Sections: Minimal CSS

- **Preserve intuitive heading appearance** when JS is not present
- **Normalize accordion and panel heading** appearance regardless of level
- **Add print support** with media queries
- **Focus rings** can be *beautiful* and they are *essential*
  - **Bonus**: let the user choose the color
- **Use `:hover` and `background-color`** for clear visual cues
- **Use `:target` and `scroll-behavior: smooth`** (responsibly) for deep links



Similar to search on the page, if someone prints your page, it's likely they want to print the content of the page. While we're at it, we can exclude interactive cues like background colors and SVG icons that either won't print any way or add confusing clutter.

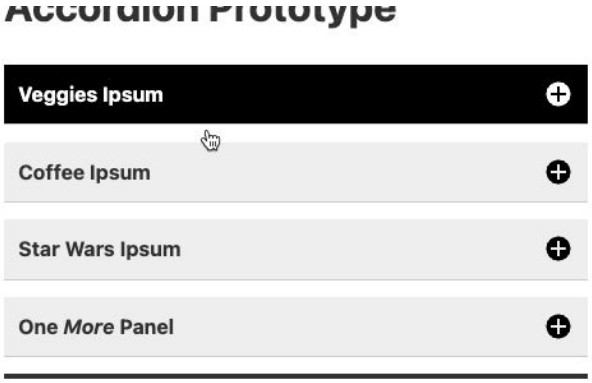
Collapsible Sections: Minimal CSS

- **Preserve intuitive heading appearance** when JS is not present
- **Normalize accordion and panel heading** appearance regardless of level
- **Add print support** with media queries
- **Focus rings** can be *beautiful* and they are *essential*
  - **Bonus**: let the user choose the color
- **Use `:hover` and `background-color`** for clear visual cues
- **Use `:target` and `scroll-behavior: smooth`** (responsibly) for deep links

Focus rings are an accessibility requirement, not an annoyance to be removed. Use the outline property instead of border to avoid layout shift and use offset to make the focus ring more appealing and distinct. I like to use AccentColor and the webkit alternative so that the user's system preferences are applied here for consistency. If you don't like how the focus ring looks on rounded corners or irregular shapes, try the CSS drop-shadow() function as an alternative. The important thing is to help your keyboard users clearly see where focus is on the page.

Collapsible Sections: Minimal CSS

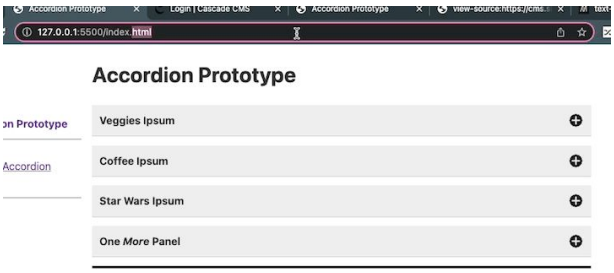
- **Preserve intuitive heading appearance** when JS is not present
- **Normalize accordion and panel heading** appearance regardless of level
- **Add print support** with media queries
- **Focus rings** can be *beautiful* and they are *essential*
  - **Bonus**: let the user choose the color
- **Use `:hover` and `background-color`** for clear visual cues
- **Use `:target` and `scroll-behavior: smooth`** (responsibly) for deep links



I know this is obvious, but if something is interactive, it's important that there are clear visual cues for sighted users as well. Buttons should have a button like appearance and respond to hover actions.

Collapsible Sections: Minimal CSS

- **Preserve intuitive heading appearance** when JS is not present
- **Normalize accordion and panel heading** appearance regardless of level
- **Add print support** with media queries
- **Focus rings** can be *beautiful* and they are *essential*
  - **Bonus**: let the user choose the color
- **Use `:hover` and `background-color`** for clear visual cues
- **Use `:target` and `scroll-behavior: smooth`** (responsibly) for deep links



Smooth scrolling is helpful for many users to understand what's going on and where they're navigating to. However, it's a lot of motion on screen and that can trigger debilitating effects for some users. The CSS file for this demo puts this feature and some other animation resets behind media queries for prefers-reduced-motion.

The :target pseudo class is a great option to add visual cues that help users understand where on the page they've navigated to.

Demo time…

Ok, let’s take a spin through the set up in Cascade Server to see these settings in action, and then we’ll spin up VoiceOver on macOS to hear how those settings positively impact the user experience for users who rely on a screen reader.

Thanks for listening! Once again, here's the QR code or the URL to take you to the repo for this talk. The readme also has links to download the Cascade Server archive file for the demo site and to the materials I researched for this talk. Many of these are from content creators and developers whom I follow and continue to learn from. I'm happy to take any questions!