Helsingin saavutettavuusmalli

Helsinki Model for Accessible Service Design

Sliders and carousels: Notes on technical implementation

Tero Pesonen

Contact: tpe at siteimprove.com

Version: 1st October 2021

Slider and Carousel Patterns

Definition

The slider is a component that collects into a single, scrollable element pieces of content that are of the same type: Images, video clips, linked product cards, and other sets whose number is considered too large to be spread out in a static grid layout.

A simple image slider can be found in the screen reader demo page at: http://siteimprove-accessibility.net/Demo/Page

Technically, a slider could be implemented as a tab list (and a tab list as a slider), but in practice the two components are used differently: Tabs fit best for a handful of categorically similar views that differ in the elements they contain, whereas sliders fit for items that may be any in number yet are structurally identical. In other words, tabs segment page contexts while sliders segment content within a page context.

A carousel is a slider that can be thumbed through in a loop, that is, it has no fixed end.

Motivation

Sliders may appear simple from usability point of view since they mainly function as content viewers. Yet, these elements impose a surprising number of accessibility requirements, as they build on non-standard controls that interact with each other in unusual ways that are, consequently, not always straightforward to optimize for non-visual or keyboard-only use.

As a matter of fact, the average slider as seen “in the wild” tends to be amongst the most challenging components for the typical user of assistive technology to interact with. Luckily, sliders are rarely crucial for using a service, but they do add value that one prefers were better available to all users.

This document strives to delineate at least some of the common accessibility requirements that carousels and sliders pose for developers and designers, as well as how to meet those requirements.

This is a longer document than the other implementation guides, as the element type id fundamentally more complex. At the same time, one acknowledges that front end developers today rarely enjoy the luxury of being able to build components like these from the scratch; one often must contend with a library solution given time limitations. When having to adapt an existing component, you make sure that at least the WCAG criteria are met, and then add on top of that what you can.

Accessibility requirements: Overview

There are at minimum three key accessibility requirements that a slider or carousel should meet:

1. The controls (buttons, thumbnails, clickable content panels, etc.) have an (i) accessible name, (ii) appropriate role, and (iii) state/value defined.

Although a standard requirement for all interactive elements, setting these attributes may be less obvious in the slider in which parts not only interact in non-standard manner but also comprise untypical subcomponents. What is, for instance, the semantical role of a clickable thumbnail list, and which sub roles and states does it host?

2. The keyboard focus order is according to a design; it is not random.

   If one builds the element into the DOM simply following its visual layout, one may end up with an unusual or even unusable order for the slider's controls when navigated by the keyboard using tabulator and arrow keys.

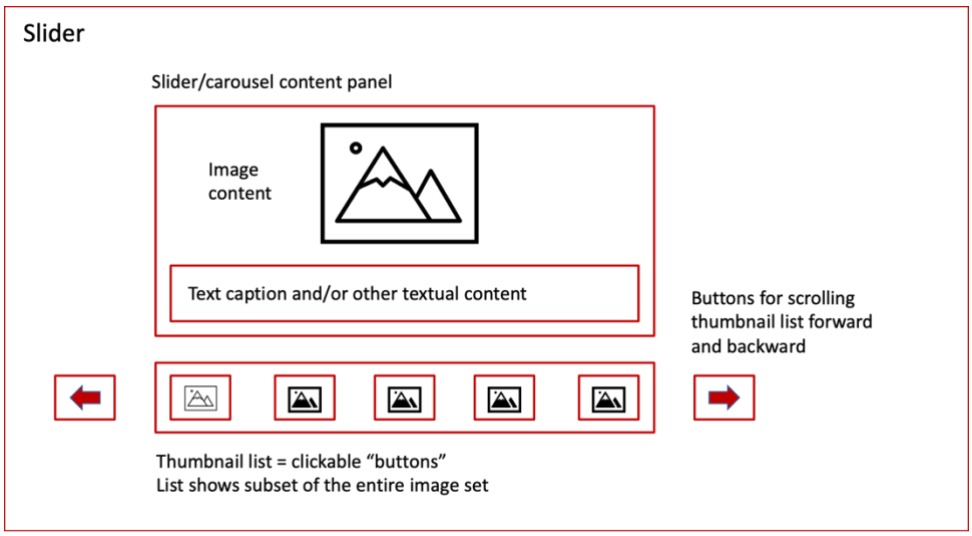
3. There is, additionally, a reading order that fosters assistive technology.

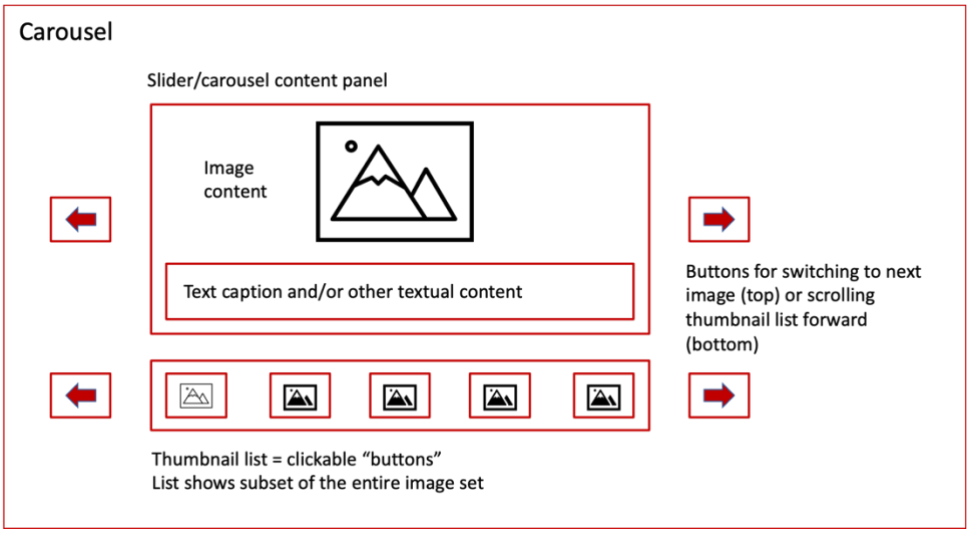
   The keyboard focus order only applies to interactive (i.e. clickable) elements, not all content that the slider contains. Assistive technology exposes also those non-interactive parts (images, texts, content wrappers and panels, etc.) that keyboard navigation skips.

   The order of these other elements therefore greatly affects how screenreader users perceive and comprehend the element. That is to say, the reading order may dictate whether the slider is easy or even feasible to use.

Requirement 1: Controls have a meaningful name, role, and value

The pictures below depict such parts as make up the demo page slider and carousel elements. Compared to the carousel, the slider lacks buttons for switching the image one at a time, and the thumbnail list will not loop. Most sliders do conform to a similar structure even as the panel content and control types vary.





Scroll / Switch buttons

Demo

The image switch buttons and thumbnail list scroll "arrows" are defined as <button> elements. They are not links since they do not transfer the user's position in the DOM. The buttons are also named with Aria-label, as they lack (visible) textContent names.

When the slider is scrolled to the beginning or end of the thumbnail list, the arrow at that end will become inactive (since there is nowhere left to scroll). The element is then set with tabindex="-1" and aria-disabled="true". The first will prevent the button from accepting keyboard focus, and the second will inform screenreader users that the

button has an inactive state. When the button again becomes active, the attributes are reversed.

One could force the button to relinquish its focused state forthwith when it is inactivated, with the focus moved to, say, the first or last thumbnail automatically. This would work well for visual users but would render the scroll event difficult to understand for screenreader users who cannot see where and why the focus jumped. For this reason, the button will not eject keyboard focus but the user will rather have to remove it themselves.

General Implementation Notes

- The basic controls of other carousels will also almost always have a button role as well: Use either a <button> tag or give the element a role=”button” attribute if it is not natively one (e.g. it is an image or).
- In the latter case, remember to add the necessary keyboard event handlers and make the element keyboard focusable with tabindex=0.
- When deciding what aria-labels to use, describe the function of the button, not its visuals (i.e. “Switch to the previous image”, not ”Arrow pointing left”). Also, avoid geometrical clues: Use “previous”/”next” in place of “left”/”right”. Assistive technology will not expose web pages after a visual geometry, so left does not usually “point” in any direction for the user.
- More information is available in the City of Helsinki Accessibility Model training material.

Thumbnails

Demo

Thumbnails are miniature images which when activated switch the content panel to show that image. The demo displays five thumbnails at any given time, so even when the list reaches its end, the thumbnail set only scrolls along so many images to prevent vacant thumbnail spots.

Technically, the thumbnails are composed as a tab list, albeit one where there is not necessarily a selected tab displayed (depending on which segment of the list is being presented at a given time). One can argue, thus, that it is not, as a matter of fact, a tab list.

Be that as it may, the tablist pattern nonetheless solves a number of semantic requirements to which the list and content panel must adhere: Firstly, a tablist denotes the selected image as the selected tab using a familiar Aria-selected pattern.

In like manner, the pattern bestows on the list and its tabs a structured beginning and end, and lets the user know at which tab the focus is at any given moment. What is more, each tab can be labelled in a manner that is automatically reflected in the content panel owing to the tablist pattern’s rules (see the guide for tab lists for more information). In other words, the tablist prevents one from having to reinvent the wheel.

There are drawbacks to choosing a tablist as well: The user must recognize the pattern, and be able to mentally map it against the other parts of the element set in order to

navigate between the tabs and the selected content (panel). This is not straightforward, as the panel must follow the list in DOM order, not the selected tab utself. Even experienced screenreader users will have to tread with thought while operating the slider or carousel.

Finally, for keyboard users the list sports a tailored focus order compared to a standard tab list. This is to align focus order with the novel condition that the list does not always show a selected tab; more on this below in its own chapter.

General Implementation Notes

In addition to a tablist, another pattern that would fit for this purpose could be built like so:

- Each thumbnail is a button with an aria-label describing its number, name, and possibly also position within the list being shown. E.g. “Image 17: Mediterranean Vista; number 4 out of 5”.
- Here, you cannot use aria-selected! Only tab-pattern “buttons”, which are not action buttons, support this status. Instead, you must use aria-pressed=”true / false”. This defines the elements as toggle buttons that are pressed/on, or unpressed/off.
- The panel container must be defined with a structural role – e.g. a <section> tag, or a with a role=”region” attribute. Preferably, it should also have a name; an aria label, or aria-labelledby=”id” referencing the selected button, will work.

Other patterns can be constructed as well.

If the carousel lacks a list of this kind and rather uses some other form of a “quick” selection, one must adjust to the requirements of that structure in a similar fashion. Of course, a carousel that employs no list is much simpler to build and use, as the discussion above no doubt strongly hints.

What about using a long “flat” list that is only visually segmented?

Please see notes on using a single unsegmented tab/button list in chapter 3, Reading order for assistive technology.

Panel Content

Demo

The panel hosts an image and a textual caption segment. For illustration, a number of labelling techniques are used: There are panels where only the image ALT text is exposed to assistive technology and the caption hence hidden (aria-hidden=”true”); there are, also, images that expose both an free-form, more elaborate ALT and the standard caption; and finally there are images with no ALT text at all.

None of these constructions is correct or incorrect per se. The purpose of the carousel/slider is what decrees in which fashion its panel content should be described to assistive technology. There may, for instance, be interactive elements within, in which case you need to consider not just how the elements are named but also how they are ordered. See the requirements on focus and reading order for discussion.

General Implementation Notes

- Mark up the content area with a role=”region” or a <section> tag to distinguish it semantically from other parts of the element set. This region can be named with aria-label or aria-labelledby.

  There are exceptions to this rule:

  - A pre-structured card/box can be placed in the DOM as-is, negating the need for a separate panel container. If you are unsure how to go about this, use a host container, as it is always correct.
  - If you use an existing accessible tab list component to display slider/carousel items, the panel is already defined and need no additional semantics. Simply place the item(s) inside the panel.
- If the panel content comprises multiple parts (e.g. an image/video with a separate caption), consider whether clarity is improved by demarcating such auxiliary content from the main content using a named region. If, however, the content is already pre-structured, as in a linked card/box, you can often leave it as-is to avoid inundating the user with excess semantics.
- If the panel content is interactive (clickable), add either a button or link role to the item that triggers the action. Do not add the role to the panel container itself, as its accessible name would then “overwrite” the entire panel sub DOM and thus render the content difficult to parse with a screenreader.
- Make sure there is an image ALT or button aria-label if the trigger element has no textContent (is missing an accessible name).

Progress indicator

The demo does not at the time of writing feature any graphical or other indicator(s) that would depict how far along the list one has scrolled. For a slider (that will not loop) this may well be useful, and some sliders incorporate a progress bar or similar visualization adjacent to the list, allowing for a quick view of one’s position in relation to the list length.

An indicator of such a fashion will likely be added to the demo in the future. For now, however, the following pattern may serve as a starting point for implementing a progress bar or a congruent indicator element.

General Implementation Notes

- Give the indicator wrapper element the following attributes and corresponding values:
  - role="progressbar"
  - aria-valuenow="selected/displayed segment/item ’number’"
  - aria-valuemin="1 or equivalent value to starting progress"
  - aria-valuemax="equivalent number to 100% progress"
- Wrap the progress indicator in its turn inside a container that is a live region: aria-live=”polite”

Now, when you update the progress indicator, give it the respective “valuenow”. A screenreader user will hear the updated progress as a live announcement and will not have to find and focus the indicator to learn its status. If, however, that is undesirable, simply leave out the live region.

Limitations

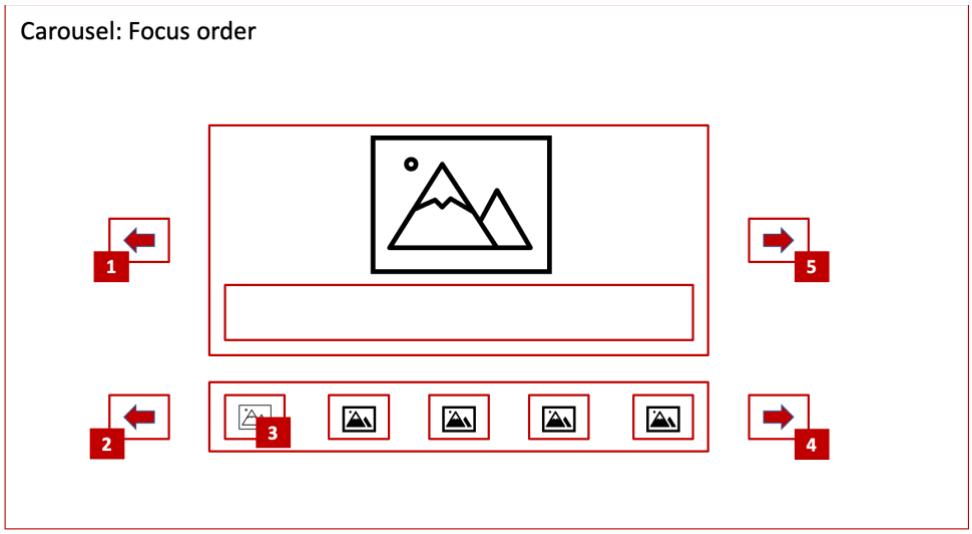
Not all screenreaders will announce the live region. This is because by specification live regions need only be monitored against an accessible name, which does not change here. If that is an issue, you can give the progress bar an aria-label in place of the valuemin, valuemax and valuenow tuple, and update the label instead – although you have to come with the phrasing for the announcement yourself.

That said, iOS VoiceOver will not, in turn, respect that aria-label change due to a bug in Apple’s VO engine. The most robust solution, therefore, is to use a visually hidden textContent inside the live region and have It, instead, describe the progress value. One conjectures most people will just go with the standard valuemin and valuemax attributes, and accept the compatibility comprises they may entail.

Requirement 2: Focus Order Follows Service Design

Demo

The picture below depicts by numbers in which order the different parts of the demo carousel receive focus when navigated with a keyboard and the tabulator (TAB) key. Note that this order is not the same as the DOM order (shown in the second picture below), as the TAB key will only focus interactive elements.



The tablist receives focus based on the following rules:

- Focus is placed on the selected thumbnail/tab, if there is one in the visible list
- Otherwise, focus is placed on the first or last thumbnail depending on whether the left or right hand scroll button has been focused last. That is, the user will enter the list from the direction they are navigating.
- Once focus is placed on a tab, the user can move between tabs by using left/right arrow keys. Enter/Space will select a tab.

This tablist focus order and keyboard navigation mode mimics the common “manual selection” mode of the tablist pattern; for more information, see the tablist implementation guide.

Discussion

This focus ordering has as its notable feature the image switch buttons’ placement: They are situated before and after the tablist and its scroll buttons, “far away” from the content panel. On one hand, this makes moving focus from one image switch button to the next cumbersome, as the entire carousel lies in between; on the other hand, the tablist now has its scroll buttons right beside it.

This makes the list easy to operate even without mouse or touch. But will that help the user more than the image switch buttons’ separation hinders her? That is a service design decision with no strict right or wrong answer.

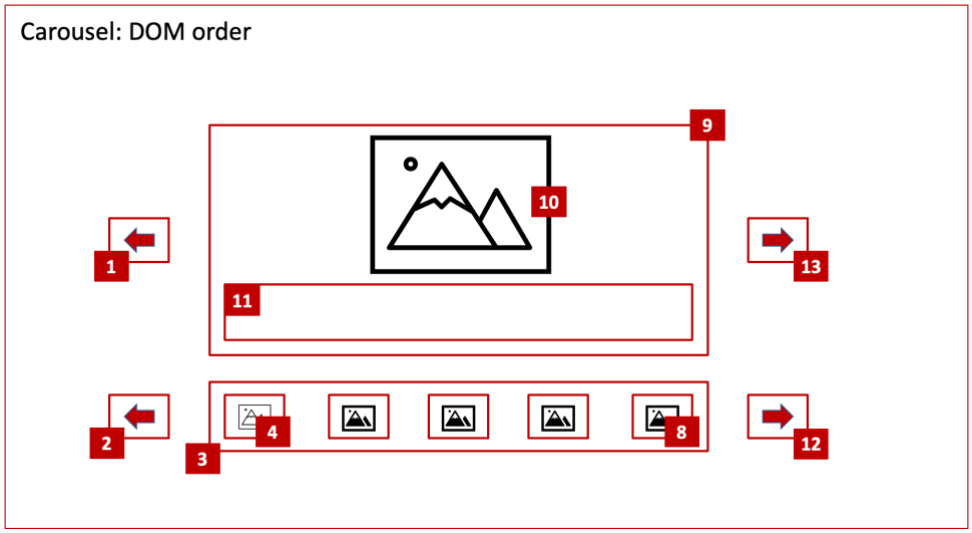
While the above choice, then, is hamrless, the following focus order would not work given that it breaks the tablist pattern for screen readers and other assistive technology:

- Place the image switch buttons between selected and next tab – that is, “wedge” (in focus order) the content panel in the middle of the tablist.
- How would the list now be defined? And if the solution is to make it a toggle button list, what happens when the selected image is not in the segment that is currently shown – where can it be accessed? This pattern, clearly, will not work.

General Implementation Notes

- Focus order is dictated by the order in which the elements lie in the DOM. It is neither affected by, nor has any bearing on, the visual layout.
- Decide the focus order based on what serves best a user who operates the element using only a keyboard. That is, you need not follow the visual layout strictly if another order is more effective.
- Arrange the element DOM according to the desired focus order, not according to an arbitrary visual layout. Instead, control the layout with CSS wherever possible.

Requirement 3: Assistive Technology Reading Order



Demo

The above picture illustrates the DOM order employed in the demo carousel. Compared to the keyboard focus order, it includes all the element parts, as all DOM nodes are exposed to assistive technology. The screenreader will focus all page content unless a tag is empty or hidden.

Noteworthy in this diagram is the location of the content panel. Since this is a tablist pattern, the panel follows immediately after the tabs – it cannot be in any other place. That requirement, consequently, pushes the right-hand scroll button onto the other side of the content panel – a propensity that may feel out of place for visual users but makes sense to a screenreader user who is unperturbed by the visual alignment of content and is solely interested in the correct semantical placement of each element. After a tab list there always is a tab content panel.

If one were to deviate from this order, one would have to dispense with the tablist and go for a toggle button list instead. That would not be incorrect, as discussed in the first chapter, but all the other parts of the design – keyboard focus order, technical definitions – would have to be reviewed against that revision, so it would not be an innocent drop-in-place change.

General Implementation Notes

If you employ a tablist/toggle button list (including clickable dots / little circles, which will likely number in that category as well), consider whether you can make the entire list available to keyboard users and assistive technology, or whether having a scrollable list, like above, still works better.

That is, even as the list may be visually segmented, it can remain unsegmented technically and so be navigable from one end to the other without the user having to click the scroll buttons.

The list can simply update its scroll position as the focus travels along.

- Benefit: A simpler focus order and no need to expose the scroll controls to keyboard and assistive technology. The carousel is easier to use.
- Drawbacks: If the list is too long, it is no longer easy to navigate from a selected item to its content panel.

  Moreover, the carousel/slider will form a barrier on the web page hindering navigation, as the user must browse the entire list inside the carousel just to pass it even when they are not interested in the carousel itself. A separate skip link may need to be added before the carousel.

  → If the list length is more than 10 or so items, it is better to not flatten it out as a single list but to segment it to parts between which the user must scroll.

  An exception might be a carousel that hosts essential content, such as product links that are not available elsewhere on the page. In that case, one might prefer to flatten out the list to make the page easier to understand, the list's length notwithstanding.

Conclusions

Although many sliders and carousels will depart from the list-based structure used in this document, it seems reasonable to assume most elements of this type share the overarching requirements laid out above. A persistent feature of these requirements is that all the parts of the slider must co-align not just in terms of keyboard focus and reading/DOM order, but also in terms of the roles, states, and even accessible names of each subcomponent. It is this complexity where every decision impacts every other decision that makes "perfect" sliders so difficult to build.

In real world projects, few people have the time to create, or even tune, a complex element like this to that extend. The pragmatist will, rather, decide what the key service design objective of the element is and work to make that task accessible – and no more than that.

It is, indeed, the opinion of this guide's author that unless the carousel is tasked a critical role on the web page, the development team can safely prioritize visual keyboard and pointer use over assistive technology when building or adapting a slider. That is, the element must be accessible, of course, warranting correct names, roles, and values for all components, but no matter what one does with the element it will remain difficult to use for screenreader users. There is a lot more return for the time invested when it is spent on optimizing the element for the users who most benefit from its inherent visuality: The keyboard and the pointer (mouse, touch) users.

This means, for instance, that you can set the focus order to one that makes visual keyboard use efficient; this way, both design and implementation are also simplified.

As an example, the demo carousel, although limited in fucntionality, has been focus-order after this principle, and even though it is therefore not necessarily optimal for screenreaders, it nonetheless remains easier to use with assistive technologies than most carousels out there.