Helsingin saavutettavuusmalli

Helsinki Model for Accessible Service Design

Navigation Menus: Notes on technical implementation

Tero Pesonen

Q-Factory

Version: 15th March 2023

Navigation menu pattern

Notes on implementation: Quick reference

If the menu is built from an element that is not a native <button> tag, you may have to apply all of the below attributes marked as SPAN before you apply the attributes marked as BUTTON; if, however, the menu button is a <button> tag, you only need to consider the attributes marked as BUTTON, since the built-in button tag already provides the aforementioned functionality.

Example: https://accessibilitydemo.net → Tab 1 / Buttons → Menu Button

The menu button is built from a span/div/img

- The menu button ("burger button", Dropdown button, …) requires the following Aria-attributes
    - Always: Role="button".
    - Never: role="link" or an anchor <a> tag. A menu is never opened from a link.
- The menu button requires the following attributes to enable visual keyboard-only use without assistive technology:
    - Always: tabindex="0": This tells the browser that the tag is interactive and hence focusable.
    - Always: Key event handler (onkeydown or onkeypress) mapped to Enter and Space keys (keycode 13 and 32), to mimic a button. In Javascript, event.preventDefault should be used to stop e.g. Chrome from scrolling the window on a space key press.
- Note that the button may require an accessible name (e.g. aria-label) if it has no textContent but only a symbol. See the notes following this quick reference for more info.

The menu button is built from a <button> tag

- No special attributes are required.
- Note that the button may still require an accessible name (e.g. aria-label) if it has no textContent but only a symbol. See the notes following this quick reference for more info.

All menu buttons (span/div/img/<button>/…) additionally require:

- Aria-expanded="true"/"false". This attribute must be updated by code to reflect the open/closed state of the menu which the button controls. The attribute will not update automatically in response to the user's actions.
- Aria-controls="menu-content-container's id": The attribute tells assistive technology that the button controls the presence and content of a specific menu panel. The can be set permanently on the button as it need not reflect whether the menu panel is visible or not.

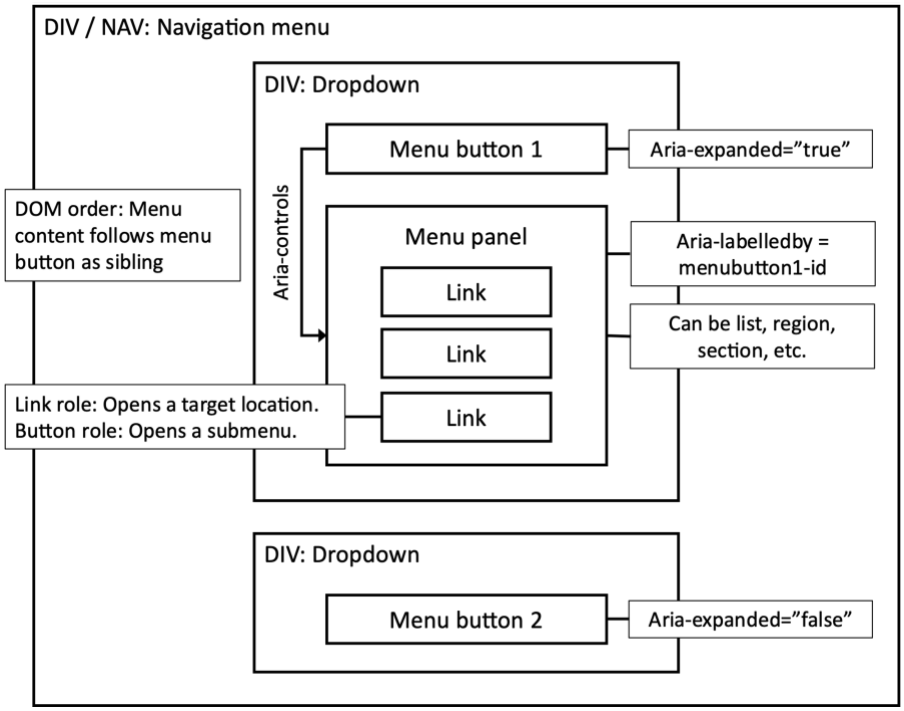
All menu buttons and the menu content may additionally require:

- Aria-label="description": Provide the screen reader user with a label in case the menu button has no textContent or innerHTML that gives it a name (e.g. <button>description</button> has a textContent and so needs no aria-label).
- If the menu has a navigation role, the menu may be wrapped inside a <nav> or given the role="navigation" attribute. Aria-label can be used to give the navigation landmark a name.

- Alternatively, it is possible to give the menu content (menu panel) a section tag or a tag with a role=”region” attribute, and an aria-label=”Menu” (or a more specific name for a particular menu amongst many), or aria-labelledby=ID (where ID is the menu button).

Submenu of a navigation/dropdown menu

- Submenus may be implemented recursively by the same pattern as above, so that a submenu entry is a menu button, too.
- If the submenu can be opened and closed, it follows the accordion pattern and thus requires an aria-expanded state.
- If the submenu will instead replace the parent menu content entirely, and the user can only navigate between these levels as if separate pages, then the menu button does not have an expanded state, since it only an action button.
- NOTE: Focus/navigation order within the menu should be specified before implementation, so that the DOM order can be built accordingly. This can be taxing to fix afterwards if the DOM order and CSS rely on one another heavily.



Explanation: Menu pattern Aria attributes

- **Role="button":** Native HTML components will automatically announce their role/type to assistive technology. That is, a button will proclaim to a screen reader, "I am a button", by which means the SC can describe the element correctly to the user. However, when a span or div is tailored as an element that corresponds to a native element type, screen readers will not automatically be able to deduce that functional relationship.

  It follows that although a span button may look and act like a button, assistive technology will not usually recognize its button-like status. The tag will therefore have to describe its type unambiguously. For example, a or that acts as a button needs role="button" added. Many other Aria roles exist for similar mapping to standard HTML element types.
- **Aria-expanded="true"/"false"** designates for assistive technology whether the content which the button controls is expanded or open (="true") or closed (="false"). The attribute is used with accordions, burger menu buttons, dropdown menu buttons, pop-up dialog triggers (when the dialog is NOT a modal but acts like an accordion), and other similar cases where a button will a) expose or hide content that instantly follows in DOM order but b) where the exposed content will not capture focus like a modal would.
    - Notes:
        - The attribute is assigned to the element which triggers something to expand, that is, the button itself; it is NOT assigned to the content which is shown or hidden (e.g. the menu panel div).
        - Screenreaders will announce the aria-expanded status. It is therefore unnecessary to describe the element's status in its label:
            - So, this suffices:
                - <button aria-expanded="false">Menu</button>

        - <img class=”menu_glyph” …></img>
    - But this is redundant:
        - <button aria-expanded=”false”>Menu (closed)</button>
        - <button aria-label=”Menu, closed.” aria expanded= ”false”> Menu </button>
    - If this accidentally happens, though, it is not a big problem.

- **Aria-haspopup=”true” or “menu”:** With this attribute, assistive technology will describe the element as one that provides a menu role. The Aria menu role entails a specific kind of a menu whose container has the role=”menu” attribute and children the “menuitem”, “menuitemradio”, “menuitemcheckbox”, or “group” roles.

  Implementation details are beyond this basic guide, as this menu triggers a forms mode on some screenreaders, in consequence of which the developer must manage keyboard focus in response to arrow keys. That is, the user cannot navigate the menu in the standard manner as a group of DOM elements, but the menu is rather treated as an application-like context or like a composite form control. On most web pages, this menu pattern is unnecessary.
- **Aria-label=”description”,** provides the element with a label which assistive technology will read out to the user when the element is focused. It is an alternative to using aria-labelledby=”ID”, where the description is rather derived from the label of the element whose ID is referenced. Note that aria-label will OVERRIDE any other accessible name the element may have. So, if the node also has textContent or innerHTML (even with childNodes), those are not read out. For instance:
    - X

      Here the SC will describe the link as, ”Link to X (opens a PDF)”, **not** “X”, **nor** “Link to X (opens a PDF), X”, since the aria-label replaces any other labels.
    - NOTE: if the element has an additional aria-describedby=”ID”, that description, derived from the reference, WILL also be read, as the purpose of aria-describedby is to endow the element with additional information beyond its name or label (e.g. instructions, etc.). In a more technical sense, aria-describedby is not the element’s accessible NAME, and is hence not negated by aria-label (or aria-labelledby). It is, thus, normal to use aria-label together with aria-describedby; this can be particularly effective in modals, but that is beyond the present discussion.

Explanation: Menu keyboard navigation with tabindexes

- Screenreader users can focus and interact with any element that can be reached by traversing the DOM tree. From accessibility point of view, this is not, however, the only use scenario for which one has to account. Users who operate the web site without mouse and screenreader are an important group as well. What is more, some assistive technologies rely on keyboard I/O, too.
- An important point: Keyboard navigation will not focus all elements but will only focus elements known to be interactive. This list includes by default all native buttons and links, for instance.

RULE: If the element is one that is not by default keyboard focusable, yet it can be interacted with (e.g. it is a span, div, web component generated from a random tag, etc.), it needs to be given a tabindex attribute value that is zero or greater. The browser will then include the element in its list of focusable elements that the user can TAB into.

- **Tabindex="0" should be normally used.**
- Tabindex > 0 will alter navigation order. Avoid this, and adjust focus order with DOM order instead, as it is easier to maintain and is compatible with all assistive technology.
- **Tabindex="-1" prevents** the element from being keyboard focused (but WILL NOT exclude it from screen readers' DOM! For SC, use aria-hidden="True" to preclude content from being focusable).
    - Moreover, tabinmdex="-1" gives the DOM element a special property: Focus can be programmatically transferred to the element by e.g. using JS DOMElement.focus() method. This is true even when the element cannot accept keyboard focus. For instance:
        - <h2 id="sample-h2" tabindex="-1">Section heading</h2>
        - Now, one can e.g.

          document.getElementById("sample-h2").focus();

          (But without tabindex=1, this will fail on most browsers!)
    - This is useful on a number of special occasions where it is expedient to help the user by moving their focus on their behalf, even though normally it is discouraged, as a focus that "jumps around" "on its own" can be very confusing.
- If the element needs to be given tabindex=0, **it will likely also need to be furnished with a keyevent handler** to ensure that keyboard users can trigger the element with "Enter" / "Space" (or other keys pertinent to its operation; this is relevant on radiogroups, certain kind of tablists, etc.). Tabindexing alone will not bind Enter or Space to a click event.
    - NOTE: Screenreaders that are operated by keyboard do not, actually, require keyevent handlers, as they send also a click event. Enter/Space keyevents are needed for visual keyboard users who are not running a screenreader.

Menu button anti-patterns: Things to avoid

- Role=”menu”, role=”menubar”, and role=”mnuitem”
    - Role=menu is nowadays used primarily for such menu contexts as correspond to application menu bars and implementations that differ from traditional web page navigation (link) menus.
    - Screenreaders may expose role=”menu” in an unusual fashion, confusing users who are unaccustomed to them on a tradition web page that does not try to mimic an application.
- Treegrid and related roles
    - Like the menu role, this is reserved for more specific use cases than a normal link menu hierarchy and will add unnecessary complexity to a standard menu.
- Menu panel will not follow menu button in DOM order, or
- Menu panel is a child of the menu button
    - This will usually lead to navigation/focus order issues. Strive for a design where (1) the menu button is a child to a menu container, (2) the menu panel, when opened, is also a child to the menu container but a sibling (not child) of the menu button.
    - This seems to work reliably for assistive technology.
    - Below, is an example of such a DOM structure for a traditional (desktop size) navigation menu with side-by-side drop down menus containing links.