Helsingin saavutettavuusmalli

Helsinki Model for Accessible Service Design

Modal Pattern: Notes on technical implementation

Tero Pesonen

Contact: tero.pesonen at q-factory.fi

Version: 4th March 2021

Modal pattern

For implementation, please take into account all the “Implementation” chapters of this guide: Screenreaders (chapters 2), keyboard navigation (chapter 3), and pointer devices (chapter 4).

1. Definitions

The term **modal** when used in these instructions refers to a web page context that has the following properties.

Modal features

- A modal is a web page section that withholds keyboard and screen reader focus within its content area until the modal closes.
- That is, when the modal opens, focus is automatically transferred into its area on behalf of the user. The user cannot leave the modal but is constrained to navigate only within its area or "layer" (e.g. a visual dialog, but could be any shape or appearance).
- When the modal closes (e.g. from a button within the modal), focus is normally returned to the element that triggered the modal context from the preceding contextual layer (another modal or the document body). Of course, a modal can trigger a new page context to load as well.

Modal pattern main Issues

To meet the above requirements, implementors must consider a number of issues; the main ones are:

- How to manage the focus transitions to/from a modal.
- How to constrain keyboard, screen reader, and pointer devices so that the user can operate only within the modal.
- How to describe to assistive technology the context change that occurs when a user enters or leaves a modal.

Other Issues that may have to be considered

The latter point, that of describing the modal to assistive technology, can be addressed with the correct use of Aria 1.1. techniques. The other points, however, cannot be managed by Aria, but need to be tackled in code, as they pertain to general web page dynamics.

Please note that neither keyboard nor a pointer device when used on their own are considered assistive technology. For this reason, the browser will not honour any Aria attributes when it receives input from the keyboard or mouse or touch device.

It follows that developers may also have to consider:

- How to deal with mouse/touch/pointer devices when the user "clicks" outside the modal.

- How to prevent "lower level" content defined with CSS position:absolute from showing "through" a modal when rendered over the same area.
- How to manage window scroll and other rendering issues

Depending on the context, an existing design system or other library component(s) may already provide solutions to these problems.

Finally, the most important decision is to determine when and where to use a modal in the first place in lieu of a pop-up type context that does not confine the user.

This is a service design decision, one that should not be treated lightly. The modal and pop-up/accordion patterns facilitate different, sometimes even opposite, uses cases given their propensity to enable and limit the user's progression in different ways.

2. Implementation: Screen readers & Aria 1.1 definitions

Mandatory attributes

From Aria point of view, a modal is a container (e.g. a div tag) with the following attributes:

- role="dialog"
- aria-modal="true"

When aria-modal=true is used, the screen reader is tasked to automatically limit the user to the modal area as if it were the only navigable content in the browser window.

The developer, therefore, need not hide other page content from the screen reader in code, but can rely on the assistive technology to do this automatically. Older techniques, such as applying aria-hidden to all content outside the modal, need no longer be employed.

Non-mandatory attributes

Additionally, the container may have the following attributes, which provide assistive technology with the desired accessible name and automated description(s):

- aria-label=”Modal name”, or aria-labelledby="ID". ID refer to an element that names the modal.
- aria-describedby="ID". ID refers to an element whose accessible name describes the modal to the user.

Naming

While different naming patterns befit different use cases, according to WCAG 1.3.1, Information and Relationships, at least aria-label or aria-labelledby should be employed for all modals. This can be done by e.g. referencing the modal's first heading with aria-labelledby.

In a simple modal that accepts a yes/no answer or only informs the user of an important event, naming and focus management can also be aligned for an even easier user experience.

For example, aria-describedby can be used to read the dialog content to the user, making it unnecessary for the code to place the user's focus at the beginning of the modal or at some 'earlier' secondary interactive element (such a cross-shaped close button at the top of the dialog). Instead, the focus can be placed on the preferred button, which permits the user to make the choice the modal requests without having to navigate and peruse the dialog content -- a non-trivial task for a screen reader user.

Modal with Aria attributes

```
<div role="dialog" aria-modal="true" id="example-modal" aria-
labelledby="example-modal-h" aria-describedby="example-modal-
description" class="ModalWindow">
    <div class="ModalTabBarrier" tabindex="0" aria-
    hidden="true"> </div>
    <h2 id="example-modal-h" tabindex="-1">Modal Heading</h2>
    <p id="example-modal-description"> ... </p>
    ...
    <div class="ModalTabBarrier" tabindex="0" aria-
    hidden="true">  </div>
</div>
```

3. Implementation: Keyboard focus management

While the above aria techniques will cater to the screen reader, they will not address keyboard navigation that takes place without assistive technology (that is, when one browses the web page using the 'TAB' key and interacts with focusable elements with Enter/Space and arrow keys.) Nor will the Aria attributes impact non-assistive pointer devices (mouse, touch).

There is no one right technique to manage keyboard focus in modals. The implementation is up to the developer and library/framework being used. That said, the demo page shows a simple and easy to adopt method that should work in most situations. It is based on TAB barriers / guards, and is implemented as follows.

Tab barriers: algorithm

1. Place two TAB barriers in the modal container as its very first and very last elements, respectively. Make sure no other content will be appended before or after the barriers.
2. Assign each tab barrier with:
    - Onfocus event listener
    - Tabindex = "0" attribute
    - aria-hidden="true" attribute
3. Define each barrier's onfocus event as follows:
    - Find the first and last interactive elements (that is, keyboard focusable DOM nodes) in the modal area.

- Command the onfocus event to move focus (with e.g. DOMNode.focus() method) to these interactive elements based on at which end of the modal the barrier resides; that is, the barrier at the end of modal will when receiving focus forthwith bounce the focus to the first (true) interactive element of the modal, and vice versa.

Tab management rationale

The effect of the above arrangement is that the user will be demarcated inside a loop when navigating forward or backward with the TAB key, unable to accidentally "drop off" the modal "back to the previous layer". Effectively, this will apply to keyboard navigation the same limits aria-modal=true applies to a screenreader.

To make sure that the focus will not escape the modal even when the user, say, clicks the address bar and then TABs back to the page, place similar barriers (e.g. as simple node copies) also as the first and last elements of document.body. They will similarly bounce focus back to the modal's first and last interactive element, respectively, giving the user the illusion that no other content in the browser window save the modal is focusable (till the modal is closed and the barriers, of course, removed.)

4. Implementation: Pointer devices

To manage pointer devices, many approaches can be utilised. The simplest method, perhaps, is to create a "blocking layer", a div with position:absolute that covers the whole viewport. The modal is then created as a child element of the blocking layer, which thus prevents outside clicks from focusing "lower layer" elements as they perforce hit the blocker div.

If the modal can be taller vertically than the viewport, it may be necessary to also scroll the viewport to the top upon creating the blocker and then prevent document.body from scrolling until the modal closes, to retain the blocker in place and to allow the window scrollbars to scroll the modal dialog area instead of the document body "below."

On the demo page, an additional technique that works without a blocking layer is also tested: This involves catching all bubbled document.body click events and then examining, based on the source of the element, whether that click originated from within or without the modal dialog. With this knowledge, one can either block the event or pass it through, effectively blocking clicks from outside the modal.

This examination can be done by ascending the DOM tree from the event trigger node and checking which of the two, the modal container or the document body, comes up first.

Of course, one has to remember to remove the document.body event listeners when the modal closes, so the blocking layer technique is probably easier to use in most situations.

This and the other techniques discussed above can be perused in the modal element source code below; note that the author is not a professional developer, so the code is meant to be only instructive, not authorative on this topic.

5. Examples

Examples can be found on the demo page:
https://accessibilitydemo.net