Helsingin saavutettavuusmalli

Helsinki Model for Accessible Service Design

Single Page Applications (SPAs): Notes on Accessibility

Tero Pesonen

Contact: tpe at siteimprove.com

Version: 31st May 2021

Overview

A Single Page Application (SPA) denotes a web service that updates the existing page DOM instead of loading a new URL even as a new “page” is opened. The application may technically only ever work with a single web page even though the user may perceive the service as hosting multiple pages and vastly different sets of content.

On a SPA, the browser is unable to cater for some of its usual tasks which the user (and developer) normally can rely on while navigating across a web site composed of traditional static pages. This is because the browser cannot know when a “new page load” event triggers, as most SPAs only ever update their DOM in-place, bypassing the traditional web browser – web page interface.

The developer, therefore, must carry out some of those interface tasks on behalf of the browser, so that the user can still perceive the web site as if it were a traditional service composed of multiple web pages. Users, of course, need not understand and adapt to implementation-level issues. They simply operate the browser in the web, and the web should behave in a normal, expected fashion.

To this end, in a single page application the developer needs to:

1. Manage browser focus following each “page” switch
2. Manage the browser’s history
3. Update and announce page titles

Page switch

For this guide, we define a page switch (or page load) as a content update in the SPA that causes a change of such magnitude as would warrant loading a new page (URL) on a normal, static web site. That is, the page being displayed changes *in purpose*, not just in content. It serves a new service context.

An example of such a change would be clicking a “Purchase” button that replaces the page content with a re-rendered view part of a purchase process. One would likely load a new web page in a traditional service for this view; hence, one would initiate a “page load” event in a SPA as well. On the other hand, showing search results below a search filed, for instance, would normally not require a page switch, as one could continue to serve the same page on a traditional web site as well, with only the bottom half of the page updated. In this case, a page load event would be non-mandatory on a corresponding SPA implementation.

A page load event on a SPA requires, in practice, that one implements all the usual procedures that a browser would take care of when opening a new page. These include placing the focus at the beginning of the new page, updating the page title attribute, inserting (or replacing) an entry in the browser history, and so forth.

Focus management following a page switch

Rule of thumb

When you switch to a new page context, move the browser focus to a pre-designated spot in the new page. Normally, this position is at the top of the page, on the first DOM element. On most pages, that element is the “skip to content” link.

- You can do this in Javascript with the HTMLElement.focus() method.
- If you need to focus an element that is not inherently focusable (span, div, etc.), give the element the following attribute: tabindex=”-1”. The tag can now be focused programmatically but will remain unfocusable by the keyboard.

Rationale

If the page is switched but focus is not transferred, the browser focus may land at a random location on the new page or even detach from it altogether, as the DOM node that hosted the focus is removed from DOM. Non-visual users in particular will find the situation difficult to comprehend. Moreover, all users are frustrated by having to start browsing the new page at a random spot possibly closer to the end than the beginning of the page.

WCAG 2.4.3 Focus Order

Exceptions

There is no exception to the rule that the focus need to be managed following a page switch. However, not all page switches need to place the focus at the top of the page. Some common exceptions include multi-page forms and similar processes where the user moves along a specific set of steps that they well understand. Although each step may result in a page switch, the user is often best served by not having to navigate back to the form following a form page switch. Instead, they should be able to simply continue working with the next step.

To this end, the focus can be placed at the beginning of each form page or set of process widgets instead of the beginning of the entire page. Now, it is easier for the user to proceed along the process. This method is demonstrated on the demo page form:

http://siteimprove-accessibility.net/Demo/Page/

If you are unsure where to place the focus after a page load event, move it to the top of the page. It is always correct.

Title management

Rule of thumb

1. Give each SPA page context a title
2. When you switch to a new page context, update the page <title> tag in <head>.

3. Non-mandatory but recommended: Announce the changed title to assistive technology users.

Rationale

When a web browser opens a new page (URL), screen readers observe the new title and announce it to the user as part of the page loading process. This helps the user understand that, first, a new page is being opened, and secondly, what the new page is.

WCAG does not specifically require that titles be announced, as it is the task of assistive technology to describe title tags (WCAG 4.1.3 Status Messages does not cover this area). In practice, however, screen readers only describe a title text when a new page is loaded; they do not actively monitor title changes, which is a problem on a SPA. For this reason, the limitation is imposed by the current assistive technology, but that fact does not help users who nonetheless are impacted by the issue.

Therefore, it is recommended, but not mandatory, that developers make sure that title changes are announced on SPAs.

WCAG 2.4.2 Page Titled

Announcing <title>

There are two main approaches.

Technique 1: Simple approach

1. Create a fixed live region in the page. A natural location for the region is at the beginning of document.body, after the skip-to-content link, although other DOM locations can also be used.
2. Give this region the following attribute: aria-live=”polite”. Like so:
<div class=”page-title-live-region visually-hidden” aria-live=”polite”>
2. When a new page loads, insert its title string as textContent of this live region.
3. Whenever you update the title tag, also update the live region textContent accordingly.

   Result: Screen reader users will hear the page title being read out to them every time the page changes, denoting a page switch.

Pros: Easy to implement and will work very reliably with all modern assistive technology.

Cons: The page title is permanently part of the document body, so it should be hidden visually. One also prefers to place it at the beginning (or end) of the page, as it remains permanently part of the page, and screen reader users can observe it whenever they navigate to that part of the page. Placing it at the beginning of the page therefore makes it appear less weird or out of place than if it were found some other part of the page.

Technique 2: A more advanced, non-permanent live region

1. Create a fixed live region like above:

<div aria-live=”polite” class=”page-title-live-region visually-hidden”>

2. When the page title is updated, insert the new title as the live region textContent like above.
3. But after each title update, now wait 500ms, and then clear the live region of any textContent.
   The delay is necessary for assistive technology to register the altered live region DOM. Without a delay, the deleted text content will not be read out.

Pros: The region remains empty save for a brief while immediately after a page switch. This way, the user still hears the new page title but will not come across the (awkwardly placed) title text string when browsing the page. To further secure this point, the region can be placed at the end of the page.

Cons: You have to program in a delay when manipulating the live region, which is more complicated and error prone than the basic approach above. There is also no guarantee that very old or exotic screen readers will work as expected with this technique.

Extra technique: Keeping title updates inside a common, shared live region “host”

This is simply a more generalized variation of the previous technique. Here, the page employs only a single live region “host” that will collect and manage all live regions and pertinent announcements that the SPA may need to generate during its lifetime.

The technique is used on the demo page for testing purposes. Its aim is to avoid cluttering the DOM with multiple live regions, the management of which can grow cumbersome in a large SPA, and result in unwanted DOM overwrites or litter as regions are placed at random here or there in a difficult-to-predict manner. If, instead, all live regions are hosted in a single place and managed via a standard interface, development, testing and regression (prevention) are easier to tackle.

Using a general-purpose live region:

1 Create a fixed container div that will host any number of live regions needed throughout the app. But do not make this region live. Like so:
<div id=”global-live-region” class=”live-region visually-hidden”>

2 When you need to make a live announcement, apply the following algorithm:

   a. Create a new container that will provide the live message:

   b. Insert the aria-live container as a child of the global live region container created above.

   c. Wait 200ms

   d. Insert the message you want to announce as this new live container’s textContent
   My announcement

   e. Wait 200ms

f. Remove the aria live container. Do not remove the permanent host div.

Pros: Any number of announcements can be generated in an easy to manage fashion without ever cluttering the DOM and without the user ever coming across the message texts on the page while navigating it.
Cons: Complicated; may incur compatibility issues with old screen readers.

Browser history management

Recommendations

1. When you switch to a new page context, add the page title to the browser's session history, and update the current state to reflect the new page.
2. Support browser back and forward operations by capturing them through the browser history API. Respond to these commands by switching to the respective page, so that the user can move back and forward in the history as if they were navigating a static web site. If possible, save page context (input values, scroll position) and restore them on a backward operation.

   More information about manipulating the browser history API is available at MDN: https://developer.mozilla.org/en-US/docs/Web/API/History_API