KRAD - Tab semantics - uplift to code standards (including accessibility)

Purpose

Primary purpose is for KRAD UI tab structures to meet the W3C accessibility standards (WCAG 2.0) - - see usage scenario below. All applications constructed with KRAD will then inherit the benefit, assuming the applications populate the small subset of content tags that require populating with text. In addition, traversing tabs that do not change the entire page contents will no longer cause an entire page refresh, so performance will be better for all users.

Detailed Description

The tab structures within the portal, sample app, and KNS code-base today lack some of the standard tag structures required to meet W3C and government standards (in the U.S., for example, section 508 and 504 of the Rehabilitation Act, which are being harmonized with WCAG 2.0 now that there is a W3C world-wide technology standard). These tag structures define the semantic relationships between and among the tabs, and the behavior upon selecting a tab.

This is not a problem for sighted users, who can infer the relationships from the relative visual placement of the tabs and other visual treatments. For example, sighted users can infer that a set of links that are layed out to look like tabs are, in fact, tabs, and will refresh the content of the space they visually perceive to be the tab-page area. However, links that do not convey that they are tabs because they lack the required standard tag structures, don't convey their tab role programmatically. In real usage terms, this means they can't be accurately interpreted by end users who use standard assistive technologies in order to access applications that use these tabs in their user interface (applications include websites for enrollment, bill payment or other financial handling, curriculum planning, online library systems, online education, in classroom technologies, or other). Users don't get any clues that selecting a link that is intended to represent one of these tabs, will refresh the content of the entire page or a part of it (the area that visually looks to be associated with the link that looks visually like a link) - unless the application developers added code to do this alerting.

KRAD will provide improved support to applications to enable them to more easily comply with code standards, by adding the missing tag structures into the code-base and templates for tabs, for all to use. In this way the appropriate alerting for special needs users doesn't have to be managed by the application.

Usage Scenarios

Users who can't see the user interface, who rely on screen-readers or other technologies to interpret and convey the tab structures and content, are given the (audible/spoken) cues that relate the tab structure, including which tab they are on, with its appropriate meaning.

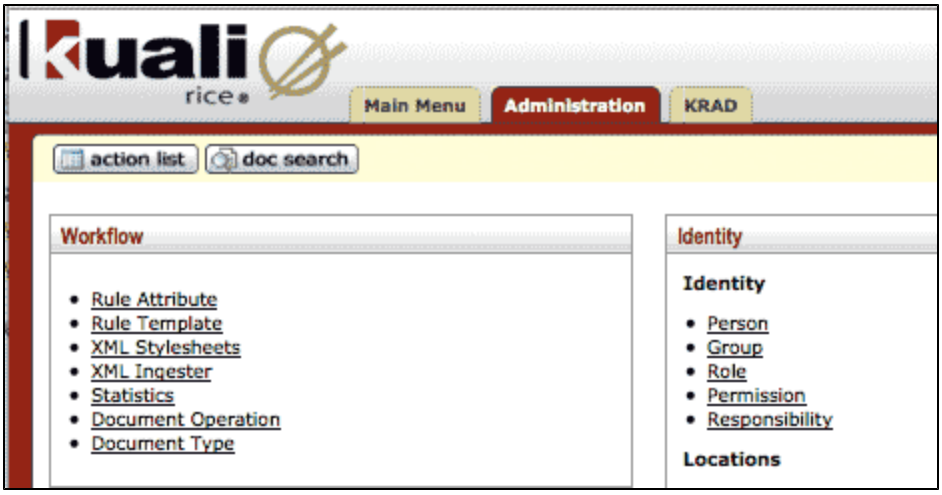
(Note: assistive technologies rely upon standard code structures in order to interpret and convey the content. Additional information is provided in the references section below.)

Mocks and Diagrams

Below are descriptions of the 2 targeted types of tab structures in the Kuali code-base and how they will be brought up to code standards:

Tab Example One: Tabs that are intended to refresh the entire web page

An example of this occurs in the Kuali portal, with the tabs for Main and Administration, and KRAD, and so on (any tabs added/customized by the Kuali applications).



When a tab is intended to refresh the entire page/view, we can mark this up as a simple HTML tablist. Tabs marked up in this way will work in both virtual mode and forms mode, in the widest range of browsers. (At this time the ARIA tablist won't work in virtual mode, only in forms mode, but it has significant advantages over the HTML tablist when it is not necessary for the entire page to refresh. We'll cover this structure later in Tab Example Two.)

With the HTML tablist, when the user changes tabs, the entire page will refresh, including the tab list itself (the tab list will lose focus).

To encode this, start with a traditional HTML tablist:

- Place a heading before the tab list, in the example below, “My Animals”. This heading identifies the tab list and becomes a good anchor point to navigate to. If the visual design doesn’t allow for visual headings, this can be hidden off screen. We show how to do this in the code snippet example below.
- The tab group is an unordered list (<ul>).
- Each tab is a list item in the tab group's unordered list.
- Each tab is also implemented as an <a> element inside the list’s items (<li>).
- Add the tab role and state information to each list item (tab), using hidden text (hidden spans, class="offscreen")
- The tab panel(s) – the content -- associated with the tabs, are placed below the list, in divs.
- The content of the tabs that are NOT currently active is defined as hidden (class=hidden).

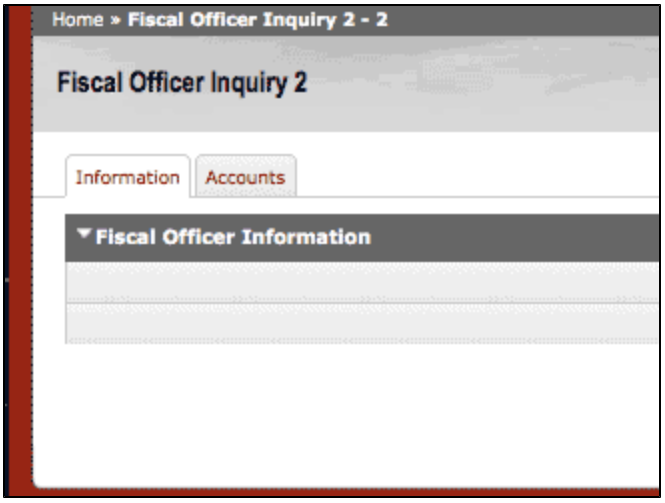
Below is a code snippet example:

```
CSS:
 .offScreen {
     position:absolute;
     left:-5000px;
     top:auto;
     width:1px;
     height:1px;
     overflow:hidden;
 }

.hidden {
     position:absolute;
     left:-10000px;
     top:auto;
     width:1px;
     height:1px;
     overflow:hidden;
 }


 HTML:
<h2 class=”offScreen”>My Animals</h2>
 <ul>
     <li>
         <a href=”#”>Dogs <span class=”offScreen”> Tab, selected</span></a>
     </li>
     <li>
         <a href=”#”>Cats<span class=”offScreen”> Tab</span></a>
     </li>
     <li>
         <a href=”#”>Birds<span class=”offScreen”> Tab</span></a>
     </li>
 </ul>

 <div>
     <h3>My Dogs</h3>
     …
 </div>
 <div class=”hidden”>
     <h3>My Cats</h3>
     …
 </div>
 <div class=”hidden”>
     <h3>My Birds</h3>
     …
 </div>
```

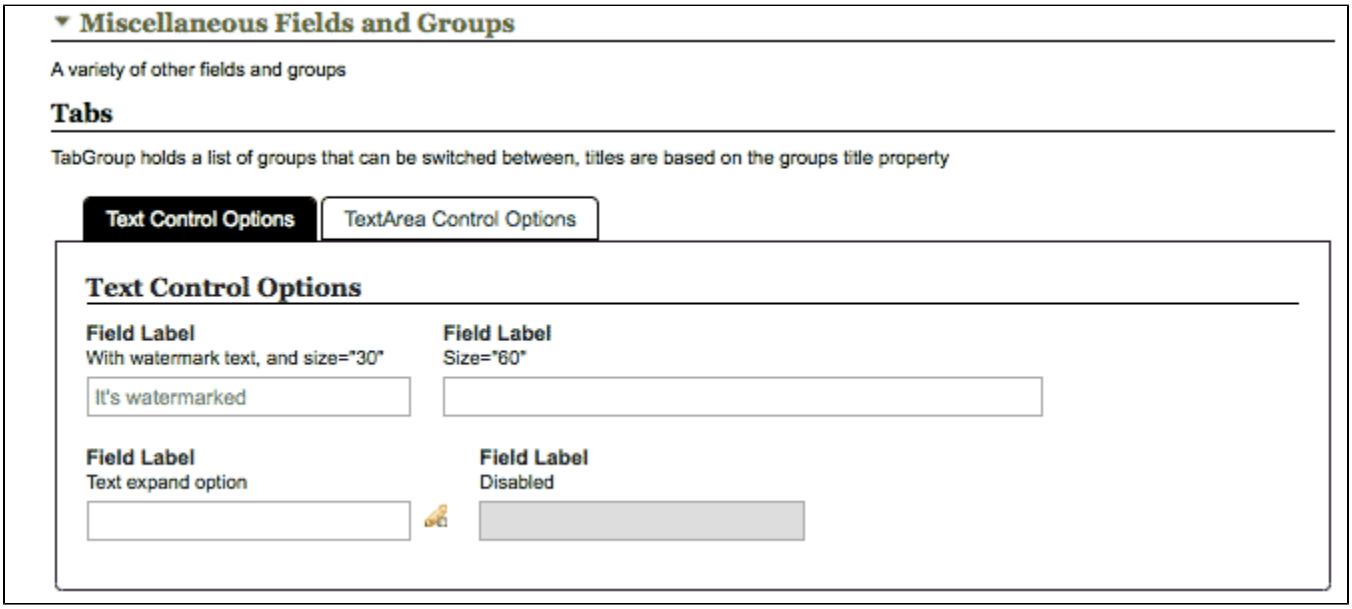
Tab Example Two: Tabs that are intended to refresh only a portion of the page, for example, to a section in a form

Below are examples of these types of tabs from two areas in the Kuali Rice portal and sample application.

From the KRAD Testing – KNS L&F - Fiscal OfficerInquiry 2 form:



From the KRAD Testing – KS L&F - UIF Components (kitchen sink) - Other Fields page (scroll down):



For tab controls that don't apply to the entire page, for example, that apply to a section of a form, our recommendation is that we use ARIA markup. In this case, it isn't OK to refresh the entire page, losing other data and losing the focus on the tablist (which happens with HTML tablists), and it is OK that the tabs are not operational in virtual mode - that they work only in Forms mode (which currently happens with ARIA tablists).

Creating an ARIA tablist

In order to encode an ARIA tablist:

First, we start with the traditional HTML tablist structure.
Second, we add ARIA tags to the unordered list tag (<ul>), to the list items (<li>), and to the divs that follow the tablist.
Third, we add tags to manage the tab order and keyboard focus.
Fourth, we add tags to manage the activation behavior (what happens when a tab is selected).

We'll go through this in the information below.

Step One: Start with the traditional HTML tablist structure (for example, see the Tab Example One section above).

Step Two: Add ARIA tags to the unordered list tag (<ul>), to the list items (<li>), and to the divs that follow the tablist:

- **On the unordered list tag (<ul>), add:**
  - Role="tablist"
  - Give the tablist a title. This can be done in either of three ways. Our recommendation is that we use the first alternative below:
    - role="tablist" title="title goes here">
    - *(... or ... we could create an HTML label element & point its "for" attribute to the tablist container ...).*
    - *(... or ... we could use the Aria-labelledby="<id of the label text element (whether is is visible or not>". This could be a*

hidden heading preceding the tablist ...).

- **On the list items (<li>), add:**
  - Role=“presentation” (conceptually, a tablist should have tabs as children, not list items, so this role hides the fact that these are marked up as list items)
  - On the <a> element inside the list’s items (<li>), define these attributes:
    - Role="tab"
    - Aria-selected="true/false" (set to true for the tab that is currently selected, false otherwise)
    - Aria-controls="<id of corresponding tabpanel>
- **On the divs following the tablist, add:**
  - Role="tabpanel"
  - Aria-labelledby="<id of associated tab element>"
  - Aria-expanded=”true / false” (depending on whether the tab panel is currently expanded or not - it should be expanded when the tab is selected)
  - Aria-hidden=”true / false” (depending on whether the tab panel is currently visible - it should be visible when the tab is selected)

Step Three: Add tags to manage the tab order and keyboard focus.

The tab list should take up one stop in the tab order (keyboard focus). This tab stop should be to the currently active tab. This "single tab stop" approach can be implemented in ARIA in one of two ways: by a Roving Tabindex or by using ARIA-ActiveDescendant.

Our recommendation is that we use the Roving Tabindex, though ARIA-ActiveDescendant would also work.

Create a Roving TabIndex:

- Each tab in the tab list has a "tabindex" value.
- For the tab that is currently selected, the tabindex value is "0"
- All the other tabs have a tabindex value of "-1" (i.e. they are skipped in the tab order but still focusable).
- Whenever the selected tab is changed (e.g. by using the arrow keys or clicking on it with the mouse), scripting is used to move focus to that tab, and update the tabindex values for all tabs to reflect the new situation (the tab that used to have tabindex="0" now gets a tabindex value of "-1", and the newly selected tab gets tabindex="0").
- With this approach it's very important to ensure there is always at least one tab with tabindex="0", otherwise the entire tablist would become unreachable by keyboard.

... or ... alternatively ... you could create an ARIA-ActiveDescendant, instead of a Roving Tabindex:

- The tablist itself is made focusable (i.e. the <ul> element gets tabindex="0").
- The focused tab is then indicated by adding an aria-activedescendant attribute to the tablist.
- The value of the aria-activedescendant attribute is the ID of the tab element that was selected.
- When the selected tab changes, the only thing that needs to be updated is the value of that attribute so that it targets the newly selected tab.
- If this approach is followed correctly, the browser and screen reader will perceive the tab referenced by aria-activedescendant as being focused.

Step Four: Add tags to manage the activation behavior (what happens when a tab is selected).

Clicking on a tab immediately updates just the relevant part of the page through AJAX. The tablist and the rest of the page stays the same. The tabs are used to activate and dynamically load the tabpanel content, but the focus remains on the actual tab, when it is navigated with the keyboard (e.g.,arrow key). The tablist doesn't lose focus, the way it does with a traditional HTML tablist, that refreshes the entire page. The mouse & keyboard behavior are the same, which is similar to how a tablist works in a desktop environment.

Code Snippet example:

Below is a finished code snippet example of an ARIA tablist with a roving tabIndex, and where AJAX refreshes the relevant tabpanel content only, not the entire page:

```
 <ul role=”tablist” title=”My Animals”>
     <li role=”presentation”>
         <a tabindex=”0” href=”#”  role=”tab” aria-selected=”true”
aria-controls=”tabpanel1”>Dogs</a>
     </li>
     <li role=”presentation”>
         <a tabindex=”-1” href=”#”  role=”tab” aria-selected=”false”
aria-controls=”tabpanel2”>Cats</a>
     </li>
     <li role=”presentation”>
         <a tabindex=”-1” href=”#” role=”tab” aria-selected=”false”
aria-controls=”tabpanel3”>Birds</a>
     </li>
 </ul>
 <div role=”tabpanel” aria-labelledby=”tab1” aria-expanded=”true” aria-hidden=”false”>
     <h3>My Dogs</h3>
     …
 </div>
 <div role=”tabpanel” aria-labelledby=”tab2” aria-expanded=”false” aria-hidden=”true”>
     <h3>My Cats</h3>
     …
 </div>
 <div role=”tabpanel” aria-labelledby=”tab3” aria-expanded=”false” aria-hidden=”true”>
     <h3>My Birds</h3>
     …
 </div>
```

Alternatively, below is a finished code snippet example of an ARIA tablist with ARIA-ActiveDescendant, instead of a Roving Tabindex: (also assumes AJAX refreshes the relevant tabpanel content only, not the entire page - same perceived behavior as the Roving Tabindex):

```
 <ul role="tablist" title=”My Animals” tabindex="0" aria-activedescendant="tab1">
     <li role="presentation">
         <a id="tab1" href="#" role="tab” aria-selected="true”
aria-controls=”tabpanel1" >Dogs</a>
     </li>
     <li role=”presentation”>
         <a id="tab2” href=”#”  role=”tab” aria-selected=”false”
aria-controls=”tabpanel2”>Cats</a>
     </li>
     <li role=”presentation”>
         <a id="tab3” href=”#” role=”tab” aria-selected=”false”
aria-controls=”tabpanel3” >Birds</a>
     </li>
 </ul>
 <div role=”tabpanel” aria-labelledby=”tab1”
    <h3>My Dogs</h3>
     …
 </div>
 <div role=”tabpanel” aria-labelledby=”tab2
    <h3>My Cats</h3>
     …
 </div>
 <div role=”tabpanel” aria-labelledby=”tab3”
    <h3>My Birds</h3>
     …
 </div>
```

Performance

If applicable, list expectations for performance (optimal and worst cases would be fine, give time in seconds):

References

Thanks go to Hans Hillen from The Paciello Group for consultations on this code structure.

Requirements Listing

List all requirements (individual verifiable statements) that indicate whether the work for this item has been complete. If there are requirements that are not essential to the functionality but would be nice to have if time allows, enter those under 'Secondary':

Primary:

1. **For tabs that are intended to refresh the entire page, use the standard html tablist structure (example, Rice Kuali portal tabs & any application tabs that should refresh entire page):**
    a. Place a hidden heading before the tab list (placed offscreen via CSS).
    b. Encode each tab as a list item in the tab group's unordered list (<ul>).
    c. Add a head refererence (<a>) element to each of the list items (<li>).
    d. Add the tab role and state information to each of the list items (tabs), using hidden text (hidden spans,)
    e. Add the content that will be associated with each tab, below the list, in divs.
    f. Add code to manage the selection state of the tabs (add the 'selected' option to the head reference <a> element of the currently selected tab's list item, and delete it from the formerly selected tab's list item).
    g. Add code to display only the div (content) of the currently selected tab (delete the 'class=hidden' option from the div of the currently selected tab's div, and ensure it is added to the divs of the tabs that are NOT currently selected).
    h. For more details, see code snippet in the tab example one in the *Mocks and Diagrams* section above.

2. **For tabs that are intended to refresh only a portion of a page, for example, only a section of a form, use the ARIA tablist structure:**

- (For example, in the *"KRAD Testing - KS L&F - UIF Components (kitchen sink) - Other Fields page"* (scroll down) and in the *"K RAD Testing – KNS L&F - Fiscal OfficerInquiry 2"* form)

1. a. First, start with the standard html tablist structure.
    i. See the details documented above.
    ii. However, step 1.e. is optional - you don't have to add the content that will be associated with each tab, below the list, in divs. Instead, you can use AJAX to bring in the relevant content, depending on which tab is active. This is the last step below (step d).
   b. Second, add ARIA tags to the unordered list tag (<ul>), and to the list items (<li>), and to the divs that follow the tablist:
    i. On the unordered list tag (<ul>), add Role="tablist" and give the tablist a title: role="tablist" title="title goes here">
    ii. On the list items (<li>), add role="presentation"
    iii. On the <a> element inside the list's items (<li>), define these attributes:
        1. Role="tab"
        2. Aria-selected="true/false" (set to true for the tab that is currently selected, false otherwise)
        3. Aria-controls="<id of corresponding tabpanel>"
    iv. On the divs following the tablist, add:
        1. Role="tabpanel"
        2. Aria-labelledby="<id of associated tab element>"
        3. Aria-expanded="true / false" (depending on whether the tab panel is currently expanded or not - it will be set to true by code logic when the tab is selected)
        4. Aria-hidden="true / false" (depending on whether the tab panel is currently visible - it will be set to false by code logic when the tab is selected)
   c. Third, add a roving tab index (html) and code logic to manage the tab order and keyboard focus (jscript):
    i. Give each tab in the tab list a "tabindex" value.
    ii. Whenever the selected tab is changed (e.g. by using the arrow keys or clicking on it with the mouse), make sure scripting moves focus to that tab, and updates the tabindex values to reflect the new situation (the tab that used to have tabindex="0" now gets a tabindex value of "-1", and the newly selected tab gets tabindex="0"). With this approach it's very important to ensure there is always at least one tab with tabindex="0", otherwise the entire tablist would become unreachable by keyboard.
        1. For the tab that is currently selected, set the tabindex value to "0"
        2. Make sure all the other tabs have a tabindex value of "-1" (i.e. they are skipped in the tab order but still focusable).
   d. Fourth (and last), add logic to manage the activation behavior (to manage what happens when a tab is selected), through an AJAX call to refresh the relevant tabpanel content only, not the entire page. See these details:
    i. The tabs are used to activate and dynamically load the tabpanel content. The content can be brought in through AJAX (the content for all tabs doesn't have to be placed in divs below the tablist in the HTML, though it can be).
    ii. Clicking on a tab should immediately update just the relevant part of the page, through AJAX.
    iii. The tablist and the rest of the page stays the same.
    iv. The tablist doesn't lose focus the way it does with a traditional HTML tablist that refreshes the entire page.
    v. The tabs are used to activate and dynamically load the tabpanel content, but the focus remains on the actual tab when it is navigated with the keyboard (e.g.,arrow key).
    vi. The mouse & keyboard behavior are the same, which is similar to how a tablist works in a desktop environment.

Secondary:

1. item
2. item

Dependencies

List any functional or technical work that must be completed before work on this item can begin:

1. item

Issues

List any issues that need to be resolved before work on this item can begin:

Functional:

1. item
2. item

Technical:

1. item
2. item

QA or Regression Testing Plan

List steps needed to test the basic functionality of this update, enhancement, bug fix

1. test/steps
2. test/steps

Checkoff

Functional Analysis Complete? **No (completed by SME)**

Needs Review by KAI? **No (completed by SME)**

Technical Analysis Complete? **No (completed by DM)**

Needs Review by KTI? **No (completed by DM)**

Estimate: **30 hours (completed by DM)**

Technical Design: Link Here **(completed by DM)**

Jira: KULRICE-5514 and KULRICE-5423

Final Documentation: Link Here **(completed by DM)**

- Added to QA: **No (completed by SME)**
-