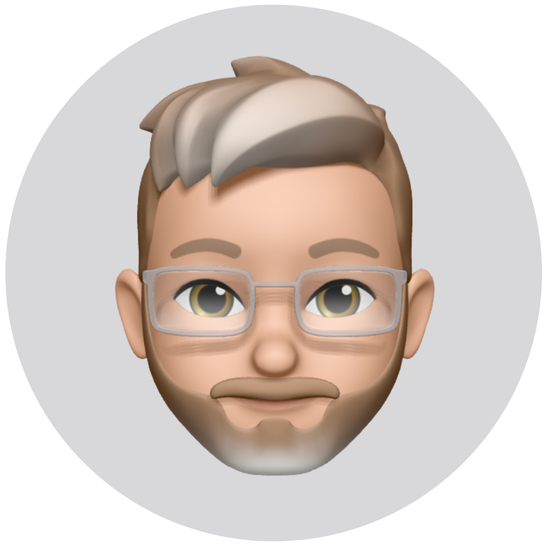
Introduction to ARIA

The University of Chicago
Center for Digital Accessibility

Hello!



Jack Auses, CPACC

Accessible Web Technology Specialist

UChicago Center for Digital Accessibility

jauses@uchicago.edu

What is ARIA?

What is ARIA?

Accessible Rich Internet Applications (ARIA) is an accessibility enhancing suite of web standards.

WebAIM: Introduction to ARIA - Accessible Rich Internet Applications

Why use ARIA?

When written semantically, much of HTML is accessible. However, HTML, JavaScript, and CSS do not natively include all the features required to make websites accessible to people who use screen readers or who rely on keyboard navigation.

ARIA is a way to address these deficiencies.

Only use ARIA when absolutely necessary

Ideally, native HTML should be used to provide the semantics required by screen readers. Sometimes this isn't possible, either because you have limited control over the code or are creating something complex that doesn't map to native HTML elements.

In such cases, ARIA can be a valuable tool.

No ARIA is better than bad ARIA

Functionally, ARIA *roles*, *states*, and *properties* are like CSS for assistive technologies. For screen reader users, ARIA influences the accessibility tree by controlling the audible "rendering" of their non-visual experience. Incorrect ARIA misrepresents visual experiences, with potentially devastating effects for the non-visual experience.

No ARIA is better than Bad ARIA - WAI-ARIA Authoring Practices 1.1

Five rules of ARIA use


The first rule of ARIA is:
You do not use ARIA

Rule #1: Use native HTML when possible

HTML is the foundation of web accessibility.

ARIA should not be used if native HTML can provide sufficient structure and semantics! When used incorrectly, ARIA can introduce significant accessibility barriers.

WebAIM: Introduction to ARIA - Rules of ARIA Use

Rule #2: Don’t change native HTML semantics, unless you really have to

Most HTML elements have default semantics that are conveyed to screen reader users. When necessary, ARIA can override and change those semantics.

WebAIM: Introduction to ARIA - Rules of ARIA Use

Rule #3: All interactive elements must be usable with the keyboard

ARIA design patterns define standard keyboard interactions for custom widgets and controls such as tabs, accordions, and other stateful UI components. This allows everyone to use the widget with a keyboard, and ensures that instructions provided by screen readers align with the actual functionality in the page.

Design Patterns and Widgets - WAI-ARIA Authoring Practices 1.2

Rule #4: Interactive elements must be properly semantic and visible

Any element that is keyboard focusable must have proper semantics either via native HTML or ARIA roles so that it is correctly identified as a link, button, form control, etc. Similarly, interactive elements must be visible.

WebAIM: Introduction to ARIA - Rules of ARIA Use

Rule #5: All interactive elements must have an accessible name

Text describing an interactive element *must* be presented to screen reader users when the element is encountered—this is called an “accessible name”. If native HTML methods aren’t available, ARIA can be used to define accessible names.

WebAIM: Decoding Label and Name for Accessibility

Main features of ARIA: roles, properties, and states

Roles

Roles define what an element is or does

Many of these are so-called landmark roles, which replicate the semantics of HTML5 sectioning elements, such as `role="navigation"` (`<nav>`) or `role="contentinfo"` (`<footer>`), but there are others that describe page components, such as `role="search"`, `role="tablist"`, `role="tab"`, without a native HTML counterpart.

Three main features of ARIA | MDN

Properties

Properties are used to give elements extra semantics

For example, `aria-required="true"` specifies that a form input needs to be filled in order to be valid, and `aria-labelledby="some-id"` enables authors to reference other elements on the page to define an accessible name, which is not possible using the HTML `<label>` tag.

Three main features of ARIA | MDN

States

States define the current conditions of elements

For example, `aria-expanded="true"` and `aria-expanded="false"` convey to a screen reader that a collapsible element, like an accordion, is currently either active/expanded/visible or inactive/collapsed/hidden.

Three main features of ARIA | MDN

Examples: when should you use ARIA?

Defining landmarks

ARIA's `role` attribute can define landmarks that either replicate the semantics of HTML5 elements such as `nav` and `header`, or go beyond HTML5 to provide semantics to different functional areas, e.g. `search`, `tablist`, `tab`, `listbox`, etc.

Native HTML Landmarks

```
<body>
  <header>
    <h1>...</h1>
    <nav>
      <ul>...</ul>
    </nav>
  </header>
  <main>
    <article>...</article>
    <nav>
      <ul>...</ul>
    </nav>
  </main>
  <footer>...</footer>
</body>
```

Using ARIA roles to define landmarks

When native HTML can’t be used, or if retrofitting old code, ARIA roles can be added to provide landmark cues for assistive technologies.

ARIA landmark roles

```
<body>
   <div role="banner">
      <h1>...</h1>
      <div role="navigation">
         <ul>...</ul>
      </div>
   </div>
   <div role="main">
      <div role="article">...</div>
      <div role="navigation">
         <ul>...</ul>
      </div>
   </div>
   <div role="contentinfo">...</div>
</body>
```

When an ARIA landmark appears more than once per page

When a type of landmark appears more than once on a page, we need to differentiate those landmarks for screen readers using `aria-label`. This is most common with main and sub navigation, but can also occur when there are multiple custom widgets—like accordions—on a page.

Multiple nav landmarks

```
<body>
   <header>
      <h1>...</h1>
      <nav aria-label="primary">
         <ul>...</ul>
      </nav>
   </header>

    <main>
      <article>...</article>
      <nav aria-label="secondary">
         <ul>...</ul>
      </nav>
   </main>

    <footer>...</footer>
</body>
```

Announcing dynamic content updates

By default, screen readers have difficulty announcing dynamic content updates. We can use `aria-live` to inform screen reader users when an area of content is updated, e.g. via `XMLHttpRequest`, or DOM APIs.

ARIA live regions | MDN

aria-live

If you have JavaScript or an API that dynamically changes the contents inside a region after page load, adding the `aria-live` property will instruct a screen reader to announce the content as it is updated.

```
<section aria-live="polite">...</section>
```

ARIA live regions | MDN

aria-live options

The `aria-live` property has three options:

1. **`off`**: Default. Updates are not announced.
2. **`polite`**: Updates are announced only if the user is idle.
3. **`assertive`**: Updates are announced to the user as soon as possible.

ARIA live regions | MDN

aria-live and screen readers

With `aria-live` a screen reader's default behavior will be to read out only the bit of text that updates, ignoring any surrounding text within the region.

ARIA live regions | MDN

aria-atomic

The `aria-atomic` property can be used in conjunction with `aria-live` to set whether or not the screen reader should announce everything within the live region as a whole, even if only part of the region changes. A value of `true` will announce everything, along with the region's label if one is defined.

```
<section aria-live="polite" aria-atomic="true">...</section>
```

ARIA live regions | MDN

Creating accessible labels and descriptions

Ideally, elements should be labeled with visible, clearly written text using native HTML methods. This approach is the simplest, easiest to maintain, and helpful to people with cognitive disabilities. However, ARIA is the only way to add accessible labels or descriptions to an element if native accessible text isn’t available.

Use ARIA when HTML labels are unavailable

Native HTML facilitates associations that support accessibility —`<label>` for form inputs, `<caption>` for data table descriptions, `<th>` for row and column headers in data tables, etc.

When HTML cannot create the necessary associations, ARIA can be used.

aria-label and aria-labelledby

ARIA provides several mechanisms for adding labels and descriptions to elements with attributes such as `aria-label` and `aria-labelledby`.

aria-labelledby

The `aria-labelledby` property programmatically associates an element with text that functions as a descriptive label, but isn't wrapped in a semantic HTML element such as `<label>`. With `aria-labelledby`, the ARIA property references the id (or, in some cases, multiple ids) of the element(s) acting as its label.

aria-labelledby example

In this example, the <section> element is labeled and identified by the text “Search Filters” within the <h2>:

```
<section aria-labelledby="filter-heading">
  <h2 id="filter-heading">Search Filters</h2>
  ...
</section>
```

aria-label

The `aria-label` property allows you to apply the label text directly in the attribute value.

aria-label example

In this example, even though the `<h2>` text is “Filters”, this `<section>` will be labeled by the more specific `aria-label` value of “Search Filters”:

```
<section aria-label="Search Filters">
  <h2>Filters</h2>
  ...
</section>
```

Constraints of ARIA labels

Considerations

Using ARIA to define labels comes with some constraints and warnings.

ARIA labels only work on labelable elements

In order to be assigned an ARIA label, an element must be *labelable*—either a link, button, or form control, or having specific HTML or ARIA semantics. Many elements are not labelable—such as `<p>`, `<div>`, and `<span>`—unless assigned an appropriate ARIA role.

Short note on aria-label, aria-labelledby, and aria-describedby

ARIA labels override HTML

ARIA labels override HTML elements' default text and accessible names.

If a form input already has an associated `<label>` and an `aria-label` or `aria-labelledby`, the `<label>` will not be read. Do not override a `<label>` with ARIA unless additional context is required for a screen reader user to understand it.

ARIA labels and visible text

The Web Content Accessibility Guidelines (WCAG) requires that the visible text label for an element be included within its accessible name (which is read by a screen reader). When using ARIA labels, ensure consistency between what sighted users see and what screen reader users hear.

Understanding Success Criterion 2.5.3: Label in Name

Vague link text

In the following example, the link text “Read More” is too vague and—if not rewritten—requires an `aria-label`:

```
<a href="...">Read more</a>
```

Fixing vague link text with aria-label

Incorrect:

```
<a href="..." aria-label="About UChicago">Read More</a>
```

Correct:

```
<a href="..." aria-label="Read more about UChicago">Read more</a>
```

Test with a screen reader

The best way to ensure that ARIA labels and descriptions have been implemented correctly is to listen with a screen reader. You can also inspect the accessible name and description of an element using your browser's developer tools.

WebAIM: Testing with Screen Readers - Questions and Answers

Enhancing the a11y of non-semantic controls and custom widgets

When using HTML, CSS, and JavaScript to create a complex widget or modify a native control, accessibility can suffer—custom controls might not be keyboard accessible and screen reader users will find it difficult to determine what the feature does if there are no semantics or other clues.

ARIA can extend the a11y of HTML

In these situations, ARIA can help fill in the blanks with a combination of roles like `button`, `listbox`, or `tablist`; properties like `aria-required`; and states such as `aria-expanded` to provide further information as to the element's purpose.

Enhancing keyboard a11y

One of the key strengths of HTML with respect to accessibility is the built-in keyboard support of elements such as buttons, form controls, and links. Generally, you can use the tab key to move between controls, the enter/return key to select or activate controls, and other keys such as the up and down arrows to move between options in a `<select>` box.

Enhancing keyboard a11y

However, sometimes you will end up having to write code that either uses non-semantic elements for buttons and other controls, or uses focusable controls for not quite the right purpose. You might be trying to fix some bad code you've inherited, or you might be building a complex widget that requires it.

tabindex

In terms of making non-focusable code focusable, ARIA extends the `tabindex` attribute with some new values:

tabindex="0"

tabindex="0"

This value allows elements that are not normally focusable (such as `<div>`, `<span>`, `<p>`, and `<a>` with no `href`) to receive focus.

This is the most useful value of `tabindex`.

tabindex="-1"

tabindex="-1"

Removes interactive elements from the default tab order. In most cases, this is not desirable. However, if added to a non-interactive element, `tabindex="-1"` allows that element to receive programmatic focus with `focus()` scripting.

This can be useful for elements that should not be navigated to directly using the tab key, but need to have keyboard focus set to them, such as a modal dialog window that should receive focus when it is opened.

Retrofitting a custom button

Here we have given some `<div>`s that are supposed to function like `<button>`s the ability to be focused by adding the `tabindex="0"` attribute:

```
<div role="button" tabindex="0">First Button</div>
<div role="button" tabindex="0">Second Button</div>
<div role="button" tabindex="0">Third Button</div>
```

Semantic HTML button

Ideally you would use <button> tags instead of any ARIA:

```
<button>First Button</button>
<button>Second Button</button>
<button>Third Button</button>
```

ARIA roles that go beyond native HTML

There are a whole host of other ARIA roles that can identify common UI features that go beyond what's available in standard HTML, for example `combobox`, `slider`, `tabpanel`, `tree`.

Deque University Code Library

Tabs

Common tab structure

The following is simple tab pattern markup with a `<ul>` containing the tabs a user would click and the corresponding tab panels that become visible or invisible depending on which tab is selected. The tab functionality is controlled by JavaScript. Nothing is conveyed to a screen reader about the purpose of this widget, nor are the tabs keyboard accessible.

Basic tab pattern

```
<div class="my-tabs">
  <ul>
    <li id="tab-one">Tab 1</li>
    <li id="tab-two">Tab 2</li>
  </ul>
  <div class="tabpanel-group">
    <div id="panel-one">...</div>
    <div id="panel-two">...</div>
  </div>
</div>
```

Adding roles to a tab pattern

Adding roles

A first step to improving the accessibility of this tab widget is to define the `tablist`, `tab`, and `tabpanel` roles for assistive technology using ARIA.

Adding roles to a tab pattern example

```
<div id="my-tabs">
  <ul role="tablist">
    <li id="tab-one" role="tab">Tab 1</li>
    <li id="tab-two" role="tab">Tab 2</li>
  </ul>
  <div class="tabpanel-group">
    <div id="panel-one" role="tabpanel">...</div>
    <div id="panel-two" role="tabpanel">...</div>
  </div>
</div>
```

Adding properties to a tab pattern

Adding properties

We now move on to adding ARIA properties to the elements in the widget so the `tabs` are programmatically related to their corresponding `tabpanels`. We will also make the tabs keyboard accessible by using the `tabindex` attribute.

Adding properties to a tab pattern example

```
<div id="my-tabs">
  <ul role="tablist">
    <li id="tab-one" role="tab" aria-controls="panel-one" tabindex="0">Tab 1</li>
    <li id="tab-two" role="tab" aria-controls="panel-two" tabindex="0">Tab 2</li>
  </ul>
<div class="tabpanel-group">
    <div id="panel-one" role="tabpanel">...</div>
    <div id="panel-two" role="tabpanel">...</div>
  </div>
</div>
```

Adding states to a tab pattern

Adding states

Finally, we will use ARIA to define the active/inactive states of the `tabs` and `tabpanels` with `aria-selected` and `aria-hidden` attributes. These states would be toggled via JavaScript.

In the following example, Tab 1 has been clicked and is active.

Adding states to a tab pattern example

```
<div id="my-tabs">
  <ul role="tablist">
    <li id="tab-one" role="tab" aria-controls="panel-one" tabindex="0" aria-selected="true">Tab 1</li>
    <li id="tab-two" role="tab" aria-controls="panel-two" tabindex="0" aria-selected="false">Tab 2</li>
  </ul>
<div class="tabpanel-group">
    <div id="panel-one" role="tabpanel" aria-hidden="false">...</div>
    <div id="panel-two" role="tabpanel" aria-hidden="true">...</div>
  </div>
</div>
```

Final thoughts

ARIA can extend the accessibility of native HTML

ARIA provides methods to convey information to assistive technology that isn’t possible with native HTML, such as defining a place where dynamic content is loaded or the expanded/collapsed state of a custom widget.

ARIA can also improve bad HTML

However, ARIA is often used as a polyfill for non-semantic HTML or bad information architecture, such as defining landmark roles when native HTML tags exist or correcting for vague link text like “click here” or “read more”.

Use ARIA only when necessary

Strive to leverage well-structured and semantic native HTML in your code and meaningful accessible text in your content as much as possible and **only use ARIA when absolutely necessary**.

Questions?

Contact the CDA

The **Center for Digital Accessibility** is here to provide digital accessibility resources for campus. Please contact us at digitalaccessibility@uchicago.edu.

Additional reference materials

Slide 1

- WebAIM: Introduction to ARIA - Accessible Rich Internet Applications
- Introduction to ARIA | Web Fundamentals | Google Developers
- ARIA - Accessibility | MDN

Slide 7

- No ARIA is better than Bad ARIA - WAI-ARIA Authoring Practices 1.2

Slides 10–11, 13

- WebAIM: Introduction to ARIA - Rules of ARIA Use

Slide 12

- Design Patterns and Widgets - WAI-ARIA Authoring Practices 1.2

Slide 14

- WebAIM: Decoding Label and Name for Accessibility

Additional reference materials

Slides 16–18

- Three main features of ARIA | MDN

Slides 26–30

- ARIA live regions | MDN

Slide 39

- Short note on aria-label, aria-labelledby, and aria-describedby

Slide 41

- Understanding Success Criterion 2.5.3: Label in Name

Slide 44

- WebAIM: Testing with Screen Readers - Questions and Answers

Slide 54

- Deque University Code Library

Thank you!

digitalaccessibility.uchicago.edu