ARIA

INTRODUCTION

Why is ARIA needed?

The HTML markup language has a **very limited set of interface controls** and interactions.

As the demand for rich interactions has increased, **JavaScript has become our saviour**!

JavaScript provides us with **many things** including:

dynamic interactions:
such as drag and drop, resizing, hide and show, open and shut, switch views etc.

widgets and components:
such as modals, in-page tabs, button drop-downs, date pickers, page loaders, sliders and much more.

However, many of these dynamic interactions and widgets are **problematic for Assistive Technologies**.

Issues with dynamic content

Assistive Technologies may not be aware that **content that has been dynamically updated** after the initial page has loaded.

This means that this dynamically added content may be **totally inaccessible** to some audiences.

Issues understanding the purpose

Assistive Technologies **may not be aware of the purpose of non-native widgets** - such as dropdown buttons, in-page tabs, accordions etc.

This means that **the purpose of a widget may be confusing** for some users.

This is especially true for widgets that use incorrect HTML elements and **rely on JavaScript in order to function**.

In these cases, the entire widget may **totally inaccessible** for some users.

Issues understanding the state

Assistive Technologies **may not be aware of the state of a widget** - such as an item that has been selected or checked.

This means that users **may not be able to tell whether they have selected or checked an option** within a widget.

This is where **ARIA can come to the rescue**!

What is ARIA?

“WAI-ARIA is a technical specification that provides a framework to **improve the accessibility and interoperability** of web content and applications.”

https://www.w3.org/WAI/intro/aria

WAI:
Web Accessibility Initiative

ARIA:
Accessible Rich Internet Applications

WAI ARIA Releases

WAI-ARIA 1.0 was published as a completed **W3C Recommendation** on 20 March 2014.

https://www.w3.org/TR/wai-aria/

WAI-ARIA 1.1 was published as a completed **W3C Recommendation** on 14 December 2017.

https://www.w3.org/TR/wai-aria-1.1/

WAI-ARIA 1.2 was published as a completed **W3C Recommendation** on 18 December 2018.

https://www.w3.org/TR/wai-aria-1.2/

ARIA specifications and documents

WAI-ARIA technical specification

Primarily for developers of Web browsers, assistive technologies, and accessibility evaluation tools.

https://www.w3.org/TR/wai-aria/

WAI-ARIA User Agent Implementation Guide

Describes how browsers and other user agents should support WAI-ARIA

https://www.w3.org/TR/wai-aria-implementation/

WAI-ARIA Authoring Practices

Describes how Web developers can develop accessible rich internet applications using ARIA, with detailed advice and examples.

https://www.w3.org/TR/wai-aria-practices/

WAI-ARIA Primer

Introduces developers to the accessibility problems that WAI-ARIA is intended to solve.

https://www.w3.org/TR/wai-aria-primer/

WAI-ARIA Roadmap
defines the path to make rich Web content accessible, including steps already taken, remaining future steps, and a timeline.

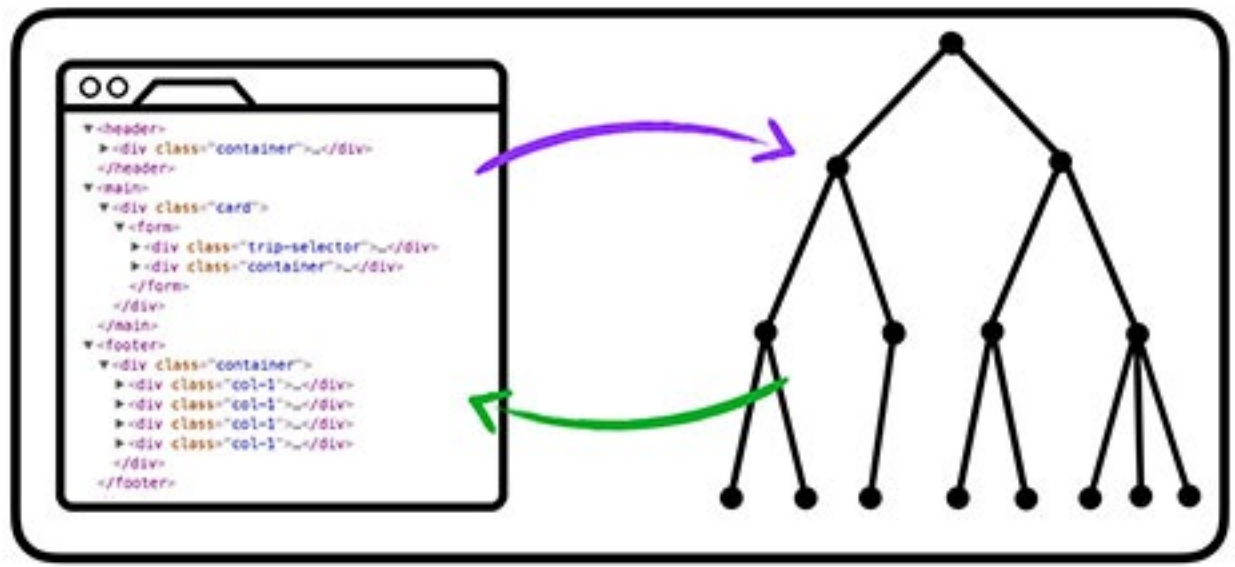
https://www.w3.org/TR/wai-aria-roadmap/

How does ARIA work?

ARIA uses a range of custom HTML attributes to convey **additional meaning** to HTML elements for assistive technologies only.

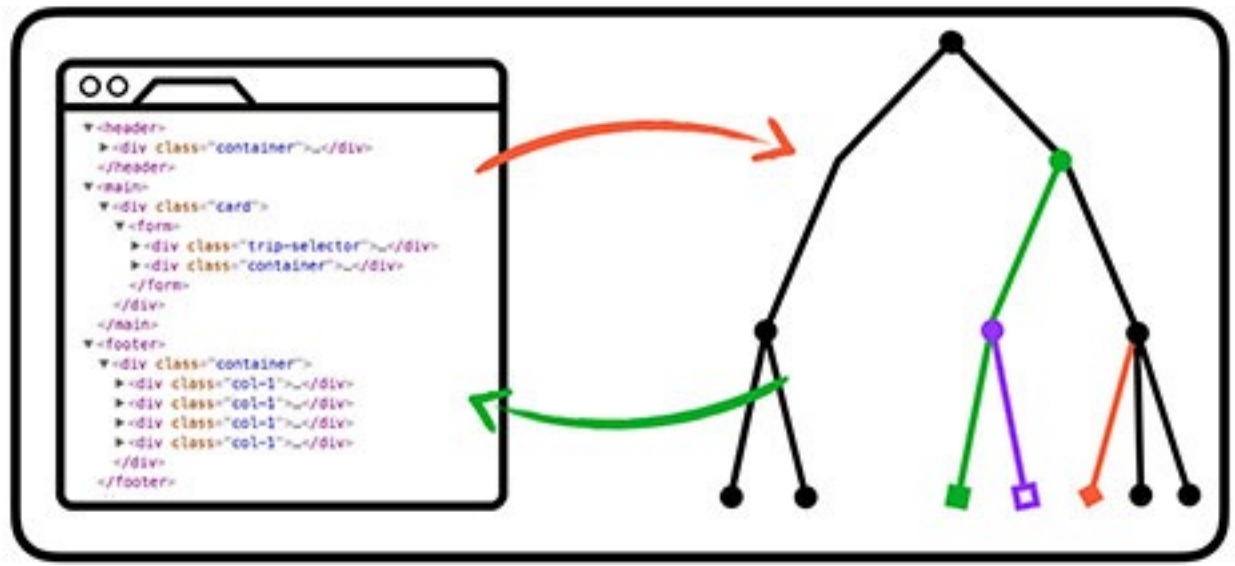
```
<ul role="menu">
</ul>

<input aria-checked="true" type="radio">

<button aria-label="Close application">
  Close
</button>
```

These HTML attributes **change and augment the Accessibility Tree**.


▼<header>
▶ …
</header>
▼<main>
▼
▼<form>
▶ …
▶ …
</form>

</main>
▼<footer>
▼
▶ …
▶ …
▶ …
▶ …

</footer>
DOM
accessibility tree


▼<header>
▶ …
</header>
▼<main>
▼
▼<form>
▶ …
▶ …
</form>

</main>
▼<footer>
▼
▶ …
▶ …
▶ …
▶ …

</footer>
DOM
+
ARIA
accessibility
tree

ARIA allows authors to **adjust the Accessibility Tree** to do the following:

1. Add semantics
2. Modify existing semantics
3. Provide extra labelling
4. Provide extra descriptions
5. Establish relationships
6. Inform ATs of different states
7. Inform ATs of live updates

ARIA only modifies the Accessibility Tree. **It does not do any of the following**:

1. Modify an element’s appearance
2. Modify the element’s behaviour
3. Add focusability
4. Add keyboard event handlers

Roles, States and Properties

ARIA attributes are broken down into **roles, states and properties**.

Role attributes allow us to inform Assistive Technologies what type of widget it is.

Is it a menu, slider, progress bar? Does it provide the structure of a web page?

```
<!-- defining a widget -->
<ul role="menu">
    ...
</ul>

<!-- defining some aspect of the page structure -->
<main role="main">
    ...
</main>
```

State attributes allow us to inform Assistive Technologies what the current state of the widget is.

Is it checked, disabled, something else?

States (such as `aria-checked`) may change frequently depending on user interaction.

```
<!-- defining different states-->
<input aria-disabled="true" type="text">

<input aria-checked="true" type="radio">
```

Property attributes allow us to inform Assistive Technologies the purpose of the element, or whether it has a relationship to other elements.

What is it? Does it interact with other elements?

Properties (such as `aria-labelledby`) very rarely change.

```
<!-- describe a relationship -->
<input aria-describedby="format"
type="text" name="a" id="a">
<span id="format">
  (must be mm/dd/yyyy)
</span>

<!-- providing additional labelling -->
<button aria-label="Close and return to application">
  Close
</button>
```

State vs Property?

The terms “states” and “properties” refer to similar features. However, there are some **subtle differences** in their meaning.

To avoid confusion, the WAI-ARIA specifications refer to “states” and “properties” simply as **“attributes”** whenever possible.

In reality, all of ARIA is just **HTML attributes** broken into the following categories:

ARIA Attributes

Roles (role)

- Abstract
- Widget
- Document Structure
- Landmark
- Live Region
- Window

States and Properties (aria-*)

- Widget
- Live Region
- Drag-and-Drop
- Relationship
- Global

ARIA support?

Browsers that support ARIA:

Firefox 3+
Internet Explorer 8+
Safari 4+ (Mac)
Chrome 17+

Assistive Technologies that support ARIA:

JAWS 8+ (Win)
Windows Eyes 5.5+ (Win)
ZoomText
VoiceOver (OS X/iOS)
NVDA (Win)
ORCA (Linux)

Keep in mind that **“support” is a general term**. Each browser/Assistive Technology has it’s own minor quirks and inconsistencies.

Also, the ARIA specification is constantly evolving, so the concept of “support” is a shifting landscape.

When in doubt, test!

Exercise 1:
Fixing a fake checkbox

Open **exercise01-fake-checkbox/start.html** in a browser and also in a text editor.

Review your work against **exercise01-fake-checkbox/finished.html**

We'll use an example where a developer may use a `<div>` element **instead of a checkbox**.

(Let's not get bogged down discussing "why" anyone would do such a thing at this point. It can and does happen)

JavaScript and CSS have been used to **make the element look and operate like a checkbox** - at least to sighted, mouse users.

Native checkbox

- [x] Apples
- [ ] Bananas

Fake checkbox

- [x] Apples
- [ ] Bananas

```
<!-- Native checkbox -->
<form action="#">

  <div>
    <input type="checkbox" id="e1" checked>
    <label for="e1">Apples</label>
  </div>

  <div>
    <input type="checkbox" id="e2">
    <label for="e2">Bananas</label>
  </div>
```

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div class="checkbox checked">Apples</div>

  <div class="checkbox">Bananas</div>

</div>
```

However, there is no semantic meaning associated with the `<div>` element, so Assistive Technologies have **no way of understanding it's purpose** and then conveying this to the user.

Assuming we **cannot change** the `<div>` element, how could we (1) add some functionality for keyboard users and (2) add some additional semantic meaning for Assistive Technologies?

Step 1:

Adding `tabindex` with a value of `"0"` makes the element able to receive focus without changing the overall tab order of the page.

(This has nothing to do with ARIA but it is very important for keyboard only users.)

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div tabindex="0" class="checkbox checked">Apples</
div>

  <div tabindex="0" class="checkbox">Bananas</div>

</div>
```

Step 2:

Adding `role="checkbox"` makes sure the element is announced as a checkbox to Assistive Technologies.

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div tabindex="0" role="checkbox" class="checkbox
checked">Apples</div>

  <div tabindex="0" role="checkbox"
class="checkbox">Bananas</div>

</div>
```

Step 3:

Adding `aria-checked="true"` to the first fake checkbox means that it's checked state will be announced to Assistive Technologies.

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div tabindex="0" role="checkbox" aria-
checked="true" class="checkbox checked">Apples</div>

  <div tabindex="0" role="checkbox"
class="checkbox">Bananas</div>

</div>
```

Keeping in mind that using a `<div>` here is very poor practice, we have now managed to **make the widget slightly more accessible**.

We used ARIA attributes to **change some nodes in the accessibility tree** so that they have a meaningful role and state.

More importantly, adding these ARIA attributes **has not changed anything** about the the appearance or on-screen behaviour of these elements.

Bad ARIA?

Over the last few years there has been a growing trend for developers to include ARIA attributes in **all sorts of areas in applications**.

Unfortunately, **ARIA attributes can be misused**, and this can lead to all sorts of problems for Assistive Technologies.

Issues with Redundancy

ARIA is sometimes used by “over-zealous” developers to provide additional ARIA attributes to native HTML elements **that already have accessible APIs**.

This can lead to **problems for Assistive Technologies**, such as the role of an element being announced more than once.

```
<!-- Do not do this -->
<input type="radio" role="radio">

<!-- This is preferred -->
<input type="radio">
```

```
<!-- Do not do this -->
<label for="name">Name</label>
<input id="name" type="text"
  required aria-required="true">

<!-- This is preferred -->
<label for="name">Name</label>
<input id="name" type="text" required>
```

Issues with being too verbose

There are times when developers use ARIA attributes to provide **detailed additional information** for Assistive Technologies.

This can sometimes lead to **excessive amounts of information** being presented to AT users.

```
<!-- Do not do this -->
<label for="email">Email</label>
<input id="email" type="email"
  aria-describedby="tip">
<span id="tip">Tip: Always include an AT
symbol as part of the email address so
that it is valid.</span>
```

Issues with copy and paste

This is where developers simple copy chunks of code from existing pattern libraries, **without understanding how the ARIA attributes in this code work**.

This can lead to problems such as `aria-label` attributes pointing to non-existent `ID` attributes, **so the ARIA does not work**.

```
<!-- Do not do this -->
<div role="dialog" aria-labelledby="modalLabel">
  <h5>Modal title</h5>
</div>

<!-- This is preferred -->
<div role="dialog" aria-labelledby="modalLabel">
  <h5 id="modalLabel">Modal title</h5>
</div>
```

Rules of using ARIA

Rule 1

If you can use a native HTML element or attribute with the semantics and behaviour you require already built in, **then do so**.

```
<!-- Do not do this -->
<div role="button">...</div>

<!-- This is preferred -->
<button>...</button>
```

Rule 2

Do not change native semantics,
unless you really have to.

```
<!-- Do not do this -->
<h2 role="tab">heading tab</h2>

<!-- This is preferred -->
<div role="tab">
  <h2>heading tab</h2>
</div>
```

Rule 3

All interactive ARIA controls **must be usable with the keyboard**.

If you create a widget that a user can click or tap or drag or drop or slide or scroll, a user **must also be able to navigate to the widget and perform an equivalent action** using the keyboard.

Rule 4

Do not use `role="presentation"` or `aria-hidden="true"` on **visible focusable elements**.

```
<!-- Do not do this -->
<button role="presentation">press me</button>
<button aria-hidden="true">press me</button>
```

Rule 5

All interactive elements **must have an accessible name**.

An interactive element **only has an accessible name** when its Accessibility API accessible name (or equivalent) property has a value.

```
<div role="region" aria-label="weather portlet">
  ...
</div>
```

Exercise 2: Providing extra context for buttons

Confusion about aria-label, aria-labelledby and aria-describedby

There are three different ARIA attributes that are **often confused**.

aria-labelledby

aria-describedby

aria-label

aria-labelledby

This is where one element is used to **provide a label or an accessible name** for another element, the widget element.

The `aria-labelledby` attribute **establishes the programmatic relationship** between the widget and the element providing the label.

The `aria-labelledby` attribute is **applied to the widget**, and the matching `ID` value is applied to the label element.

```
<!-- Widget -->
<div role="dialog" aria-labelledby="modalLabel">
  <!-- Widget label -->
  <h5 id="modalLabel">Modal title</h5>
</div>
```

aria-describedby

This is where one element is used to **provide an accessible description** for another element, the widget element.

The `aria-describedby` attribute **establishes a programmatic relationship** between the widget and the element providing the descriptive information.

The `aria-describedby` attribute is **applied to the widget**, and the matching `ID` value is applied to the label element.

```
<label for="a">Phone</label>
<!-- Widget -->
<input id="a" type="text" aria-describedby="i1">

<!-- Description -->
<p id="i1">Input instructions</p>
```

aria-label

The `aria-label` attribute is used to **provide an accessible name or label directly to an element**.

```
<button type="button" aria-label="Close and return to
account details">
```

The `aria-label` attribute is **for assistive technologies only**, and is not visible on screen.

Usage

The `aria-labelledby`, `aria-describedby` and `aria-label` attributes should **only be applied to specific types of elements**.

1. **Interactive** elements:

```
<!-- Interactive elements 1 -->
<a href="#"></a> (if href present)
<audio controls></audio>
<button></button>
<details></details>
<embed></details>
<iframe></iframe>
<img usemap="#a"> (if usemap present)
```

```
<!-- Interactive elements 2 -->
<input> if the type is not hidden
<label></label>
<menu type="toolbar"></menu> (if toolbar present)
<object usemap="#a"></object> (if usemap present)
<select></select>
<textarea></textarea>
<video controls></video> (if controls present)
```

2. Elements with **implicit landmark roles**:

```
<!-- Implicit landmark roles -->
<header></header>
<footer></footer>
<main></main>
<nav></nav>
<aside></aside>
<section></section>
<form></form>
```

3. Elements with **explicit roles** - where any `role` attribute is applied.

```
<!-- Some explicit roles -->
<div role="dialog"></div>
<div role="tab"></div>
<div role="tabpanel"></div>
```

If these three attributes are applied to any other types of elements, **they may not work across all browsers and assistive technology combinations**.

The problem

Imagine you have a modal window and in the top right corner there is a `<button>` element with an **“X” symbol**.

Or, in the middle of a banking application screen there is a `<button>` element with the words **“Delete”**.


Delete

These two `<button>` elements are meaningful for sighted users as there are **visual clues nearby** that provide additional context.

But what about a Screen Reader user who uses the `TAB` keystroke and focuses on either of these `<button>` elements **without any additional context**?

Ideally, we should provide these `<button>` elements with additional context so that Screen Reader users **can understand their purpose**.

“Close and return to Account details”

“Delete this Personal Savings Account Transaction”

Using aria-label to solve the problem

Open **exercise02-extra-context/start.html** in a browser and also in a text editor.

Review your work against **exercise02-extra-context/finished.html**

We can do this by **applying** the `aria-label` attribute to both of the `<button>` elements.

The `aria-label` value will provide a hidden label that is only available to Assistive Technologies. This value will **override the actual button content**.

```
<button type="button" aria-label="Close and return to
account details">
  x
</button>

<button type="button" aria-label="Delete this
Personal Savings Account Transaction">
  Delete
</button>
```