Understanding ARIA

ARIA (Accessible Rich Internet Applications) and the associated accessibility standards (WCAG) have always been hard to for me to parse from a standpoint of when to use as well as how do you use when you need to.

This post is my first step in understanding ARIA, WCAG, How it works and why it is important.

Definitions

The first step in understanding something is to work towardsd defining the terms we will use.

What is ARIA

ARIA (Accessible Rich Internet Applications), or more formally WAI-ARIA, is a set of specifications that aim to make web applications more accessible.

According to the WAI-ARIA Overview

> WAI-ARIA, the Accessible Rich Internet Applications Suite, defines a way to make Web content and Web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with HTML, JavaScript, and related technologies.

What is WCAG?

The Web Content Accessibility Guidelines cover techniques to make content more accessible to users with disabilities

According to the Web Content Accessibility Guidelines (WCAG) 2.1 recommendation:

> Web Content Accessibility Guidelines (WCAG) 2.1 covers a wide range of recommendations for making Web content more accessible. Following these guidelines will make content more

> accessible to a wider range of people with disabilities, including accommodations for blindness and low vision, deafness and hearing loss, limited movement, speech disabilities, photosensitivity, and combinations of these, and some accommodation for learning disabilities and cognitive limitations; but will not address every user need for people with these disabilities.

As I understand it, ARIA itself provides roles, states, properties and focus management that interact with assistive technologies tools and APIs. This will result in more accessible web applications.

WCAG provides guidelines, techniques and best practices for making content more accessible. This may include using ARIA roles, states and technologies where appropriate.

What is semantic markup and why it's important

A lot of times we'll hear about semantic markup and its importance on one hand and how you can create really odd or broken markup and it being valid HTML.

The example that still catches my attention. People say that this is valid HTML:

```
<html>
  <title>demo page</title>
  <h1>Demo Page Title</h1>
  <p>Content goes here</p>
</html>
```

It is not valid HTML. It relies on the backwards compatibility requirements of modern browsers regarding broken HTML content.

Chrome, for example, will insert the missing elements to make the page render correctly. This is how Chrome will show the page in DevTools:

```
<html>
```

```
  <head>
    <title>demo page</title>
  </head>
  <body>
    <h1>Demo Page Title</h1>
    <p>Content goes here</p>
  </body>
</html>
```

So this is the first reason why semantic important. Browsers will do the they can to fully render the page so it would work best if we were to write the full code for the elements we want without taking shortcuts.

A second, and related, issue with semantic HTML is to use the correct element for a given situation.

When you create semantic elements in your document the browser gives you affordances like:

- Search engines will consider its contents as important keywords to influence the page's search rankings (see SEO)
- Screen readers can use it as a signpost to help visually impaired users navigate a page
- Finding blocks of meaningful code is significantly easier than searching through endless divs with or without semantic or namespaced classes
- Suggests to the developer the type of data that will be populated
- Semantic naming mirrors proper custom element/component naming

Just by looking at the element below we know what it is and what it purpose on the page is.

```
<h1>Page title</h1>
```

You could also create your own element that will look like an h1 element but you loose all the advantages that the premade buttons get.

```
<span style="font-size: 32px; margin: 21px 0;">Not a top-level heading!</s
```

```
</span><!-- Or -->

<span class="h1-heading">Not a top-level heading!</span>
```

You should always use a predefined element where one is available. The `div` and span elements should only be used when there is no predefined element available.

- <article>
- <aside>
- <details>
- <figure>
- <figcaption>
- <footer>
- <header>
- <main>
- <mark>
- <nav>
- <section>
- <summary>
- <time>

Whenever you're looking at the structure of a page you should always look at where semantic elements would be the most appropriate.

> For more information see Semantic HTML5 Elements Explained

When is a button not a button

Most of the time we will see buttons coded like this:

```
<button>Save</button>
```

But we also see people coding buttons like this:

```
<div
  id="saveChanges">
```

```
  Save
</div>
```

This could be used as a button but, as it is, lacks a lot of affordances that we'd get from a standard `button` element.

Some of the things you'd have to fix to make this "button" more accessible.

Roles

`div` elements are neutral, they don't represent anything in particular so, if we want to assign a given role to a `div` element you need to be explicit about it.

```
<div
  id="saveChanges"
  role="button">
  Save
</div>
```

This role is dependent on the functionality that you want to emulate. There is a full list of ARIA roles in MDN.

Keyboard navigation and Focus Management

The `button` element can be tabbed into and can be activated by pressing the `enter` or `space` keys.

Since our div "button" is not an actual button then it's up to use to make sure that the `div` works the same as a native button.

tabindex

The tabindex attribute controls whether a user can tab through the element. It has three possible values:

- -1: prevents navigation through the element using the keyboard. The element can still be accessed programmatically
- 0: Allows the element to be navigated via the keyboard in document

- > 1: The element will be navigated after all elements with 0 value and those with lower positive values.
    - tabindex="4" is focused before tabindex="5" and tabindex="0", but after tabindex="3". If multiple elements share the same positive tabindex value, they are focused in document order
    - Do not use values greater than 0 in `tabindex`. They will make it harder for people using assistive technologies to navigate your documents

The button now looks like this:

```
<div
  id="saveChanges"
  role="button"
  tabindex="0">
  Save
</div>
```

event handling

Using `tabindex` allows users to tab into the document but that's not enough.

We can usually click on a button with a pointer device and activate it with either the `space` or `enter` keys. Since we're not using the native button element we need to handle both clicks and keyboard navigation events.

We are also futureproofing the code by handling as many buttons as there are on the page.

We capture a reference to the buttons using querySelectorAll

We use the spread syntax to create an array of our buttons and then feed it to the forEach method in which we add the necessary events to each button.

The `doSomething` function is where the code we want to execute will run. There function will get updated when we look at ARIA attributes.

```
const buttons = document.querySelectorAll(".button");
```

```
[...buttons].forEach((button) => {
  button.addEventListener("pointerdown", doSomething);
  button.addEventListener("keydown", (event) => {
    if (event.key == "Enter" || event.key == " ") {
      doSomething();
    }
  });
});

function doSomething() {
  alert("Button activation event, do something");
}
```

Aria States and Properties

In addition to the `aria-role` attribute we added earlier, there are other that we may want to add to our custom button.

If the button is a toggle button, the `aria-pressed` attribute tells assistive technology whether a button is pressed or not. Since we created a custom button, we need to explicitly set the attribute in the HTML element.

```
<div
  id="saveChanges"
  role="button"
  tabindex="0"
  aria-pressed="false">
  Save
</div>
```

We then modify the `doSomething` function to toggle the value of the property on interaction (pointer or keyboard-bassed).

1. set the default for the `aria-pressed` attribute to false.
2. We the attribute we created in an if/else statement
    1. If the `aria-pressed` is true we set it to false using setAttribute; otherwise we set it to true.

```
function doSomething(button) {
  let pressed = button.getAttribute('aria-pressed') === 'false';

  if (pressed) {
    button.setAttribute('aria-pressed', 'true');
  } else {
    button.setAttribute('aria-pressed', 'false')
  }
}
```

Styling

Because a `div` element has no style of its own and we want to make it look like a button, we have to do it explicitly.

The button has two selectors. The `.button` selector handles the default state for all elements with the class

```
.button {
  border: 3px solid limegreen;
  border-radius: 15px;
  width: 5em;
  padding: 0.25em;
  text-align: center;
}
```

`.button:focus` handles when a specific element with the class `.button` gets focus.

```
.button:focus {
  outline: none;
  box-shadow: 0 0 0 2px #006ae3;
}
```

Final result

The final, working code, can be seen in this Codepen:

This is one example of what we can do with one type of buttons. Other elements may have other accessibility requirements that need to be implemented in Javascript and CSS.

The ARIA Authoring Practices provides tools to create accessible widgets. The patterns section is particularly important when trying to figure out how to code our own widgets.

Links and Resources

No ARIA is better than bad ARIA : The article discusses discusses why not using ARIA is better than using ARIA incorrectly and how it can degrade the way assistive technology works on your pages.

Accessible Rich Internet Applications (WAI-ARIA) 1.1 : Describes the technical aspect of building Accessible Rich Internet Applications : Provides definitions of ARIA roles, states and properties, and focus management

WCAG 2 Overview : Introduces the Web Content Accessibility Guidelines (WCAG) 2.1 as the current version of accepted practices to create accessible content. : This is different from ARIA in that it doesn't directly define the structure of a document but it handles content, broadly defined as: : : Natural information

such as text, images, and sounds : : code or markup that defines structure, presentation, etc. : There is a WCAG 2.2 Candidate Recommendation that is expected to become a recommendation in December 2022

WCAG 3 Introduction : WCAG 3 is the next generation of the W3C accessibility guidelines : The release date is uncertain