Using ARIA

Accessibility is one of the most important aspects of development and one that we don’t pay as much attention as we should.

We will look at ARIA (Accessible Rich Internet Applications), what it is and how we can use it in our content to help improve the accessibility of web applications and pages. we will also discuss When it’s better to use native elements and DOM interactions rather than create our own.

we will also look at ARIA best practices and authoring guidelines. These are particularly suited to custom elements and components we create using Polymer or React-based applications.

As a last step we will look at a page using Voice Over, the screen reader bundled as part of MacOS. This will give us an idea of what a user with visual disabilities experiences when they read the content.

What is ARIA

ARIA is a W3C specification that describes how to increase the accessibility of web pages, in particular, dynamic content, and user interface components developed with Ajax, HTML, JavaScript, and related technologies.

The specification provides a set of roles, states, and properties that define accessible user interface elements and can be used to improve the accessibility and interoperability of web content and applications. These semantics are designed to allow an author to properly convey user interface behaviors and structural information to assistive technologies in document-level markup

HTML provides a full suite of accessibility affordances for the built in HTML elements; for example, a button is given the implicity role of button without assiging a `role` or `aria-role` attribute. This is the main reason why we should always use them when possible. Where we have to create our own elements we can use ARIA to provide cues for assistive technology devices on how to handle the custom elements.

One use for ARIA elements and attributes is to give assistive technology devices an association between title and its content.

```
<h2 id="table1-desc">Table title</h2>

<div class="content" aria-labelledby="table1-desc">
  <p>Content</p>
</div>
```

The example below compares a native button HTML element with a custom element.

The HTML button element looks like this

```
<button name="button">Click me</button>
```

This has many built in accessibility considerations:

- It's focusable
- It's part of the navigation order for the page
- Can be navigated with keyboard
- It creates a focus ring around it when you click on it

But there are times when we have to create our own element to represent a button either because it's generated dynamically or because we need specific functionality. However, that doesn't meant we can get away with not providing accessibility acommodations.

In the example below we're making a clickable button based on an HTML a element with an SVG image as the text. We've also added `tabindex` and `aria-pressed` attributes and assigned a `role` attribute.

Using Javascript we can change the value of aria-pressed attribute. we will look at it when we explore the Javascript portion of our custom element

Role, Property, State, and Tabindex Attributes

Role	Attribute	Element	Usage
`button`		`a`	▪ Identifies the element as a `button` widget. ▪ Accessible name for the button is defined by the text content of the element.
	`tabindex="0"`	`a`	▪ Includes the element in the tab sequence. ▪ Needed on the `a` element because it does not have a `href` attribute.
	`aria-pressed="false"`	`a`	▪ Identifies the button as a toggle button. ▪ Indicates the toggle button is not pressed.
	`aria-pressed="true"`	`a`	▪ Identifies the button as a toggle button. ▪ Indicates the toggle button is pressed.

```
<a tabindex="0" role="button" id="toggle" aria-pressed="false">
  Mute
  <svg aria-hidden="true">
```

```
    <use xmlns:xlink="http://www.w3.org/1999/xlink"
    xlink:href="images/mute.svg#icon-sound"></use>
  </svg>
</a>
```

The CSS will style using attribute selectors for the role attribute and aria-pressed attributes and the SVG element. The code below provides the following functionality:

- A base state
- A hover state
- A focus state that removes the default focus ring
- A custom focust state unsing the `::before` pseudo element that replaces the default focus we removed
- The same four states for a button with the `aria-pressed` attribute
- Styling for the SVG element

```
[role="button"] {
  display: inline-block;
  position: relative;
  padding: .4em .7em;
  border: 1px solid hsl(213, 71%, 49%);
  border-radius: 5px;
  box-shadow: 0 1px 2px hsl(216, 27%, 55%);
  overflow: visible;
  color: #fff;
  text-shadow: 0 -1px 1px hsl(216, 27%, 25%);
  background: hsl(216, 82%, 51%);
  background-image: linear-gradient(to bottom, hsl(216, 82%, 53%), hsl(216
}

[role="button"]:hover {
  border-color: hsl(213, 71%, 29%);
  background: hsl(216, 82%, 31%);
  background-image: linear-gradient(to bottom, hsl(216, 82%, 33%), hsl(216
    cursor: default;
  outline: none;
}
```

```
[role="button"]:focus {
  outline: none;
}

[role="button"]:focus::before {
  position: absolute;
  z-index: -1;
  /* button border width - outline width - offset */
  top: calc(-1px - 3px - 3px);
  right: calc(-1px - 3px - 3px);
  bottom: calc(-1px - 3px - 3px);
  left: calc(-1px - 3px - 3px);
  border: 3px solid hsl(213, 71%, 49%);
  /* button border radius + outline width + offset */
  border-radius: calc(5px + 3px + 3px);
  content: '';
}

''[role="button"]:active {
  border-color: hsl(213, 71%, 49%);
  background: hsl(216, 82%, 31%);
  background-image: linear-gradient(to bottom, hsl(216, 82%, 53%), hsl(216
  box-shadow: inset 0 3px 5px 1px hsl(216, 82%, 30%);
}

[role="button"][aria-pressed] {
  border-color: hsl(261, 71%, 49%);
  box-shadow: 0 1px 2px hsl(261, 27%, 55%);
  text-shadow: 0 -1px 1px hsl(261, 27%, 25%);
  background: hsl(261, 82%, 51%);
  background-image: linear-gradient(to bottom, hsl(261, 82%, 53%), hsl(261
}

[role="button"][aria-pressed][aria-pressed]:hover {
  border-color: hsl(261, 71%, 29%);
  background: hsl(261, 82%, 31%);
  background-image: linear-gradient(to bottom, hsl(261, 82%, 33%), hsl(261
```

```
    cursor: default;
}

[role="button"][aria-pressed="true"] {
  padding-top: .5em;
  padding-bottom: .3em;
  border-color: hsl(261, 71%, 49%);
  background: hsl(261, 82%, 31%);
  background-image: linear-gradient(to bottom, hsl(261, 82%, 63%), hsl(261
  box-shadow: inset 0 3px 5px 1px hsl(261, 82%, 30%);
}

[role="button"][aria-pressed="[aria-pressed="true"]:hover {
  border-color: hsl(261, 71%, 49%);
  background: hsl(261, 82%, 31%);
  background-image: linear-gradient(to bottom, hsl(261, 82%, 43%), hsl(261
  box-shadow: inset 0 3px 5px 1px hsl(261, 82%, 20%);
}

[role="button"][aria-pressed][aria-pressed]:focus::before {
  border-color: hsl(261, 71%, 49%);
}

[role="button"] svg {
  margin: .15em auto -.15em;
  height: 1em;
  width: 1em;
  pointer-events: none;
}
```


The Javascript takes care of the user interaction with our button.

We first deefine constants for the icon images and create an `init` functions to capture the button and set up event listeners for click and keydown events.

```
const ICON_MUTE_URL  = 'images/mute.svg#icon-mute';
const ICON_SOUND_URL = 'images/mute.svg#icon-sound';

function init () {
'images/mute.svg#icon-sound'ent.getElementById('toggle');

  'toggle'// Add event listeners to the various buttons
  toggleButton.addEventListener('click', toggleButtonEventHandler);
  toggleButton.addEventListener('keydown', toggleButtonEventHandler);
}
```

The next function handle events for the button both keyboard and click events.

The keydown event will handle pressing either the space key (keycode 32) or the enter key (keycode 13) to trigger the button state.

the `click` event handles both elements with a role of button or are a button element (the tag name is button).

```
function toggleButtonEventHandler (event) {
  const type = event.type;

  // Grab the keydown and click events
  if (type === 'keydown') {
    'keydown'// If either enter or space is pressed, execute the funtionev
      toggleButtonState(event);

      event.preventDefault();
    }
  }
  else if (type === 'click') {
    // Only a'click'// Only allow this event if either role is correctly s
    // or a correct element is used.arget.getAttribute('role') === 'button
      toggleButtonState(event);
    }
  }
}
'role'
```

The next function is the heart of the script and changes attributes in the a element and the `svg` child.

We set up variables to hold information about the event we're holding, the `aria-pressed` attribute and the value we want to start with (`true`)

We get the icon we want by catching the first use element inside the button and set it to the mute version of the icon.

IF the aria-pressed attribute has a `true` value then we change it to false and swap the icon to the unmuted version.

Finally we set the `aria-pressed` attribute to the new value and the `xlink:href` value to the new ICON.

The `xlink:href` attribute inside the svg element needs a little explanation. Because SVG is an XML-based language the regular href attribute will not work and we have to use XLINK, a vocabulary designed to link XML-based resourceas.

```
function toggleButtonState (event) {
  let button = event.target;
  let currentState = button.getAttribute('aria-pressed');
  let newState = 'true';

  let icon = button.getElementsByTagName('use')[0];
  let newIconState = ICON_MUTE_URL;

  'true'// If aria-pressed is set to true, set newState to false(currentSt
    newState = 'false';
    newIconState = ICON_SOUND_URL;
  }

  // Set'true'// Set the new aria-pressed state on the button
  button.setAttribute('aria-pressed', newState);
  icon.setAttribute('xlink:href', newIconState);
}
```

We set up the `init` function to run when the window is loaded. We could also use `domContentLoaded` instead but in this particular example either event works.

```
window.onload = init;
```

As you can see, implementing our own controls, even one as simple as a button, is a fairly complex task that requires a lot of scripting that the native button element gives you for free. This is always worth considering.

Best practices

Rather than reinvent the wheel I'll look at two collections of accessibility best practices: WAI-ARIA Authoring Practices 1.1 and Ebay's MIND Patterns: Accessibility Patterns for the Web (suggested by Rob Dodson).

we will take the accordion example from the ARIA Authoring Practices and explore what we need to have in an accessible component. Note that, as of this writing, MIND does not have suggestions for an accordion element.

This is a longer exercise than the one we did for the button demo earlier. Longer because of the element complexity, because there are many more moving parts and because there is no native equivalent on the web platform.

Before we get started there are a couple terminology items we need to get out of the way: headers and panels as they refer to the according object.

Accordion Header
Label for or thumbnail representing a section of content that also serves as a control for showing, and in some implementations, hiding the section of content.

In some accordions, there are additional elements that are always visible adjacent to the accordion header. For instance, a menubutton may accompany each accordion header to provide access to actions that apply to that section. And, in some cases, a snippet of the hidden content may also be visually persistent.

Accordion Panel
Section of content associated with an accordion header.

Next we look at the keyboard interactions that we need and should have for our

Keyboard Interaction

- Required elements
  - `Enter` or `Space`:
    - When focus is on the accordion header for a collapsed panel, expands the associated panel. If the implementation allows only one panel to be expanded, and if another panel is expanded, collapses that panel
    - When focus is on the accordion header for an expanded panel, collapses the panel if the implementation supports collapsing. Some implementations require one panel to be expanded at all times and allow only one panel to be expanded; so, they do not support a collapse function
- Optional Elements
  - `Down Arrow`: If focus is on an accordion header, moves focus to the next accordion header. If focus is on the last accordion header, either does nothing or moves focus to the first accordion header
  - `Up Arrow`: If focus is on an accordion header, moves focus to the previous accordion header. If focus is on the first accordion header, either does nothing or moves focus to the last accordion header.
  - `Home`: When focus is on an accordion header, moves focus to the first accordion header
  - `End`: When focus is on an accordion header, moves focus to the last accordion header.
  - `Control + Page Down`: If focus is inside an accordion panel or on an accordion header, moves focus to the next accordion header. If focus is in the last accordion header or panel, either does nothing or moves focus to the first accordion header
  - `Control + Page Up`: If focus is inside an accordion panel, moves focus to the header for that panel. If focus is on an accordion header, moves focus to the previous accordion header. If focus is on the first accordion header, either does nothing or moves focus to the last accordion header.

ARIA Roles, States, and Properties:

- The title of each accordion header is contained in an element with role button
- Each accordion header `button` is wrapped in an element with role heading that has a value set for aria-level that is appropriate for the information architecture of the page

        - If the native host language has an element with an implicit `heading` and `aria-level`, such as an HTML heading tag, a native host language element may be used
        - The `button` element is the only element inside the `heading` element. That is, if there are other visually persistent elements, they are not included inside the `heading` element
- If the accordion panel associated with an accordion header is visible, the header `button` element has aria-expanded set to `true`. If the panel is not visible, aria-expanded is set to `false`
- The accordion header `button` element has aria-controls set to the ID of the element containing the accordion panel content
- If the accordion panel associated with an accordion header is visible, and if the accordion does not permit the panel to be collapsed, the header `button` element has aria-disabled set to `true`
- Optionally, each element that serves as a container for panel content has role region and aria-labelledby with a value that refers to the button that controls display of the panel
    - Avoid using the `region` role in circumstances that create landmark region proliferation, e.g., in an accordion that contains more than approximately 6 panels that can be expanded at the same time
    - Role `region` is especially helpful to the perception of structure by screen reader users when panels contain heading elements or a nested accordion

The Code

As with the button example we will look at the HTML, CSS and Javascript separately to get an idea of what is involved to build this accessible component.

The HTML uses a description list to group together headers (using the `dt` element) and panels (using dd)

The code introduces the following ARIA attributes, some of them seen in the button demo:

- role provides a way to clasify elements according to function, similar to the Role 1.0 Recommendation. The semantics provided in ARIA are specific to accessibility roles, the Roles Recommendation provides more generic use caases
- aria-disabled Indicates that the element is perceivable (we can see it and read it in the page) but disabled (we can't interact with it, so it is not

editable or otherwise operable)
    - Used in conjunction with the `disabled` attribute
    - This is different than being hidden using the aria-hidden or being read-only using the aria-readonly
- aria-level defines the position of the element within the page hierarchy
    - Multiple elements in a set may have the same value for this attribute
    - This attribute can be used to provide an explicit indication of the level when that is not possible to calculate from the document structure or the aria-owns attribute
- aria-expanded indicates whether the element, or another grouping element it controls, is currently expanded or collapsed
    - If the element with the `aria-expanded` attribute controls the expansion of another grouping container that is not 'owned by' the element, the author should reference the container by using the `aria-controls` attribute.
- aria-controls Identifies the element (or elements) whose contents or presence are controlled by the current element
    - For the accordion we'll use the heading to control the associated panels
- aria-labelledby identifies the element (or elements) that labels the current element

```
<h1 id="page-title">Accordion Demo</h1>
<dl id="accordionGroup" role="presentation" class="Accordion">
  <dt role="heading" aria-level="3">
    <button   aria-expanded="true"
              class="Accordion-trigger"
              aria-controls="sect1"
              id="accordion1id">
      <span class="Accordion-title">Personal Information</span>
      <span class="Accordion-icon"></span>
    </button>
  </dt>
  <dd   id="sect1"
        role="region"
        aria-labelledby="accordion1id"
        class="Accordion-panel">
    <div>
      <fieldset>
```

```
        <p>
          <label for="cufc1">Name
            <span aria-hidden="true">*</span>:</label>
          <input  type="text"
                  value=""
                  name="Name"
                  id="cufc1"
                  class="required"
                  aria-required="true">
        </p>
        <p>
          <label for="cufc2">Email
            <span aria-hidden="true">*</span>:</label>
          <input  type="text"
                  value=""
                  name="Email"
                  id="cufc2"
                  aria-required="true">
        </p>
        <p>
          <label for="cufc3">Phone:</label>
          <input type="text" value="" name="Phone" id="cufc3">
        </p>
        <p>
          <label for="cufc4">Extension:</label>
          <input type="text" value="" name="Ext" id="cufc4">
        </p>
        <p>
          <label for="cufc5">Country:</label>
          <input type="text" value="" name="Country" id="cufc5">
        </p>
        <p>
          <label for="cufc6">City/Province:</label>
          <input type="text" value="" name="City_Province" id="cufc6">
        </p>
      </fieldset>
    </div>
  </dd>
```

```
<dt role="heading" aria-level="3">
  <button   aria-expanded="false"
            class="Accordion-trigger"
            aria-controls="sect2"
            id="accordion2id">
    <span class="Accordion-title">Billing Address</span>
    <span class="Accordion-icon"></span>
  </button>
</dt>
<dd   id="sect2"
      role="region"
      aria-labelledby="accordion2id"
      class="Accordion-panel" hidden>
  <div>
    <fieldset class="billing flex">
      <p>
        <label for="b-add1">Address 1:</label>
        <input type="text" name="b-add1" id="b-add1" />
      </p>
      <p>
        <label for="b-add2">Address 2:</label>
        <input type="text" name="b-add2" id="b-add2" />
      </p>
      <p>
        <label for="b-city">City:</label>
        <input type="text" name="b-city" id="b-city" />
      </p>
      <p>
        <label for="b-state">State:</label>
        <input type="text" name="b-state" id="b-state" />
      </p>
      <p>
        <label for="b-zip">Zip Code:</label>
        <input type="text" name="b-zip" id="b-zip" />
      </p>
    </fieldset>
  </div>
</dd>
```

```
<dt role="heading" aria-level="3">
  <button   aria-expanded="false"
            class="Accordion-trigger"
            aria-controls="sect3"
            id="accordion3id">
    <span class="Accordion-title">Shipping Address</span>
    <span class="Accordion-icon"></span>
  </button>
</dt>
<dd   id="sect3"
      role="region"
      aria-labelledby="accordion3id"
      class="Accordion-panel" hidden>
  <div>
    <fieldset>
      <p>
        <label for="m-add1">Address 1:</label>
        <input type="text" name="m-add1" id="m-add1" />
      </p>
      <p>
        <label for="m-add2">Address 2:</label>
        <input type="text" name="m-add2" id="m-add2" />
      </p>
      <p>
        <label for="m-city">City:</label>
        <input type="text" name="m-city" id="m-city" />
      </p>
      <p>
        <label for="m-state">State:</label>
        <input type="text" name="m-state" id="m-state" />
      </p>
      <p>
        <label for="m-zip">Zip Code:</label>
        <input type="text" name="m-zip" id="m-zip" />
      </p>
    </fieldset>
  </div>
</dd>
```

```
  </dl>
</body>
</html>
```

The CSS controls the layout and animation for the different items we've defined in markup

```
.Accordion {
    border: 1px solid hsl(0, 0%, 82%);
    border-radius: .3em;
    box-shadow: 0 1px 2px hsl(0, 0%, 82%);
}

.Accordion > * + * {
    border-top: 1px solid hsl(0, 0%, 82%);
}

.Accordion-trigger {
    background: none;
    border: 0;
    color: hsl(0, 0%, 13%);
    display: block;
    font-size: 1rem;
    font-weight: normal;
    margin: 0;
    padding: 1em 1.5em;
    position: relative;
    text-align: left;
    width: 100%;
}

.Accordion dt:first-child .Accordion-trigger {
    border-radius: .3em .3em 0 0;
}

.Accordion-trigger:focus,
.Accordion-trigger:hover {
```

```
    background: hsl(0, 0%, 93%);
}

.Accordion-icon {
    border: solid hsl(0, 0%, 62%);
    border-width: 0 2px 2px 0;
    height: .5rem;
    position: absolute;
    right: 1.5em;
    top: 50%;
    transform: translateY(-60%) rotate(45deg);
    width: .5rem;
}

.Accordion-trigger:focus .Accordion-icon,
.Accordion-trigger:hover .Accordion-icon {
    border-color: hsl(0, 0%, 13%);
}

.Accordion-trigger[aria-expanded="tr[aria-expanded="true"].Accordion-icon
    transform: translateY(-50%) rotate(-135deg);
}

.Accordion-panel {
    margin: 0;
    padding: 1em 1.5em;
}

fieldset {
    border: 0;
    margin: 0;
    padding: 0;
}

input {
    border: 1px solid hsl(0, 0%, 62%);
    border-radius: .3em;
    display: block;
```

```
    font-size: inherit;
    padding: .3em .5em;
}
```

I've broken the Javascript in sections to make sure I'm not inundating readers with bunches of terminology that I'm working on understanding myself.

For all the elements that have the class `Accordion` we create an array anf run the instructions below.

We set up constant holding information about the array, if allows toggle (has the attribute `data-allow-toggle`) and whether it allows multiple panels (has the attribute `data-allow-multiple`).

Then we create arrays for elements inside the accordion; the triggers (`Accordion-trigger` class) and the panel (`Accordion-panel` class).

```
Array.from(document.querySelectorAll('.Accordion')).forEach(function (acco

  // Allow for each toggle to both open and close individually
  const allowToggle = accordion.hasAttribute('data-allow-toggle');
  'data-allow-toggle'// Allow for multiple accordion sections to be expand

  // Create the array'data-allow-multiple'// Create the array of toggle el
  const triggers = Array.from(accordion.querySelectorAll('.Accordion-trigg
  const panels = Array.from(accordion.querySelectorAll('.Accordion-panel')
```

I've broken the `click` event into two sections to handle the case where we don't allow multiple panels and the two cases for `allowToggle` and `isExpanded`.

The idea for the first block is that, if we don't allow multiple panes to be open we want the open panes to close when we open a new one.

```
  accordion.addEventListener('click', function (event) {
    const target = event.target;

    if (target.classList.contains('Accordion-trigger')) {
```

```
      const isExpanded = target.getAttribute('aria-expanded') == 'true';

      if (!allowMultiple) {
        triggers.forEach(function (trigger) {
          if (trigger.getAttribute('aria-expanded') == 'true') {
            document.getElementById(trigger.getAttribute('aria-controls')
                    .setAttribute('hidden', '');
            trigger.setAttribute('aria-expanded', 'false');
          }
        });
      }
```

Then we test if we allow toggle events and whether the panel is expanded. If we allow both of these then we remove the `hidden` attribute and switch the value of the `aria-expanded` attribute to false.

If we don't allow toggle and the element is not expanded we remove the `hidden` attribute from the element with `aria-control` and set the `aria-expanded` attribute to true

```
      if (allowToggle && isExpanded) {

        document.getElementById(target.getAttribute('aria-controls'))
                .setAttribute('hidden', '');
        target.setAttribute('aria-expanded', 'false');
      }
      else if (!allowToggle &'hidden'anded) {
        document.getElementById(target.getAttribute('aria-controls')).rem
        'aria-controls'// Set the expanded state on the triggering element
        target.setAttribute('aria-expanded', 'true');
      }

      event.preventDefault();
    }
  });
```

The keyboard navigation for the accordion is more complicated as we have to consider where we are in the accordion or inside one of its children.

The keydown event listener intoruces control modifiers to keyboard events. `ctrlModifier` only returns true if its assigned keys **and** the control key (represented by `event.ctrlKey`) are pressed.

The accordion works with the following key codes:

- 33 = Page Up
- 34 = Page Down
- 38 = Up,
- 40 = Down
- 35 = End
- 36 = Home

If the element we're testing for has the class `Accordion-trigger` then we test which element are we over. If it's an element the key wwe pressed were the up or down key or the `control + PageUp` or `control + PageDown` combinations.

```
// Bind keyboard behaviors on the main accordion container
accordion.addEventListener('keydown', function (event) {
  const target = event.targe'keydown'nst key = event.which.toString();
    const ctrlModifier = (event.ctrlKey && key.match(/33|34/));

  /33|34/// Is this coming from an accordion header?
  if (target.classList.contains('Accordion-trigger')) {
    'Accordion-trigger'// Up/ Down arrow and Control + Page Up/ Page Dow
    if (key.match(/38|40/) || ctrlModifier) {
      const index = triggers.indexOf(target);
      const direction = (key.match(/34|40/)) ? 1 : -1;
      const length = triggers.length;
      const newIndex = (index + length + direction) % length;

      triggers[newIndex].focus();

      event.preventDefault();
    }
```

If we match ehter home or end we move to the first or last panel respectively.

```
      else if (key.match(/35|36/)) {
        // 35 = End, 36 = Home keyboard operations
        switch (key) {
          // Go to first accordion
          case '36':
            triggers[0].focus();
            break;
          '36'// Go to last accordion
          case '35':
            triggers[triggers.length - 1].focus();
            break;
        }

        event.preventDefault();
      }
    }
```

If we use `control + PageUp` or `control + PageDown` then we want to make sure we do two things: prevent the default browser behavior and focus on the panel that we want to.

```
    else if (ctrlModifier) {
      // Control + Page Up/ Page Down keyboard operations
      // Catches events that happen inside of panels
      panels.forEach(function (panel, index) {
        if (panel.contains(target)) {
          triggers[index].focus();

          event.preventDefault();
        }
      });
    }
  });
});
```

An accordion is useful but it's a lot of work to make it accessible. Still, it's a good example of how to build accessible components.

Test the results

Now that we've created accessible components. Now we get to test them.

To test accessible components we'll use 3 tools:

- The Accessibility Audit built into Chrome Dev Tools
- aXe and aXe Coconut from Deque Systems
- Screen Readers

Lighthouse Accessibility Test

In recent versions of Google Chrome there is a new **Audits** panel in Dev Tools. I start here because it uses Lighthouse to run the test and aXe Core under the hood so, unless you have specific requirements why aXe Core or Coconut are required, this may be the only tool you need outside of a screen reader.

To get started open Dev Tools (`Control+Shift+I` in Windows, `Command+option+i` on Mac). You will see something similar to the image below. Click on `Audits`.

Figure 1: Audit Panel in Chrome Dev Tools

This will present you with a list of possible audits to run. In this case we want to uncheck all audits except accessibility. Then we click Run `Audit`.

Figure 2: Available Audits in Chrome Dev Tools

Chrome will run its tests and give you a score and a list of items to change shown below.

Figure 3:

Accessibility
Audits
Results

It’s important to realize that, while automated testing is good and will get you most of the way there, manual testing and decision making are still important.

In the results shown in figure 3 I know that the constrast issues are the colors that I use for syntax highlighting (using Prism.js against the theme background). It’s up to me to decide if I want to change the theme to provide better contrast or keep the theme and the score of 91.

aXe and aXe Coconut Browser Extensions

aXe browser extension for Chromeand Firefox automates testing and evaluation of your page’s accessibility.

> if you're using Firefox, be aware that the Firefox Extension may not work with the latest vesions. Deque provides an explanation about what version of aXe supports what version of the browser.

If you need the latest functionality, for example you’re working with ShadowDOM and Custom Elements you can use aXe Coconut Chrome Extension to use the latest aXe features and tests. The instructions are the same for both and they can be installed concurrently; think of Coconut as the aXe version of Chrome Canary.

For Chrome the process is simple:

Download the appropriate extension from the Chrome Web Store

Install the extension when prompted

Open DevTools and select aXe or aXe Coconut (Figure 4 is open with aXe)

Figure 4: aXe ready to run

Click Analyze on the left-hand frame

Axe will prodeuce a report with all accessibility violations (figure 5 shows aXe Coconut)

Figure 5: aXe Coconut report

As with the Lighthouse report there are things that we'll have to manually decide if there are errors or not and whether we need to change the code to fix the problems aXe reported.

Testing With A Screenreader

The last part of the accessibility evaluation is to use a screen reader to read the page back to you. This will not catch accessibility violations like Lighthouse or aXe do but it'll give you an idea of easy it is for screen readers to understand the content of your page or app.

Rather than try to walk you through using a screen reader I'll link to two awesome introductory tutorials from Rob Dodson, part of his A11ycasts series in Youtube: one for Voice Oveer (built into macOS) and NVDA (free for Windows).

It has been an eye opening experience to hear my content read back to me as an editor would; someone who doesn't understand the content as well as I do and who doesn't read what he meant to write rather than what's actually written.

It's Not Just The Tech

When we think about accessibility most of us think about screen readers but there's a lot more. The automated and manual techniques are a starting point. After you make all the changes suggested by the tools you still have to make decisions about your content.

This is the beginning of my learning accessibility. Will report back along the learning path.

Links and Resources

- Accessibility

        - ARIA, Accessibility APIs and coding like you give a damn! – Léonie Watson
        - Leonie Watson: Using ARIA to solve everyday accessibility problems
        - Inert Attribute
- ARIA
    - Using ARIA
    - Best Practices
        - WAI-ARIA Authoring Practices 1.1
        - MIND Patterns: Accessibility Patterns for the Web
- Voice Over
    - Voice Over Getting Started Guide
    - Voice Over Commands
- aXe and aXe Coconut
    - Test the leading edge of accessibility with aXe Coconut and aXe-core 3.0-alpha
    - Browser Extensions
        - aXe Chrome Extension (Stable)
        - aXe Coconut Chrome Extension
        - WAVE
        - Firefox Devtools Integration
- Chrome specific
    - How To Do an Accessibility Review
    - The new way to test accessibility with Chrome DevTools - A11ycasts #23