Web Accessibility (Web A11y)

Accessibility is taking care of people with disabilities during design and construction of anything - Ramp in a mall, lifts which can be operated by anyone etc. In case of web apps / sites, it means making it usable for disabled people using Assistive Technologies (ATs).

Types of Disabilities

We should keep in mind the different disabilities people may have, so we can design and develop web apps / sites that are usable by them. Such people may use Assistive Technologies (ATs) to help them navigate and use the web.

- *Vision* : blind, partially blind, color blind
- *Hearing* : deafness, hard-of-hearing
- *Physical* : broken bones (temporary), lack of motor control etc. (they have difficulty using mouse because of tremors, hand shivers).
- *Cognitive* : Dyslexia, Autism, psychiatric problems etc.
- *Changing abilities* : Old age, broken bones etc.

Types of Assistive Technologies (ATs)

- Screen readers (SRs) - help outline web pages, read aloud page elements, and navigate to various page elements, announcing names (labels/titles etc.), announce form input states, read out live regions (a section of a page where content changes dynamically - eg. alert messages on form submission) etc.
- Keyboard - Vision, hearing impaired people etc. often use keyboard to navigate and use pages - tab, arrow keys and some other modifier keys help in this process. SRs are often used in conjunction with keyboard.
- Refreshable Braille Displays - used by the vision-impaired.
- Speech recognition software and devices
- Screen magnifiers
- Other equipment and mechanisms like foot pedals, eye-tracking etc.

Popular Screen Readers (SRs)

- JAWS (Job Access With Speech) - for Windows only, has paid license. If you use the free version you will need to restart every 40 minutes or so. https://support.freedomscientific.com/Downloads/JAWS
- NVDA (NonVisual Desktop Access) - for Windows only, free (but you can donate if you like). It is best used with Mozilla Firefox. https://www.nvaccess.org/download/
- VoiceOver - for Mac OS X, native, free and comes pre-installed on Mac
- Narrator - on Windows, native, free.

Planning for A11y

- Design for a11y early on, and choose tools and technologies that support a11y.
- Design user personas with disabilities

EXERCISE: Read about A11y in Google Search page

A11y Standards

- USA : Section 508 of US Laws - US Federal government agencies stick to these standards for tools they use, developer and require software vendors too to follow them in software developed for Federal agencies - https://www.section508.gov/
- W3C : WCAG 2.0, 2.1 - Web Content A11y Guidelines - a11y guidelines for web apps in general - https://www.w3.org/WAI/standards-guidelines/wcag/
- W3C : WAI-ARIA - Web A11y Initiative - Accessible Rich Internet Applications - for accesssible custom widgets that use JS, Ajax etc. - https://www.w3.org/WAI/standards-guidelines/aria/

Section 508

This is an amendment to Rehabilitation Act of 1973 (USA). It was enacted in 1998, and updated many times later. Section 508 mandates US Federal agencies to make Information and Communication Technology (ICT) they make use to be accessible. Development, procurement, maintenance and use of such technology should take care of a11y. **Software vendors for such agencies are thus mandated by law to conform to a11y standards.**. Conformance to AA level of WCAG, ensures Section 508 requirements are also taken care.

Recommended: Take the following training to understand more about Section 508 - https://www.section508.gov/508-training/courses/new/

WCAG

WCAG is a set of guidelines for improving a11y of web apps published by Web A11y Initiative (WAI) of W3C. WCAG 1.0 was published in 1999, WCAG 2.0 in 2008, and WCAG 2.1 in 2018. WCAG 2.0 defines 12 guidelines in 4 categories (abbreviated P.O.U.R.), and 3 levels of conformance. WCAG 2.1 added 1 criteria and is fully backwards-compatible with WCAG 2.1 - if a site is WCAG 2.1 compliant at some level of conformance, it is surely WCAG 2.0 compliant at the same level. WCAG 2.1 added criteria by focusing on needs of people with cognitive disabilities, and a11y of touch devices.

WCAG Guidelines - P.O.U.R, Levels of Conformance, and Success Criteria

WCAG defines **3 levels of conformance - A (lowest), AA, AAA (highest)** and **success criteria** to meet the 4 P.O.U.R guidelines. Make special note of which guidelines / sub-guidelines are required to which which of the levels of conformance (A/AA/AAA).

- Perceivable: ensure web content is perceivable by at least one of the senses for every person
  - Images: Text alternatives
  - Time-based media - Alternatives for video / audio
  - Adaptable - Present content in semantic way
  - Distinguishable - Make it easy to see and hear content (
- Operable: all UI controls usable by everyone (in some way)
  - Keyboard Accessible - fucntionality available using keyboard
  - Enough time - to read and use content and UI elements
  - Prevent seizures - no flashing content, dizzying animations etc.
  - Navigable - helpful to use to navigate, locate content and focus
  - Input modalities - support for touch-friendly devices by supporting gestures, accessible touch targets etc. (this is the guideline added in WCAG 2.1)
- Understandable: easy to understand content (is predictable and simple)
  - Readable - text content
  - Predictable - consistent and predictable UI and navigation
  - Input assistance - help users on errors
- Robust: works across a variety of browsers (desktop and mobile) and ATs - also, future-proof
  - Compatible - with various browsers and ATs

NOTE:

1. Generally **AA level of conformance is expected** because AAA is way too much to achieve and not required for general web apps.
2. All pages, and all parts of a page MUST meet the guidelines for the desired level of conformance
3. Use the WCAG 2.1 requirements - success criteria and techniques to evaluate your implementation. You can expland on a criteria to see the full description and useful techniques to meet the criteria. https://www.w3.org/WAI/WCAG21/quickref/)

WAI-ARIA

WAI-ARIA is covered in detail in a separate section later.

A11y Techniques

General techniques

- Validate HTML and CSS using validators - editors these days are usually powerful enough, so this may not be required. A Nu HTML checker is an experimental project to validate HTML in a better way.
- Stick to consistent design - eg. logout link, notifications on top right of page etc.
- Design CSS classes etc. with flexibility and user customization in mind (toggling animation, changing fonts and sizes, application theme etc.)
- Avoid flickering, flashing and dizzying animations
- Provide way to stop / pause moving / auto-updating content - eg. carousels, live news/social media feeds, stock tickers, live sport scores.
- Do not use only CSS to convey meaning. For example red color for error messages - this will not be helpful for color blind and blind users.
- Provide keyboard shortcuts if possible for user interactions, especially for custom components. The accesskey attribute is to simplify this, but is not well supported as of now.
- Warnings on time limits - eg. time left to log off, enter OTP, before page will refresh and entered data lost etc.
- Unnecessary scrollbars should be avoided.
- Avoid complex wording
- Do not remove focus ring (or some such indicator) - you can change it if required to suit your taste - the focus ring helps sighted users who may have hand tremors etc. and use keyboard navigation. button is clicked.
- Color and style
    - Ensure color contrast for text and images
    - Use dashed lines, markers for highlighting in maps etc. as a means for distinguishing apart from colors - never rely upon color as it may not be distinguishable by some people/
    - Ensure CSS styles allow for easy customization - for example use relative units like em/rem/% when compared to absolute units like pt/px/cm etc. The relative units make it easy to scale font sizes, box dimensions etc. based on devices by centralizing the base value. These also help scale font size etc. when zooming in/out.
    - Use of flex box *order* property to rearrange UI element in an order other than source-order if required
- Typography
    - Prevent long lines

    - Sufficient spacing between lines
    - Left-justified text is preferable
    - Adjustable font size, right type faces
    - Sufficient size for font (16px - 20px is usual value for body copy) According to WCAG 1.4.12 guideline, we can roughly say, the following settings should not make content unusable in any element (what is shown is for html and body elements, but this is applicable to all elements) - so content should be tested with these values set and checked (and then reset back to values desired by us!). We should not get unnecessary scrollbars, text that is hidden away etc. due to these settings.

```
html, body {
  font-size: 1em;
  line-height: 1.5;
  letter-spacing: 0.12em;
  word-spacing: 0.16em;
}

p {
  margin-top: 2em;
  margin-bottom: 2em;
}
```

- Accessible content
    - Providing lang attribute for the page, and wherever necessary
    - Semantic markup - choose tags as per meaning of content. Also stick to single `h1`, `main` etc.
    - Do not disable user scaling of the page. Set the viewport meta tag such. Avoid using the maximum-scale and minimum-scale values as they can restrict zooming on the page, making the page inaccessible to people with low vision etc.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

    - `title` attribute may be used for giving more information - eg. what happens when an action
    - Use ul, ol, and dl appropriately - ul for lists with no inherent ordering, ol for lists with inherent ordering, and dl for definition lists (FAQs, glossary)
    - Use `caption` for captioning table. Table caption element must be the first child of the table element (it can be placed visually at the bottom using CSS `caption-side: bottom`). Use `scope` attribute for scoping header cells to row/col/rowgroup/colgroup. Use thead, tbody, tfoot. Use multiple tbody elements to divide table body into logical sections if it is required.
    - Use figure and figcaption etc. for captioning images
    - Consistent identification of UI elements (eg. search box is called *Search* everywhere)
    - Large touch-targets, far-apart touch targets
    - Use visually hidden text to provide text for a11y but prevent text for sighted users
        - Eg. Use icon + text - helps vision-impaired - text can be optionally visually hidden
    - Describing purpose of form inputs, proper error messages
    - Stick to native HTML elements and controls as much as you can. For eg. do not enable navigation via JavaScript - use a plain HTML link. Where you need the look of a button but want navigation you can always use CSS styles to style a link to look like a button.
- Accessible navigation
    - Page hierarchy that is understandable and well-outlined and sectioned - one h1, no skipped heading levels, one main, proper use of semantic tags, outlining using article, section, header, footer, aside etc., appropriate use of lists

- Enclose nav links in a list
- Make page navigable by keyboard
- Avoid switching from a list of links to a dropdown on small screens for navigation menu. This causes unexpected change in behavior. Unlike link clicks, disabled users do not expect page-to-page navigation when an option is selected. It is better to provide a Menu of links that collapses and comes into view through a button click (the *Hamburger Menu* - it is again preferable to use the word Menu instead o the Hamburger icon) - use `display: none` to hide this and not `sr-only` (explained later), as `sr-only` would make it unnecessarily focusable.
- Add a *skip link* at the beginning, in order to be able to go directly to main content by skipping things like the top-level navigation menu on a page
- Use the same order for nav links listed in multiple places - eg. site navigation links listed in the main menu, as well as footer
- Link text that adds context where required (avoid "Read more" kind of links - some extra context that may can be visually hidden may be provided)
- Source order is to be preferred for navigation using keyboard (focus shifts from element-to-element in source order).
- Apart from SEO, a site map can also help a11y. A general structure for site map is a set of nested lists of links. One outline for such markup is such. Note that this is for a sitemap HTML page and NOT a sitemap.xml defined according to sitemaps.org specification

```
<nav aria-label="site-map">
    <h2>Navigation Section Heading</h2>
    <ul>
        <li>
            <a href="target-1">Link 1</a>
            <ul>
                <li><a href="target-1a">Link 1a</a></li>
                <li><a href="target-1b">Link 1b</a></li>
                <li><a href="target-1c">Link 1c</a></li>
                ...
            </ul>
        </li>
        ...
    </ul>
    ...
</nav>
```

- Use appropriate ARIA attributes - ARIA is explained in a separate section later.

Landmarks

Landmarks refer to sectioning containers. These are significant as they help ATs "outline" the document, thus improving semantics, and also present menus to quickly navigate to the main parts of the page. Some of these are `header`, `h1` - `h6`, `nav`, `main`, `aside`, `footer` etc. If any landmark element needs to be added as one, appropriate **ARIA landmark roles** are available.

```
<div role="search">
    <label for="search-users">Search</label>
    <input type="search" id="search-users" />
</div>
```

Hidden Content

- Often content needs to be visually hidden (from people without disabilities), but present for ATs to read and present more context to disabled users. `display : none`, `visibility: hidden`, `width: 0; height: 0`, and the HTML `hidden` attribute hide text from many ATs too. This is a class that will hide content in browsers but

still make it available for ATs. You will find variations of this class used by other developers (usually involving the clip CSS property), but the idea is more or less the same.

```
.sr-only {
  position: absolute !important;
  top: auto;
  width: 1px;
  height: 1px;
  overflow: hidden;
  white-space: nowrap;
}
```

NOTE: Some UI libraries use this name for the class. You may need to be careful when using other libraries to avoid conflicts. Another popular name used for this class is `visually-hidden`.

- For content that needs to be hidden from ATs too, you can use usually used CSS. For example,

```
[hidden] {
  display: none;
}
```

- Sometimes content needs to be visible but hidden from ATs. In such case use `aria-hidden` attribute

```
<span aria-hidden="true">I will be visible but will not be announced by a Screen Reader</span>
```

Skip Links

Many times such visually hidden text may also need to be visible on focus. For example - *skip links*. Skip links are links that helps people using keyboard navigation. They are visually hidden but appear on the screen focus. They help the user to skip over a set of links (usually main or sidebar navigation links) and go directly to some content they may be interested in on the page (usually the main content). In such cases, adding a class like `focusable` apart from `sr-only` can do the trick. As you can observe, this mainly resets properties set by `sr-only`

```
.sr-only.focusable:active, .sr-only.focusable:focus {
    position: static !important;
    height: auto !important;
    width: auto !important;
    overflow: visible !important;
}
```

Keyboard / Programmatic Focus and tabindex values

- The HTML attribute tabindex is used to set behavior of focus when tabbing using keyboard.
- When using keyboard, `TAB` moves focus forward, `SHIFT+TAB` moves backward.
- Tab order is ideally the default - i.e. the source order of focusable elements in the document.
- Not all HTML elements are focusable. The focusable ones are (mainly) - button, input, links (anchor tags with href set)
- Other elements can be made focusable by adding the tabindex attribute. The following is the meaning of various `tabindex` values.
  - tabindex="-1" - element is focusable ONLY using JavaScript (.focus() DOM method) and does not gain focus through keyboard navigation. This is useful to focus on error messages for example on form submission. The element gains focus by click too.
  - tabindex="0" adds the element in the source order. The element will be focusable using keyboard navigation. **This is the preferred value for tabindex**.

- tabindex="1", "2", etc. defines relative order of focus (1 comes first, then 2 etc.). Positive values should be avoided unless absolutely necessary as they can go against the expected order of focus. **Reference**: https://developer.paciellogroup.com/blog/2014/08/using-the-tabindex-attribute/#:~:text=The%20HTML%20tabindex%20attribute%20is,web%20content%20for%20keyboard%20users.

Descriptive links

- Avoid *Read More* kind of links as they do not give context which sighted users have. Give descriptive links as far as possible - eg. *Read more of the premium plan features*. This may not be possible always. In such case a technique like the one below can help

```
<a href="#target">Read more <span class="sr-only">premium plan features</span></a>
```

Forms

- Provide a label. Placeholder may not be recognized by ATs. Also placeholders do not give context even for sighted users once text input is not empty. If labels are not required for sighted users (eg. search), make sure to provide visually hidden labels.

```
  <label for="search-users" class="sr-only">Search</label>
  <input type="search" id="search-users" />
```

- Clearly describe expected input - eg. Password requirements. Use ARIA attributes `aria-describedby` to set the relationships between input and the description.

```
<input type="password" aria-describedby="password-help" />
<div id="password-help">
  Choose a strong password. A minimum of 1 lowercase, 1 uppercase, 1 digit, and 1 special charact
</div>
```

- Do not use only color to indicate required fields, errors etc. Use explicit text like *Required* or an indicator like an asterisk (*). In case of asterisk, it is better to provide a message at the beginning of the form like **All fields marked with (*) are required**.
- Clear error messages on error. If you like, you may also summarize the list of errors on top of the form like so, and focus after populating errors when user tries to submit. The aria-live attribute makes known to ATs that the section of the page will change dynamically and hence it should announce its content to user when that happens.

```
<section aria-live="assertive" tabindex="-1">
  <a href="#password">Password is missing a special character.</a>
  ...
</section>
```

- Prefer native controls over custom controls - making custom controls function similar to native controls is too much work and often unnecessary. If custom control is required, do make it accessible using ARIA attributes, and handling necessary user interactions using JavaScript.

SVG

Scalable Vector Graphics (SVG) provide a way to construct vector graphices using code. They can be used to construct images lige logos etc.

- Inline SVG, i.e. SVG embedded in HTML is better for a11y
- Use role="img" to declare an SVG as an image, so ATs consider them so
- Set title, and a longer description (if required) like so. These are not rendered visually in an SVG (but can be made to do so if required).

```
<svg id="my-svg" version="1.2" width="300" height="200" aria-labelledby="my-svg-title" aria-descr
  <title id="my-svg-title">Title of the image</title>
  <desc id="my-svg-description">Longer decription of the same SVG image may be provided for bette
</svg>
```

For support across various browsers and ATs it is recommended that ARIA attributes be used in conjunction.

Reference: https://www.sitepoint.com/tips-accessible-svg

Time-based Media (Video, Audio)

- Provide audio transcriptions. You can transcribe speech using speech recognition software like Google Docs Voice Typing, Windows Speech Recognition, Apple Dictation etc. YouTube has auto-subtitling feature. Make sure to include other details with the transcripts - name of speaker, non-verbal cues like voice tone, scenery, music that is playing, ambient noises etc. The transcript can be provided in a separate page if desired so.
- You can provide open captions (part of video usually), or closed captions, i.e. CC (can be turned on / off).

WAI-ARIA

The Web A11y Initiative - Accessible Rich Internet Applications (WAI-ARIA), or ARIA in short, is a W3C published specification on increasing a11y of web pages, especially those with custom UI components like custom dropdowns, modal dialogs etc. (i.e. those developed using ajx, JS etc.).

NOTE OF CAUTION: ARIA is powerful as it instructs/guides ATs to understand and interpret your content better. If ARIA attributes are used incorrectly you will end up misguiding them. Bad ARIA is worse than no ARIA!

Five Rules of ARIA

1. Do not use ARIA when an HTML alternative exists - it takes much more work to get interactions right and often this is unnecessary as alternative exists.

   ```
   <span role="button">Button</span> <!-- (Bad) -->
   <button>Button</button> <!-- (Good) -->
   <button class="action-button"><i class="fas fa-setings"></i></button> <!-- Good to enclose icons
   <a href="#target"> <!-- Good to enclose all areas that all have the same navigation target in a s
    <img src="image/path" />
    <h3>Profile name</h3>
   </a>
   ```

2. Do not change native HTML element semantics using ARIA (in such case ARIA attribute values will be considered by ATs).

   ```
   <!-- Bad, since anchor tag has a different purpose. Also being a link it is activated using ENTER
   <a href="#" role="button">Button</a>
   ```

3. Keyboard support must be provided for interactive widgets defined using ARIA roles.
4. If some element is visible and focusable, you must not hide it form ATs, i.e. you must NOT use `role="presentation"`, or `aria-hidden="true"` in it. As discussed earlier `aria-hidden="true"` hides the element from ATs. The `role="presentation"` setting removes native HTML semantics for the element.
5. All interactive elements should have an **accessible name** - There are various ways to provide a name for a element which is used by ATs to announce during focus etc. (`aria-label` for example is one way). For good practices on providing accessible names check here - https://simplyaccessible.com/article/accessible-name/

ARIA Roles

ARIA Roles provide semantics to UI elements. They are defined using the `role` atttribute in HTML.

```
<div role="alert">Your message has been sent.</div>
```

There are many roles and are categorized like so..

- Widget roles - these define UI widgets. Widgets may be *standalone* or *composite* - *button*, *checkbox*, *radio* etc. are standalone, *menu*, *menubar*, *radiogroup*, *tablist* are composite. Some standalone roles are part of composite widgets - eg. *menuitem*, *tab*, *tabpanel*.
- Document structure roles - These roles describe structure that orgaizes content. They are usually not interactive. Eg. *application*, *headin*, *figure*, *list*, *img*, *article*, *tooltip*, *separator* etc. Where possible HTML5 semantic tags must be used instead.
- Landmark roles - These define navigational landmarks. Example - *banner*, *main*, *form*, *navigation*, *search*. Where possible HTML5 semantic tags must be used instead.
- Live Region roles - To mark content that changes dynamically (using JS). ATs announce changes to content in them (also see section on `aria-live` attribute). Example - *alert*, *log*, *status*.
- Window roles - Act as logical windows. Example - *alertdialog*, *dialog*

ARIA States and Properties

- **ARIA States** assign states like disabled based on current state of a UI widget. Some of the ARIA states are `aria-disabled`, `aria-invalid`, `aria-busy` (used in live regions which is explained below), , `aria-hidden` (hidden from ATs).
- **ARIA Properties** support widget roles (the ARIA role attribute) and often used with it. Apart from this there are other properties relating to live regions, drag-and-drop, and relationships. Some of the ARIA properties are `aria-label`, `aria-checked`, `aria-autocomplete`, `aria-modal`, `aria-valuemax`, `aria-placeholder`, `aria-controls`, `aria-owns`. Some sample ARIA properties are explained below.
  - Use `aria-label` to label sections appropriately. For example, you can label different navs on the page such (aria-labels need not be unique)

    ```
    <nav aria-label="site">...</nav>
    <nav aria-label="page">...</nav>
    ```

    Use `aria-labelledby` and `aria- describedby` for providing labels and more detailed descriptions in a separate element. **EXERCISE**: Understand the difference between `aria-label`, `aria-labelledby`, and `aria-describedby` here - https://developers.google.com/web/fundamentals/accessibility/semantics-aria/aria-labels-and-relationships and here - https://stackoverflow.com/questions/19616893/difference-between-aria-label-and-aria-labelledby
  - Use `aria-live="assertive"` or `aria-live="polite"` on live regios - eg. those updated through JS like error messages, or that change content like carousel, stock tickers, sports scores etc. Also some roles define live regions - `role="alert"`, `role="log"`, `role="status"`. Refer: https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Live_Regions
  - Use `aria-current="page"` for link representing current page in a site navigation menu, or for current page in breadcrumbs, `aria-current="step"` for current step in a list of steps, `aria-current="date"` for current date in a calendar, `aria-current="true"` in general for anything that is *current*!

Note on Building Interactive Widgets

- Composite widgets consist of multiple elements. The appropriate ARIA roles must be applied to the container element, as well as the elements that the container element *owns* (for example, an element with `role="menu"` _owns_ elements with role="menuitem"```).
- Additionally ARIA properties and states must be set on the elements. There are rules stating which ARIA states and properties can be applied to which ARIA roles. In the specficiations for the role the applicable ones are mentioned. Make sure to check which states and properties can be used on which roles.
- Keyboard support must be provided for interactive widgets defined using ARIA roles.

- You will find sample implementations for popular widgets here - WAI ARIA Authoring Practices (with code examples for custom widgets) and WAI ARIA - Code examples for custom widgets). However there are only sample implementations and may not work well in some ATs. They also do not take care of mobile devices (touch events and gestures). You can use these as a base to start creating interactive widgets.

Exercise: Look into some of the interactive widgets and build them.

Accessibility Tree

It is a subset of the DOM tree. It shows all attributes relevant to the accessibility of a DOM node. This includes *name*, *role* and ARIA states and properties (coveying *state* and *value*). This tree is what is used by ATs to understand the web page and present their navigation as well. The a11y tree can be viewed in Chrome by going to the Elements tab -> Accessibilty menu on the right (same place where CSS styles etc. appear).

Reference: https://developers.google.com/web/fundamentals/accessibility/semantics-builtin/the-accessibility-tree

Accessibility Testing

- As the first-level testing, you need to check if the a11y tree (a tree consisting of a subset of DOM nodes) is constructed as per your intent. This tree is constructed by browsers, and serves as a basis for ATs including SRs to understand your pages. Hence the AT should expose name, value and states of elements as you intend.
- As the second-level testing, SRs can be used. **ARIA is still not fully and consistently supported by all SRs**. So you need to test across various SRs. You can also use popular a11y testing tools apart form SRs - eg. WAVE evaluation tool for Chrome (also available online on https://wave.webaim.org/), AXE Web Accessibility Testing, Web Disability Simulator for Chrome, HTML5 Outliner for Chrome etc. https://chrome.google.com/webstore/category/extensions
- As the third-level testing, a11y tests can be built into the development workflow using tools like AXE Core (https://www.deque.com/axe/). This was a11y tests can be run automatically and ensured before intergration of a change into the codebase. For Node.js apps for example, this is the package - https://www.npmjs.com/package/axe-core.
- Finally test with real disabled users whenever possible - this is the best way for testing and much better than all of the steps above when possible.

References

- Web a11y tutorial by W3C
- Web A11y reource on Webaim.org
- Articles on A11y on MDN
- A11y - Web Fundamentals by Google
- A11y - Yale University
- A11y - Web.dev
- Section 508 and a11y related trainings
- U.S. Access Board
- WCAG 2.1 requirements - success criteria and techniques
- WAI-ARIA resources
- WAI-ARIA 1.1 Specifications
- WAI-ARIA 1.2 Specifications - Draft
- WAI ARIA Authoring Practices
- WAI ARIA - Code examples for custom widgets)
- ARIA Landmarks Example
- Keyboard A11y - Webaim.org