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 and Assistive Technologies.

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.
- Other equipment like foot pedals 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.

Accessibility Testing

- As the first-level testing, you need to check if he 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. 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.

Planning for a11y

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

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 to follow them in siftware developed for Federal agencies too. 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/

WCAG

WCAG 2.0 defines 12 guidelines in 4 categories (abbreviated P.O.U.R.) - Perceivable (P) - ensure web content is perceivable by at least one of the senses for every person - Operable (O) - all UI controls usable by everyone (in some way) - Understandable (U) - easy to understand content (is predictable and simple) - Robust (R) - works across a variety of browsers (desktop and mobile) and ATs - also, future-proof

- Define levels of conformance - A (lowest), AA, AAA (highest)
- Success criteria

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

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.

Reference: 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/)

NOTE: WCAG added criteria by focusing on needs of people with cognitive disabilities, and a11y of touch devices.

WCAG Guidelines

Make special note of which guidelines / sub-guidelines are required to which which of the levels of conformance (A/AA/AAA).

- Perceivable
  - 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
    - 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
    - Readable - text content
    - Predictable - consistent and predictable UI and navigation
    - Input assistance - help users on errors
- Robust
    - Compatible - with various browsers and ATs

General A11y techniques

- Design CSS classes etc. with flexibility and user customization in mind (toggling animation, changing fonts and sizes, application theme etc.)
- Do not use CSS only to convey meaning. For example red color for error messages - this will not be helpful for color blind and blind users.
- 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
- Toggling animations - avoid flickering, flashing and dizzying animations, and provide option to turn animation off anyway
- Providing keyboard shortcuts
- Describing purpose of form inputs, proper error messages
- Warnings on time limits - eg. time left to log off, enter OTP, before page will refresh and entered data lost etc.
- 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.
- 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.
- `title` attribute may be used for giving more information - eg. what happens when an action 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
    - Sufficient size for font (16px - 20px is usual value for body copy)

    - Adjustable font size, right type faces
- Accessible navigation / content
    - Semantic markup
    - 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
    - Navigable by keyboard
    - Adding 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
    - 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).

Visually 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.

```css
.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`.

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`

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

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.

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>
```

- Clear error messages on error
- 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**.
- 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.

References

- Web a11y tutorial by W3C
- WCAG 2.1 requirements - success criteria and techniques