Keyboard checklist

Accessibility - Keyboard checklist epic ticket: STP-1384 - Accessibility - Keyboard checklist TO DO

Please note

- This checklist should be completed before the screen reader checklist.
- The WCAG provides a section dedicated to key interaction: WCAG A-AAA 2.1 (Only level A and AA compliance is required, level AAA is optional)

Required

1.General

- Set a language for the page: `<html lang="en">` (WCAG A 3.1.1)
- Setting `outline: 0` is an anti-pattern. Use the :focus:visible class to have the focus ring appear on keyboard interactions (and not mouse interactions). (WCAG AA 2.4.7)
- For **offscreen elements**, like a hidden hamburger menu, set `display: none` or `visibility: hidden` when the element is not visible, and then change to `display: block or flex` or `visibility: visible` when the element comes back on screen.
- For users with cognitive and learning differences and challenges, target a 9th grade reading level. Use https://readable.com/ to understand the reading level of your tool. (WCAG 3.1.5 AAA Only level A and AA compliance is required, level AAA is optional)
- Avoid jargon and include explanations and definitions for concepts that are specific to our tools, including abbreviations and acronyms. (WCAG AAA 3.1.3 and WCAG AAA 3.1.4 Only level A and AA compliance is required, level AAA is optional)
- Ensure character-only shortcut keys can be turned off or modified. This is **not applicable** for our viz tools, but is being included for completeness. (WCAG A 2.5.4)
- Ensure HTML uses complete start and end tags, elements are nested according to their specifications, elements do not contain duplicate attributes, and any IDs are unique. (WCAG A 4.1.1 This requirement will be removed in WCAG 2.2 and is considered fulfilled via WCAG 2.1 when using HTML.)

2.Semantics

- Use **HTML5 semantic elements** as they have built-in ARIA roles (except in older browsers). A few common sectioning elements are listed below. Here's an exhaustive list of other semantic elements. (WCAG A 1.3.1 and AAA 1.3.6 Only level A and AA compliance is required, level AAA is optional)

`<aside>`	The aria role is "complementary". A section of the document, designed to be complementary to the main content at a similar level in the DOM hierarchy, but remains meaningful when separated from the main content. Our Antd Sider component is rendered as an <aside> and includes an aria-label that is used both for the Sider and the Drawer component which is used on smaller/mobile screen sizes.
`<footer>`	A <footer> typically contains information about the author of the section, copyright data or links to related documents. Our app /tool footers are typically provided by the vizhub template, but allow for customization within the app/tool. The aria role is "contentinfo" when the <footer> is directly part of the <body>
`<form>`	The aria role is "form" when it has an accessible name using aria-label, aria-labelledby, or title attributes.
`<header>`	The <header> includes introductory content, typically a group of introductory or navigational aids. It may contain some heading elements but also a logo, a search form, an author name, and other elements. Our app/tool headers are typically provided by the vizhub template, but allow for customization within the app/tool. The aria role is "banner".
`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`	Section headings. Be sure to use them sequentially and don't skip levels. These are important for the headings rotor that screen reader users often utilize.

`<main>`	The <main> represents the dominant content of the tool. Our <main> is typically rendering through the App.jsx component. **Example from WHO Rehab - App** `return ( <Spin indicator={<Loader loading={debouncedLoading} />} spinning={debouncedLoading}> {status !== 'loading' && ( <main className={classnames.app}> <ViewToggle /> <ControlPanel /> <ViewWrapper /> </main> )} </Spin> );`
`<nav>`	The <nav> represents a section of a page whose purpose is to provide navigation links. Our <nav> elements are typically controls to change the view. **Example from WHO Rehab - ViewToggle** `return ( <nav className={`${classnames.viewTabs} ${classnames.containerStyle}`}> <ThemeProvider theme={theme}> <Tabs indicatorColor="primary" textColor="primary" value={activeView} variant="scrollable" onChange={handleChange} > {views} </Tabs> </ThemeProvider> </nav> );`
`<section>`	The <section> represents a generic standalone section of a document, which doesn't have a more specific semantic element to represent it. Our tools might use a section to contain the visualization content. **Example from WHO Rehab - ViewContainer** `export default function ViewContainer({ children }) { return ( <section className={`${classnames.viewContainer} ${classnames.containerStyle}`}> {children} </section> ); }`

- When using **React**, try to return a semantic element or a React Fragment from your render functions instead of a `<div>` . (WCAG A 1.3.1 and AAA 1.3.6 Only level A and AA compliance is required, level AAA is optional)

Example from WHO Rehab - ControlPanel

```
return (
    <div
      aria-label="Control panel to change chart settings."
      className={`${classnames.controlPanelContainer} ${classnames.containerStyle}`}
    >
      {status === 'success' && (
        <>
          {isVisible.location && <LocationControl />}
          {isVisible.multiLocation && <MultiLocationControl />}
          {isVisible.condition && <ConditionControl />}
          {isVisible.age && <AgeControl />}
          {isVisible.sex && <SexControl />}
          {isVisible.measure && <MeasureControl />}
          {isVisible.metric && <MetricControl />}
          {isVisible.year && <YearControl />}
          {isVisible.yearRange && <YearRangeControl />}
          {isVisible.rateOfChange && <RateOfChangeControl />}
          {isVisible.conditionLevel && <ConditionLevelControl />}
          {isVisible.uncertainty && <UncertaintyControl />}
          {isVisible.reset && <ResetControl />}
        </>
      )}
    </div>
  );
```

3.Focus (WCAG AA 2.4.7)

- Keyboard and switch device users rely on **focus** to alert them to their position within the web page or tool. These users tab or click through the interactive/UI elements on the page. All interactive elements should be focusable and have a visual indicator of focus (see more in Styling). Without a visual indicator, the keyboard user has no idea where they are on the page.
- Focus is really **only needed on interactive elements**, don't add a tab-index to every part of your page. If you've properly employed semantic and, in particular, header elements, users will be able to navigate effectively.
- When possible, use **standard UI elements** as they are implicitly focusable with a tab order and keyboard event handling built in (e.g., <a>, <button>, default <input> elements).
- Ensure your **tab order/visual order** follows the DOM order. Tab order is the order in which a keyboard user will move through the tool when they press the tab key. (WCAG A 2.4.3)
    - Tab order is controlled through the tabindex attribute (or `tabIndex={0` } prop in React).
    - Values for tabindex: "-1" removes element from typical tab order and allows for focus() to be added, "0" makes the element part of the typical tab order, any value >0 is an anti-pattern
    - tabindex is only needed on custom interactive controls (i.e., not on the standard interactive elements) or when you need to bring focus to an element. Most Antd and MUI components already include a tab index when needed.
- Ensure there are no keyboard traps, i.e., places where a keyboard user might get stuck and not be able to navigate away using the keyboard alone. (WCAG A 2.1.2)
    - When necessary, you can add the ability for a user to press ESC to exit an element. See example below.

Add ability to use ESC

```
// in Sider.jsx
import { useEvent } from 'react-use';
...
// For accessibility, the trigger button should close when the user presses Escape
  const buttonRef = useRef();
  const isFocused = useIsFocused(buttonRef);
  const onKeyDown = useCallback(
    ({ key }) => isFocused && key === 'Escape' && toggleVisible(),
    [isFocused, toggleVisible],
  );
  useEvent('keydown', onKeyDown);

const trigger = (
    <Tooltip
      arrow={{ pointAtCenter: true }}
      placement={visible ? 'bottomRight' : 'right'}
      title={`${visible ? 'Hide' : 'Show'} settings panel`}
      trigger={['hover', 'focus']}
    >
      <Button
        ref={buttonRef} // add buttonRef to the trigger button
        aria-label={triggerAriaLabel}
        className={classnames.trigger}
        style={styleTrigger}
        onClick={toggleVisible}
      >
        <FontAwesomeIcon icon={visible ? faXmark : faSliders} style={iconStyle} />
        <span>{visible ? 'Hide settings' : 'Settings'}</span>
      </Button>
    </Tooltip>
  );

// in useIsFocused.js
import { useCallback, useEffect, useMemo, useState } from 'react';

export default function useIsFocused(ref) {
  const [activeElement, setActiveElement] = useState(document.activeElement);

  const handleActiveElement = useCallback(() => {
    setActiveElement(document.activeElement);
  }, [setActiveElement]);

  useEffect(() => {
    document.addEventListener('focus', handleActiveElement, true);
    document.addEventListener('blur', handleActiveElement, true);

    return () => {
      document.removeEventListener('focus', handleActiveElement);
      document.removeEventListener('blur', handleActiveElement);
    };
  }, [handleActiveElement]);

  return useMemo(() => document.hasFocus() && ref.current === activeElement, [activeElement, ref]);
}
```

- We also need to ensure that keyboard users can't enter closed control panels. This is easily accomplished with the 'inert' attribute. See example below.

Inert attribute

```
// in Sider.jsx
...
const [visible, setVisible] = useState(siderOpen);
...
  const toggleVisible = useCallback(() => {
    onSiderChange(!visible);
    setVisible(!visible);
  }, [onSiderChange, visible]);
...
// Example for Drawer component, used on mobile
<Drawer
...
  <div inert={visible ? null : ''} style={{ width: '100%' }}>
    {children}
  </div>
</Drawer>

// Example for Sider component, used on desktop
<AntSider
...
  {/* Use 'inert' to prevent keyboard/screen reader focus entering Sider when it is closed. */}
  <div inert={visible ? null : ''}>{children}</div>
</AntSider>
```

4.Skip Links (WCAG A 2.4.1)

- Add a **skip links** option for keyboard users, allowing them to skip navigation and go directly to the main content.
- The new vizhub template includes a skip links to the main content section. So, if your app uses the template, you won't need to add this.

Add skip links <a>

```
// add link before <nav> or whatever you want to skip
<a href="#main-content" class="skip-links">Skip to main content</a>
...
// For older browsers, you'll need to add tabindex=-1
<main id="main-content" >
```

Style the skip-link class

```
/* Style skip-link so that it's offscreen until it's focused through the keyboard */
.skip-link {
  position: absolute;
  top: -40px;
  left: 0px;
  z-index: 100;
  /* other styles */
}
.skip-link:focus {
  top: 0;
}
```

5.Videos

- Include controls (including audio) for videos/animation, most importantly be sure to include pause. (WCAG A 1.4.2 and WCAG A 2.2.1)
- Don't use flashing or strobing content that flashes more than three times in any one second period (WCAG AAA 2.3.2 Only level A and AA compliance is required, level AAA is optional)

6.Custom interactive elements (WCAG A-AAA 2.1 and WCAG A 3.2.1)*

WG TODO: Viz team to evaluate if we can switch button sets to drop downs for various control panels - probably want to maintain for main view switching

WG TODO: Viz team to investigate correct semantic elements for labeling and containing controls

WG TODO: Viz team to review Container and Label abstraction in the Starter repo and update code accordingly

WG TODO: Viz team to investigate how to make tooltips accessible to keyboard users and screen readers

WG TODO: Viz team to investigate other accessible form element / control libraries. Some controls such as the ClearAllIcon and the Slider are not WCAG A 2.5.2 Pointer Cancellation compliant.

- When **creating custom interactive elements** (e.g., menu, menu-button, checkbox, slider, radio buttons, alert), the ARIA design patterns guide provides specifics about functionality and required/optional aria attributes.
    - Switch any info button tooltips to an accessible tooltip
- It is often easier to use interactive elements from a UI component library (see below). These libraries typically have proper accessibility functionality and semantics built-in. With some tweaks as noted below, most Antd and MUI components will be accessible
- (Only level A and AA compliance is required, level AAA is optional)

7.Timed interactions*

WG TODO: Viz team to investigate removing alerts altogether and updating the wording in an application + role='alertdialog'

- If your tool uses auto-closing Alerts (like on the copy sharelink url alert in the vizhub template), you will need to disable auto-close and allow for manual close only. See the alert design pattern for reference. (WCAG A 2.2.1)
    - The vizhub template uses the reakit Dialog component, role ="alertdialog", which is a closeable alert (see example below). This will most likely switch to ariakit upon introduction to the monorepo. For tools that use AntDesign, there is an Alert component with a closeable prop

Reakit closeable alert

```
import { Dialog, DialogDisclosure } from 'reakit/Dialog';

import { forwardRef, useCallback, useEffect, useRef } from 'preact/compat';
import { useTranslation } from 'react-i18next';
import { Button } from 'reakit';
import { Dialog, DialogDisclosure } from 'reakit/Dialog';

import { useClickAway, useDialog, useMenu } from 'src/hooks';
import * as defaultProps from './default-props';
import * as propTypes from './prop-types';

import './menu-dialogue.scss';

function MenuDialogue({ children, ...props }, ref) {
  const clickAwayRef = useRef(null);
  const menu = useMenu();
  const dialog = useDialog();
  const { t } = useTranslation();

  // Close alert when user clicks the close button or presses ESC, then
  // pass focus back to the main menu button
  const handleClose = useCallback(() => {
    dialog.alertMessageSet(undefined);
    dialog.alertTypeSet(undefined);
    dialog.hide();
    if (menu.unstable_disclosureRef.current) {
      menu.unstable_disclosureRef.current.focus();
    }
  }, [dialog, menu.unstable_disclosureRef]);

  useEffect(
    () => (dialog.alertMessage ? dialog.setVisible(true) : dialog.setVisible(false)),
    [dialog],
  );

  // Close alert when user clicks away with mouse
  useClickAway(clickAwayRef, handleClose);

  return (
    <>
        // DialogDisclosure is a button that opens the dialog when clicked
      <DialogDisclosure ref={ref} {...dialog} {...props}>
        {children}
      </DialogDisclosure>
      <Dialog
        ref={clickAwayRef}
        {...dialog}
        aria-labelledby="alert-message-id"
        className={`alert alert--${dialog.alertType}`}
        role="alertdialog"
        onKeyDown={handleClose}
      >
        <span id="alert-message-id">{dialog.alertMessage}</span>
        <Button
          aria-label={t('Close alert')}
          className="alert-close"
          data-testid="alert-close"
          type="button"
          onClick={handleClose}
        >
          <i className="fa-light fa-xmark" />
        </Button>
      </Dialog>
    </>
  );
}
```

- If there is a time limit on an interaction, however, a user should be able to turn it off, adjust it (10x default limit), or extend it (be given 20 seconds to extend via a simple interaction like pressing the spacebar).
- Ensure users can pause, stop, or hide any moving, blinking, or scrolling that starts automatically, lasts more than five seconds, and is presented in parallel with other content. Most of our viz tools do not have such moving content. In the case of Play buttons on certain charts, we just need to ensure we allow for pausing, especially if we activate the Play button automatically. (WCAG A 2.2.2)

8.Input modalities - Navigation and other elements*

WG TODO: Viz team to investigate other accessible form element / control libraries

- Let users operate touchscreens with one finger and reduced gestures (WCAG A 2.5.1). In particular, we need to address horizontal scrolling (which equates to a path-gesture) in Material UI nav tabs at mobile sizes. Be sure to use the tablet and smaller screen size functionality where we include left/right arrow buttons to horizontally scroll the tabs on smaller screens and ensure it is included in mobile. We should NOT use solely the horizontal swipe (path-gesture) to scroll between nav tabs.
- Ensure that we are using either standard html form elements, antd components, or MUI components for controls. This should ensure we meet the pointer cancellation requirements (WCAG A 2.5.2) which it easier for users to prevent accidental or erroneous pointer input by cancelling pointer operations. (UW accessibility team to review our tools and provide recommendations.)

9.Input modalities - Maps*

WG TODO: Viz team to determine best map package to switch to

- Let users operate touchscreens with one finger and reduced gestures (WCAG A 2.5.1). For tools with maps, ensure the map is navigable using L/R and up/down arrow keys. Adding L/R and up/down arrow buttons on the map is most ideal but not required.

Resources

UI component libraries

The following React component libraries provide good keyboard accessibility.

- Material UI (Good support for keyboard and screen readers.)
- Antd (While keyboard accessibility is good, screen reader accessibility is rather less so for certain components–e.g., Select and Tree Select–at this time.)
- Reakit (which you will see still used in the vihub template) has been replaced by AriaKit (Good support for keyboard and screen readers.)

Testing

- Navigate through your tool using solely the tab, arrow, and enter keys. Press the space bar to scroll.
    - Please note:
        - You have to turn on tab use in **Safari**. Preferences > Advanced > Accessibility, then check on "Press tab to highlight...".
        - In **Firefox,** you may also need to check off an option to get the tab to work. Preferences > General > Browsing, then check off "Always use the cursor keys to navigate...".
- Be sure you can interact with/change all viz controls and enter information into any input elements or forms.
- Be sure you can see a focus-ring on all interactive elements. In some cases, you will need to use an alternative to the focus-visible outline, like underlining text or adding a border due to how Antd styles their focused elements. We need to have a very clear and obvious indicator that doesn't rely on color alone.
- Test on Chrome, Safari, and Firefox. We can do some Windows browser testing by using LambdaTest or using an IT provided PC.
- Add a live expression so you can see currently focused element. This helps in debugging when you can't see a focus ring.
- Accessibility bookmarklets will show you things like landmarks, headings, etc. directly on your page. Installation feels strange in that you literally drag and drop these links into your bookmarks bar! This wont' show you your focus ring, however.
- The aXe chrome extension and WAVE tool will do a full accessibility evals on any externally deployed page, but won't pick up focus ring issues.