Accessibility

Web accessibility (also known as a11y) refers to the practice of creating websites that can be used by anyone — be that a person with a disability, a slow connection, outdated or broken hardware or simply someone in an unfavorable environment. For example, adding subtitles to a video would help both your deaf and hard-of-hearing users and your users who are in a loud environment and can't hear their phone. Similarly, making sure your text isn't too low contrast will help both your low-vision users and your users who are trying to use their phone in bright sunlight.

Ready to start but aren't sure where?

Checkout the Planning and managing web accessibility guide provided by World Wide Web Consortium (W3C)

Skip link

You should add a link at the top of each page that goes directly to the main content area so users can skip content that is repeated on multiple Web pages.

Typically this is done on the top of `App.vue` as it will be the first focusable element on all your pages:

```template
<span ref="backToTop" tabindex="-1" />
<ul class="skip-links">
  <li>
    <a href="#main" ref="skipLink" class="skip-link">Skip to main content</a>
  </li>
</ul>
```

To hide the link unless it is focused, you can add the following style:

```css
.skip-links {
  list-style: none;
}
```

```
  margin: 1em auto;
  top: 0;
  position: fixed;
  left: 50%;
  margin-left: -72px;
  opacity: 0;
}
.skip-link:focus {
  opacity: 1;
  background-color: white;
  padding: 0.5em;
  border: 1px solid black;
}
```

Once a user changes route, bring focus back to the very beginning of the page, right before the skip link. This can be achieved by calling focus on the `backToTop` template ref (assuming usage of `vue-router`):

```vue
<script setup>
import { ref, watch } from 'vue'
import { useRoute } from 'vue-router'

const route = useRoute()
const backToTop = ref()

watch(
  () => route.path,
  () => {
    backToTop.value.focus()
  }
)
</script>
```

Read documentation on skip link to main content

Content Structure

One of the most important pieces of accessibility is making sure that design can support accessible implementation. Design should consider not only color contrast, font selection, text sizing, and language, but also how the content is structured in the application.

Headings

Users can navigate an application through headings. Having descriptive headings for every section of your application makes it easier for users to predict the content of each section.

- Nest headings in their ranking order: `<h1>` - `<h6>`
- Don't skip headings within a section
- Use actual heading tags instead of styling text to give the visual appearance of headings

Read more about headings

```template
<main role="main" aria-labelledby="main-title">
  <h1 id="main-title">Main title</h1>
  <section aria-labelledby="section-title-1">
    <h2 id="section-title-1"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- Content -->
  </section>
  <section aria-labelledby="section-title-2">
    <h2 id="section-title-2"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- Content -->
    <h3>Section Subtitle</h3>
    <!-- Content -->
  </section>
</main>
```

Landmarks

Landmarks provide programmatic access to sections within an application. Users who rely on assistive technology can navigate to each section of the application and skip over content. You can use ARIA roles to help you achieve this.

HTML	ARIA Role	Landmark Purpose
header	role="banner"	Prime heading: title of the page
nav	role="navigation"	Collection of links suitable for use when navigating the document or related documents
main	role="main"	The main or central content of the document.
footer	role="contentinfo"	Information about the parent document: footnotes/copyrights/links to privacy statement
aside	role="complementary"	Supports the main content, yet is separated and meaningful on its own content
search	role="search"	This section contains the search functionality for the application
form	role="form"	Collection of form-associated elements

section	role="region"	Content that is relevant and that users will likely want to navigate to. Label must be provided for this element

Read more about landmarks

Semantic Forms

When creating a form, you can use the following elements: `<form>`, `<label>`, `<input>`, `<textarea>`, and `<button>`

Labels are typically placed on top or to the left of the form fields:

```template
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      :type="item.type"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

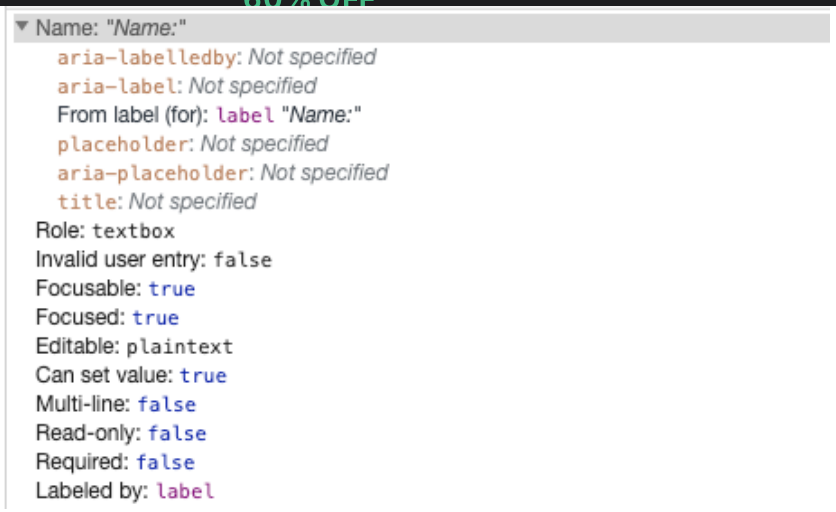
Notice how you can include `autocomplete='on'` on the form element and it will apply to all inputs in your form. You can also set different values for autocomplete attribute for each input.

Labels

Provide labels to describe the purpose of all form control; linking `for` and `id`:

```template
<label for="name">Name: </label>
<input type="text" name="name" id="name" v-model="name" />
```

If you inspect this element in your Chrome DevTools and open the Accessibility tab inside the Elements tab, you will see how the input gets its name from the label:



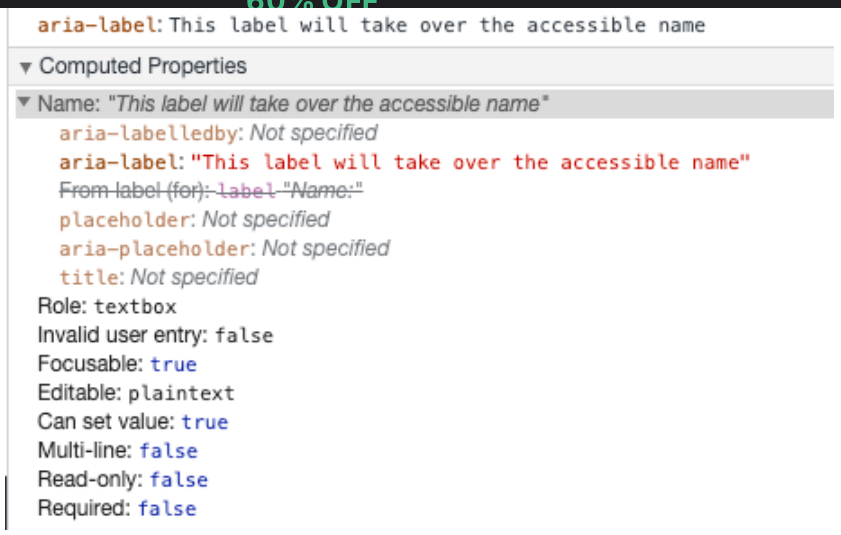
> ⚠ **Warning:**
>
> Though you might have seen labels wrapping the input fields like this:
>
> ```template
> <label>
>   Name:
>   <input type="text" name="name" id="name" v-model="name" />
> </label>
> ```
>
> Explicitly setting the labels with a matching id is better supported by assistive technology.

`aria-label`

You can also give the input an accessible name with `aria-label`.

```template
<label for="name">Name: </label>
<input
  type="text"
  name="name"
  id="name"
  v-model="name"
  :aria-label="nameLabel"
/>
```

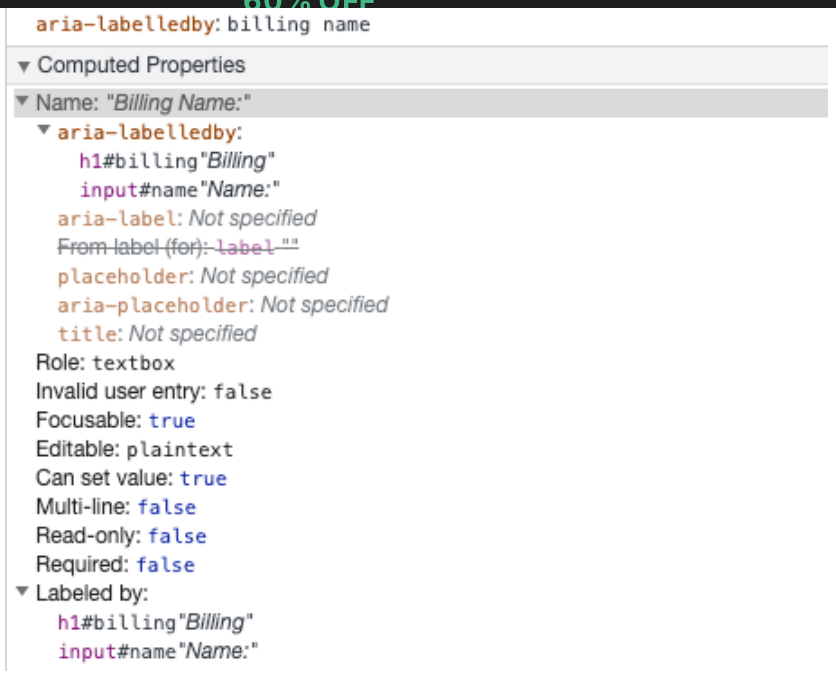
Feel free to inspect this element in Chrome DevTools to see how the accessible name has changed:



`aria-labelledby`

Using `aria-labelledby` is similar to `aria-label` except it is used if the label text is visible on screen. It is paired to other elements by their `id` and you can link multiple `id`s:

```template
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Name: </label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

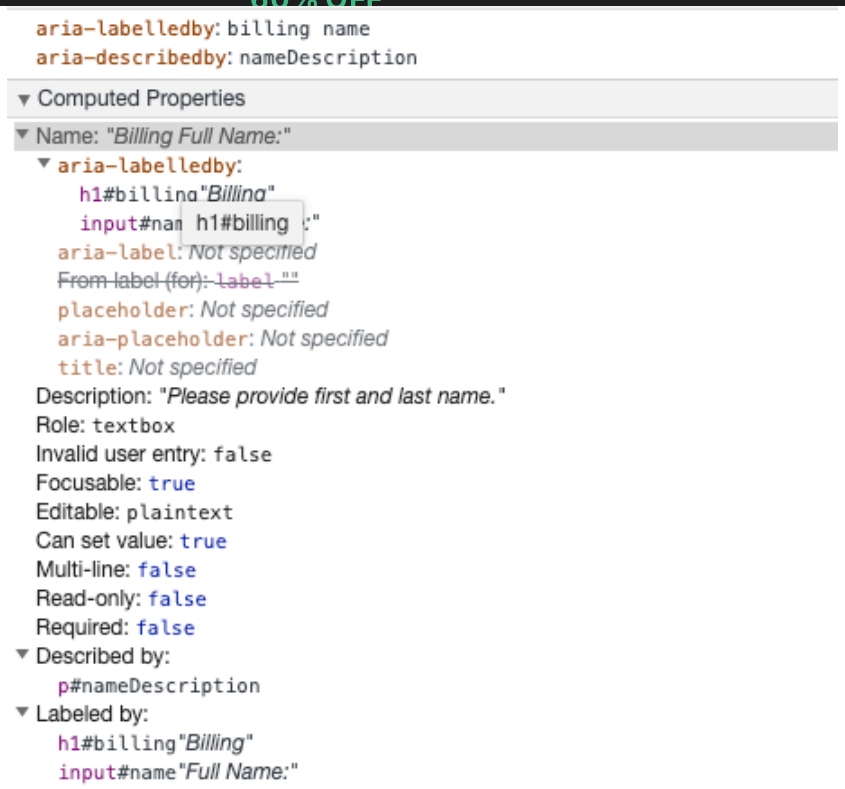


`aria-describedby`

aria-describedby is used the same way as `aria-labelledby` except provides a description with additional information that the user might need. This can be used to describe the criteria for any input:

```template
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Full Name: </label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
      aria-describedby="nameDescription"
    />
    <p id="nameDescription">Please provide first and last name.</p>
  </div>
  <button type="submit">Submit</button>
</form>
```

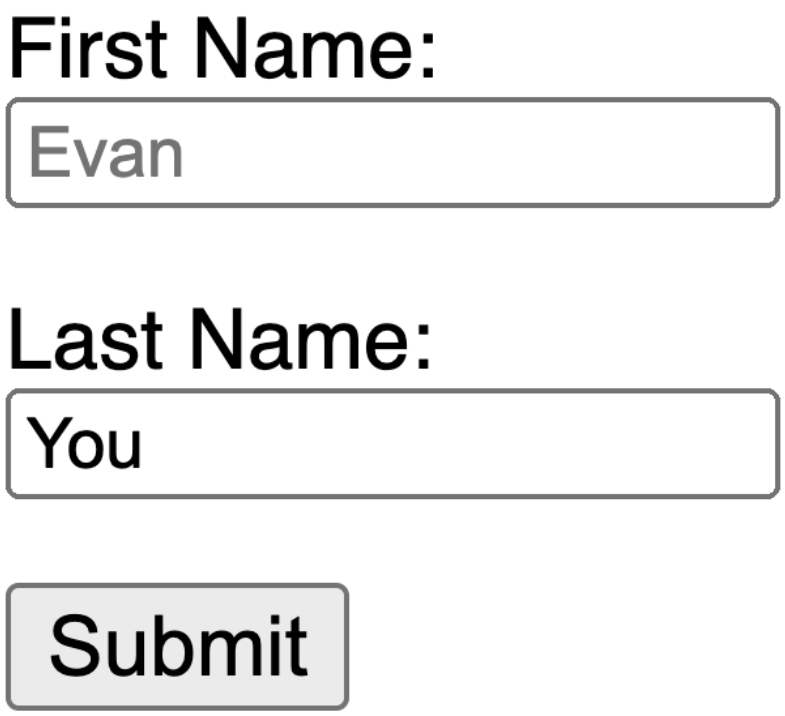
You can see the description by inspecting Chrome DevTools:



Placeholder

Avoid using placeholders as they can confuse many users.

One of the issues with placeholders is that they don't meet the color contrast criteria by default; fixing the color contrast makes the placeholder look like pre-populated data in the input fields. Looking at the following example, you can see that the Last Name placeholder which meets the color contrast criteria looks like pre-populated data:



```
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      type="text"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
      :placeholder="item.placeholder"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

```css
/* https://www.w3schools.com/howto/howto_css_placeholder.asp */

#lastName::placeholder {
  /* Chrome, Firefox, Opera, Safari 10.1+ */
  color: black;
  opacity: 1; /* Firefox */
}

#lastName:-ms-input-placeholder {
  /* Internet Explorer 10-11 */
  color: black;
}

#lastName::-ms-input-placeholder {
  /* Microsoft Edge */
  color: black;
}
```

It is best to provide all the information the user needs to fill out forms outside any inputs.

Instructions

When adding instructions for your input fields, make sure to link it correctly to the input. You can provide additional instructions and bind multiple ids inside an `aria-labelledby`. This allows for more flexible design.

```template
<fieldset>
  <legend>Using aria-labelledby</legend>
  <label id="date-label" for="date">Current Date: </label>
  <input
    type="date"
    name="date"
    id="date"
    aria-labelledby="date-label date-instructions"
```

```
</fieldset>
```

Alternatively, you can attach the instructions to the input with `aria-describedby`:

```template
<fieldset>
  <legend>Using aria-describedby</legend>
  <label id="dob" for="dob">Date of Birth: </label>
  <input type="date" name="dob" id="dob" aria-describedby="dob-instructions" />
  <p id="dob-instructions">MM/DD/YYYY</p>
</fieldset>
```

Hiding Content

Usually it is not recommended to visually hide labels, even if the input has an accessible name. However, if the functionality of the input can be understood with surrounding content, then we can hide the visual label.

Let's look at this search field:

```template
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <button type="submit">Search</button>
</form>
```

We can do this because the search button will help visual users identify the purpose of the input field.

We can use CSS to visually hide elements but keep them available for assistive technology:

```css
.hidden-visually {
  position: absolute;
  overflow: hidden;
  white-space: nowrap;
  margin: 0;
  padding: 0;
  height: 1px;
  width: 1px;
  clip: rect(0 0 0 0);
  clip-path: inset(100%);
}
```

`aria-hidden="true"`

Adding `aria-hidden="true"` will hide the element from assistive technology but leave it visually available for other users. Do not use it on focusable elements, purely on decorative,

```template
<p>This is not hidden from screen readers.</p>
<p aria-hidden="true">This is hidden from screen readers.</p>
```

Buttons

When using buttons inside a form, you must set the type to prevent submitting the form. You can also use an input to create buttons:

```template
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <!-- Buttons -->
  <button type="button">Cancel</button>
  <button type="submit">Submit</button>

  <!-- Input buttons -->
  <input type="button" value="Cancel" />
  <input type="submit" value="Submit" />
</form>
```

Functional Images

You can use this technique to create functional images.

- Input fields
  - These images will act as a submit type button on forms

```template
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <input
    type="image"
    class="btnImg"
    src="https://img.icons8.com/search"
    alt="Search"
  />
</form>
```

- Icons

```template
<form role="search">
  <label for="searchIcon" class="hidden-visually">Search: </label>
  <input type="text" name="searchIcon" id="searchIcon" v-model="searchIcon" />
  <button type="submit">
    <i class="fas fa-search" aria-hidden="true"></i>
    <span class="hidden-visually">Search</span>
```

Standards

The World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI) develops web accessibility standards for the different components:

- User Agent Accessibility Guidelines (UAAG)
  - web browsers and media players, including some aspects of assistive technologies
- Authoring Tool Accessibility Guidelines (ATAG)
  - authoring tools
- Web Content Accessibility Guidelines (WCAG)
  - web content - used by developers, authoring tools, and accessibility evaluation tools

Web Content Accessibility Guidelines (WCAG)

WCAG 2.1 extends on WCAG 2.0 and allows implementation of new technologies by addressing changes to the web. The W3C encourages use of the most current version of WCAG when developing or updating Web accessibility policies.

WCAG 2.1 Four Main Guiding Principles (abbreviated as POUR):

- Perceivable
  - Users must be able to perceive the information being presented
- Operable
  - Interface forms, controls, and navigation are operable
- Understandable
  - Information and the operation of user interface must be understandable to all users
- Robust
  - Users must be able to access the content as technologies advance

Web Accessibility Initiative – Accessible Rich Internet Applications (WAI-ARIA)

W3C's WAI-ARIA provides guidance on how to build dynamic content and advanced user interface controls.

- Accessible Rich Internet Applications (WAI-ARIA) 1.2
- WAI-ARIA Authoring Practices 1.2

Resources

Documentation

- WCAG 2.0
- WCAG 2.1
- Accessible Rich Internet Applications (WAI-ARIA) 1.2
- WAI-ARIA Authoring Practices 1.2

Assistive Technologies

- Screen Readers
  - NVDA
  - VoiceOver
  - JAWS
  - ChromeVox
- Zooming Tools
  - MAGic
  - ZoomText
  - Magnifier

Testing

- Automated Tools
  - Lighthouse
  - WAVE
  - ARC Toolkit
- Color Tools
  - WebAim Color Contrast
  - WebAim Link Color Contrast
- Other Helpful Tools
  - HeadingMap
  - Color Oracle
  - NerdeFocus
  - Visual Aria
  - Silktide Website Accessibility Simulator

Users

making people with disabilities the largest minority group in the world.

There are a huge range of disabilities, which can be divided roughly into four categories:

- *Visual* - These users can benefit from the use of screen readers, screen magnification, controlling screen contrast, or braille display.
- *Auditory* - These users can benefit from captioning, transcripts or sign language video.
- *Motor* - These users can benefit from a range of assistive technologies for motor impairments: voice recognition software, eye tracking, single-switch access, head wand, sip and puff switch, oversized trackball mouse, adaptive keyboard or other assistive technologies.
- *Cognitive* - These users can benefit from supplemental media, structural organization of content, clear and simple writing.

Check out the following links from WebAim to understand from users:

- Web Accessibility Perspectives: Explore the Impact and Benefits for Everyone
- Stories of Web Users

Edit this page on GitHub

Previous
Performance

Next
Security