Accessibility in Angular

Contents

The web is used by a wide variety of people, including those who have visual or motor impairments. A variety of assistive technologies are available that make it much easier for these groups to interact with web-based software applications. In addition, designing an application to be more accessible generally improves the user experience for all users.

For an in-depth introduction to issues and techniques for designing accessible applications, see the Accessibility section of the Google's Web Fundamentals.

This page discusses best practices for designing Angular applications that work well for all users, including those who rely on assistive technologies.

> For the sample app that this page describes, see the live example / download example.

Accessibility attributes

Building accessible web experience often involves setting ARIA attributes to provide semantic meaning where it might otherwise be missing. Use attribute binding template syntax to control the values of accessibility-related attributes.

When binding to ARIA attributes in Angular, you must use the `attr.` prefix, as the ARIA specification depends specifically on HTML attributes rather than properties of DOM elements.

```
<!-- Use attr. when binding to an ARIA attribute -->
<button [attr.aria-label]="myActionLabel">...</button>
```

Note that this syntax is only necessary for attribute *bindings*. Static ARIA attributes require no extra syntax.

```
<!-- Static ARIA attributes require no extra syntax -->
<button aria-label="Save document">...</button>
```

NOTE:

> By convention, HTML attributes use lowercase names (`tabindex`), while properties use camelCase names (`tabIndex`).
>
> See the Binding syntax guide for more background on the difference between attributes and properties.

Angular UI components

The Angular Material library, which is maintained by the Angular team, is a suite of reusable UI components that aims to be fully accessible. The Component Development Kit (CDK) includes the `a11y` package that provides tools to support various areas of accessibility. For example:

- `LiveAnnouncer` is used to announce messages for screen-reader users using an `aria-live` region. See the W3C documentation for more information on aria-live regions.
- The `cdkTrapFocus` directive traps Tab-key focus within an element. Use it to create accessible experience for components like modal dialogs, where focus must be constrained.

For full details of these and other tools, see the Angular CDK accessibility overview.

Augmenting native elements

Native HTML elements capture a number of standard interaction patterns that are important to accessibility. When authoring Angular components, you should re-use these native elements directly when possible, rather than re-implementing well-supported behaviors.

For example, instead of creating a custom element for a new variety of button, you can create a component that uses an attribute selector with a native `<button>` element. This most commonly applies to `<button>` and `<a>`, but can be used with many other types of element.

You can see examples of this pattern in Angular Material: `MatButton`, `MatTabNav`, `MatTable`.

Using containers for native elements

Sometimes using the appropriate native element requires a container element. For example, the native `<input>` element cannot have children, so any custom text entry components need to wrap an `<input>` with additional elements. While you might just include the `<input>` in your custom component's template, this makes it impossible for users of the component to set arbitrary properties and attributes to the input element. Instead, you can create a container component that uses content projection to include the native control in the component's API.

You can see `MatFormField` as an example of this pattern.

Case study: Building a custom progress bar

The following example shows how to make a simple progress bar accessible by using host binding to control accessibility-related attributes.

- The component defines an accessibility-enabled element with both the standard HTML attribute `role`, and ARIA attributes. The ARIA attribute `aria-valuenow` is bound to the user's input.

src/app/progress-bar.component.ts

```
import { Component, Input } from '@angular/core';

/**
 * Example progressbar component.
 */
@Component({
  selector: 'app-example-progressbar',
  template: `<div class="bar" [style.width.%]="value"></div>`,
  styleUrls: ['./progress-bar.component.css'],
  host: {
    // Sets the role for this component to "progressbar"
    role: 'progressbar',

    // Sets the minimum and maximum values for the progressbar role.
    'aria-valuemin': '0',
    'aria-valuemax': '100',

    // Binding that updates the current value of the progressbar.
    '[attr.aria-valuenow]': 'value',
  }
})
export class ExampleProgressbarComponent  {
  /** Current value of the progressbar. */
  @Input() value = 0;
}
```

- In the template, the `aria-label` attribute ensures that the control is accessible to screen readers.

src/app/app.component.html

```
<label>
  Enter an example progress value
  <input type="number" min="0" max="100"
      [value]="progress" (input)="progress = $event.target.value">
</label>

<!-- The user of the progressbar sets an aria-label to communicate what the
progress means. -->
<app-example-progressbar [value]="progress" aria-label="Example of a progress
bar">
</app-example-progressbar>
```

Routing and focus management

Tracking and controlling focus in a UI is an important consideration in designing for accessibility. When using Angular routing, you should decide where page focus goes upon navigation.

To avoid relying solely on visual cues, you need to make sure your routing code updates focus after page navigation. Use the `NavigationEnd` event from the `Router` service to know when to update focus.

The following example shows how to find and focus the main content header in the DOM after navigation.

```
router.events.pipe(filter(e => e instanceof NavigationEnd)).subscribe(() => {
  const mainHeader = document.querySelector('#main-content-header')
  if (mainHeader) {
    mainHeader.focus();
  }
});
```

In a real application, the element that receives focus will depend on your specific application structure and layout. The focused element should put users in a position to immediately move into the main content that has just been routed into view. You should avoid situations where focus returns to the `body` element after a route change.

Additional resources

- Accessibility - Google Web Fundamentals
- ARIA specification and authoring practices
- Material Design - Accessibility
- Smashing Magazine
- Inclusive Components
- Accessibility Resources and Code Examples
- W3C - Web Accessibility Initiative
- Rob Dodson A11ycasts
- Codelyzer provides linting rules that can help you make sure your code meets accessibility standards.

Books

- "A Web for Everyone: Designing Accessible User Experiences", Sarah Horton and Whitney Quesenbery
- "Inclusive Design Patterns", Heydon Pickering

More on accessibility

You may also be interested in the following:

- Audit your Angular app's accessibility with codelyzer.