Accessibility for teams

By Rob Dodson

Rob is a contributor to Web**Fundamentals**

Making your site more accessible can be a daunting task. If you are approaching accessibility for the first time, the sheer breadth of the topic can leave you wondering where to start. After all, working to accommodate a diverse range of abilities means there are a correspondingly diverse range of issues to consider.

Remember, accessibility is a team effort. Every person has a role to play. This article outlines criteria for each of the major disciplines (project manager, UX designer, and developer) so that they can work to incorporate accessibility best practices into their process.

Project manager

An overriding goal for any project manager is to try to include accessibility work in every milestone; making sure it's just as much a priority as other topics like performance, and user experience. Below are a few checklist items to keep in mind when working through your process.

- Make accessibility training available to the team.
- Identify critical user journeys in the site or application.
- Try to incorporate an accessibility checklist into the team process.
- Where possible, evaluate the site or application with user studies.

Accessibility training

There are a number of great free resources for learning about web accessibility. Setting aside time for your team to study the topic can make it easier to include accessibility early in the process.

Some resources provided by Google include:

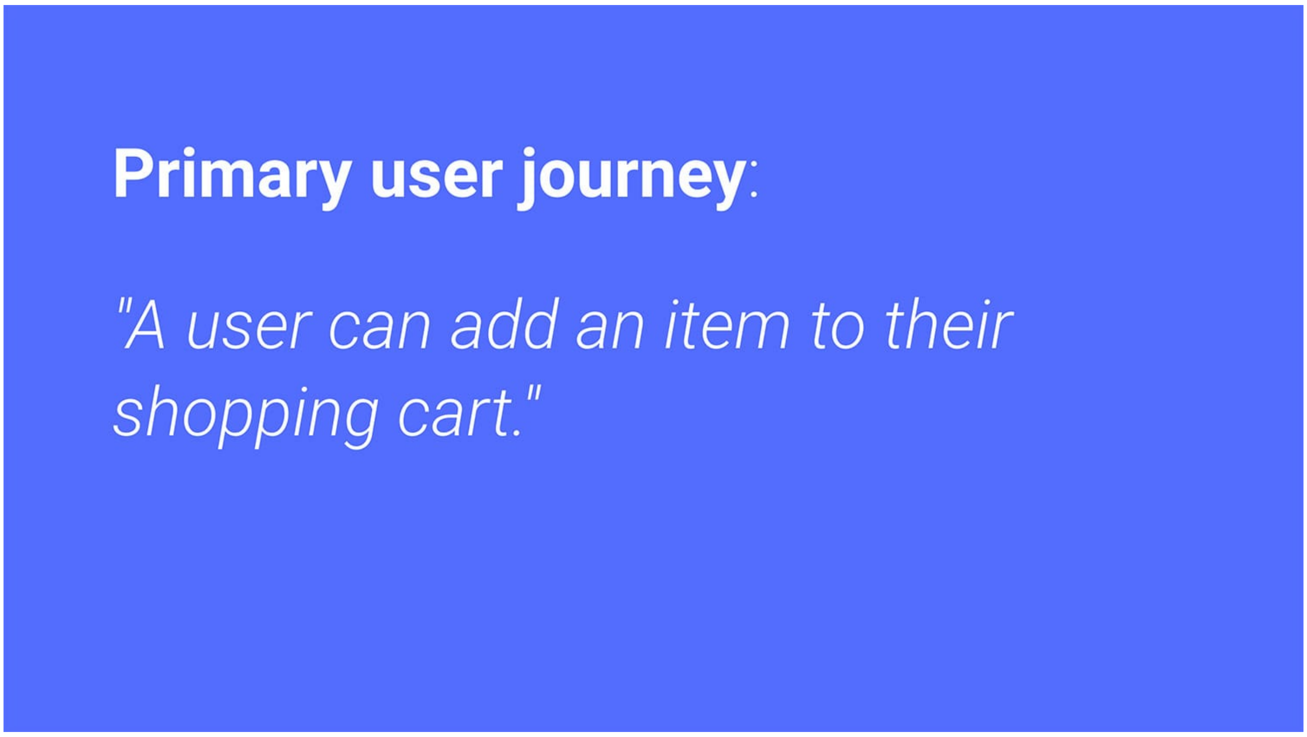
Web Accessibility by Google — a multi-week interactive training course.

Accessibility Fundamentals — written accessibility guides and best practices.

Material Guidelines: Accessibility — a set of UX best practices for inclusive design.

Identifying critical user journeys

Every application has some primary action that the user needs to take. For example, if you're building an e-commerce app, then every user will need to be able to add an item to their shopping cart.



Primary user journey:

"A user can add an item to their shopping cart."

Some actions may be of secondary importance, and perhaps only performed occasionally. For example, changing your avatar photo is a nice feature, but may not be critical for every experience.

Identifying the primary and secondary actions in your application will help you prioritize the accessibility work ahead. Later, you can combine these actions with an accessibility checklist to keep track of your progress and avoid regressions.

Incorporating an accessibility checklist

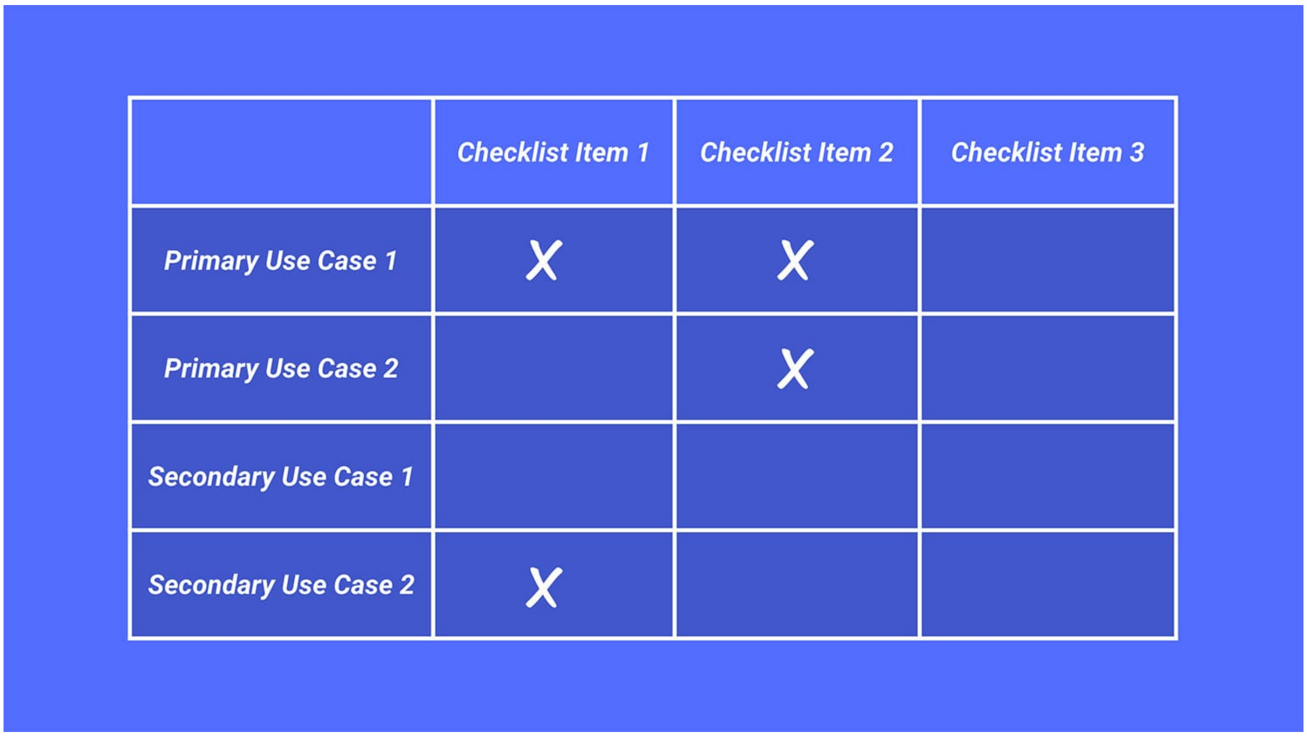
The topic of accessibility is quite broad, so having a checklist of important areas to consider can help you make sure you're covering all of your bases.

There are a number of accessibility checklists out there, a few industry examples include:

WebAIM WCAG Checklist

Vox Accessibility Guidelines

With a checklist in hand, you can look over your primary and secondary actions to start to triage what work still needs to be done. You can get pretty tactical about this process and even build a matrix of primary and secondary actions and determine for each step in those processes, whether there are any missing accessibility bits.



	Checklist Item 1	*Checklist Item 2*	*Checklist Item 3*
Primary Use Case 1	X	X	
Primary Use Case 2		X	
Secondary Use Case 1			
Secondary Use Case 2	X		

Evaluating with user studies

Nothing beats sitting down with actual users and observing them as they try to use your app. If you're retrofitting accessibility into an existing experience, this process can help you quickly identify areas that need improvement. And if you're starting a new project, early user studies can help you avoid spending too much time developing a feature that is difficult to use.

Aim to incorporate feedback from as diverse of a user population as possible. Consider users who primarily navigate with the keyboard, or rely on assistive technology like screen readers or screen magnifiers.

UX designer

Because people tend to design using their own biases, if you don't have a disability and don't have colleagues with disabilities, you might be unintentionally designing for only some of your users. As you work, ask yourself "what are all the types of users who might rely on this design?" Here are some techniques you can try to make your process more inclusive.

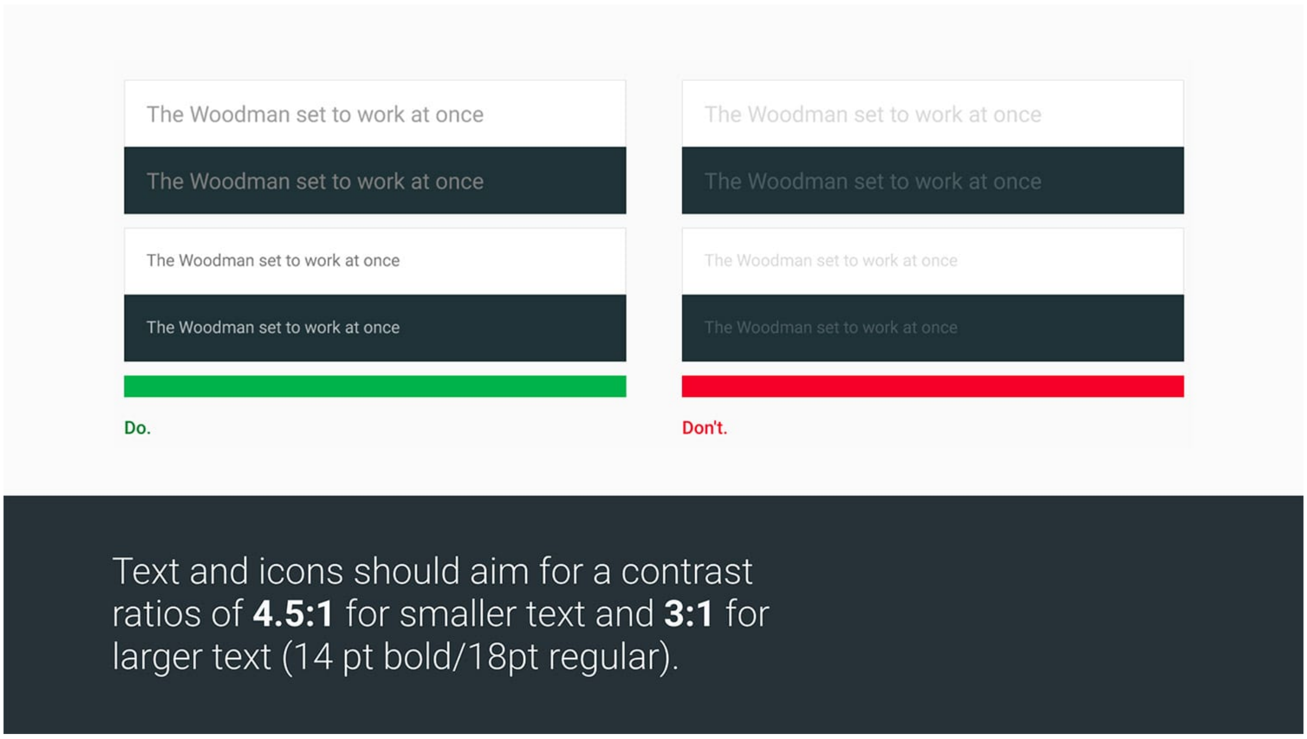
- Content has sufficient color contrast.
- The tab order is defined.
- Controls have accessible labels.
- There are multiple ways to interact with the UI.

Content has good color contrast

The primary goal of most sites is to convey some information to the user, either through written text or images. However, if this content is low contrast, it may be difficult for some users (particularly those with a vision impairment) to read. This may negatively affect their user experience. To address this concern, aim for all text and images to have sufficient color contrast.

Contrast is measured by comparing the luminance of a foreground and background color. For smaller text (anything below 18pt or 14pt bold) a minimum ratio of 4.5:1 is recommended. For larger text, this ratio can be adjusted to 3:1.

In the image below, the text on the left hand side meets these contrast minimums, whereas the text on right hand side is low contrast.


The Woodman set to work at once
The Woodman set to work at once
The Woodman set to work at once
The Woodman set to work at once
Do.
The Woodman set to work at once
The Woodman set to work at once
The Woodman set to work at once
The Woodman set to work at once
Don't.
Text and icons should aim for a contrast ratios of 4.5:1 for smaller text and 3:1 for larger text (14 pt bold/18pt regular).

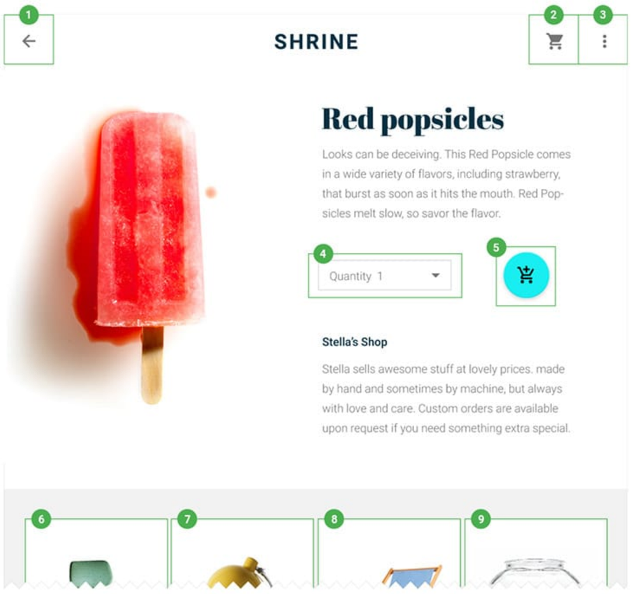
There are a number of tools for measuring color contrast, such as Google's Material Color Tool, Lea Verou's Contrast Ratio app, and Deque's aXe.

Tab order is defined

The tab order is the order in which elements receive focus as the user presses the tab key. For users who navigate primarily with a keyboard, the tab key is their primary means of reaching everything on screen. Think of it like their mouse cursor.

Ideally the tab order should follow the reading order and flow from the top of the page to the bottom, with more important items appearing higher up in the order. This makes it more efficient for anyone using a keyboard to quickly reach these items.

Tab order should **follow the order of the visual layout**, from the top to the bottom of the screen. It should traverse **from the most important to the least important** item.


SHRINE

Red popsicles

Looks can be deceiving. This Red Popsicle comes in a wide variety of flavors, including strawberry, that burst as soon as it hits the mouth. Red Popsicles melt slow, so savor the flavor.

Quantity 1

Stella's Shop

Stella sells awesome stuff at lovely prices. made by hand and sometimes by machine, but always with love and care. Custom orders are available upon request if you need something extra special.

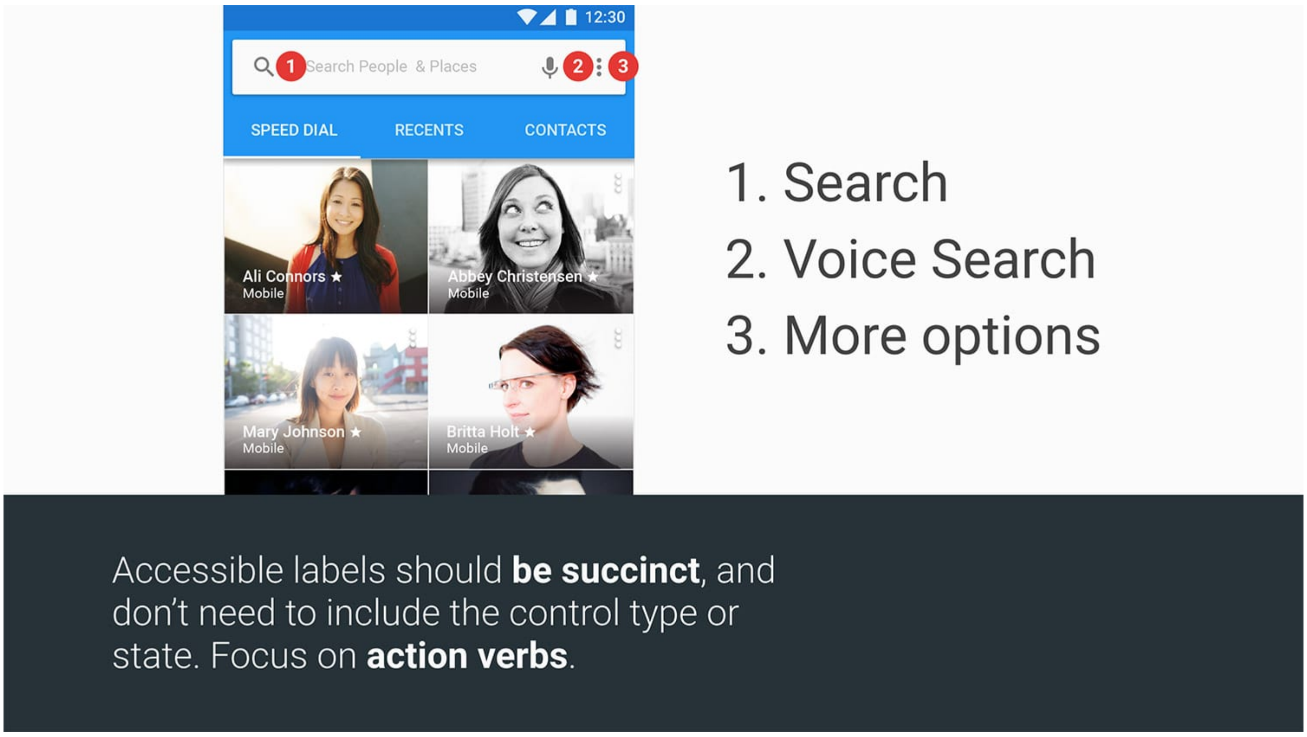
The mock interface above is numbered to show the tab order. Creating a mock like this can help by identifying the intended tab order. This can then be shared with the developers and QA testers to make sure it is properly implemented and tested.

Controls have accessible labels

For users of assistive technology like screen readers, labels provide information that would otherwise be visual only. For example a search button that's just a magnifying glass icon can have an accessible label of "Search" to help fill in the missing visual affordance.

Here are a few simple suggestions to follow when designing accessible labels:

- Be succinct - It can be tedious to listen to long descriptions.
- Try not to include control type or state - If the control is coded properly then a screen reader will announce this automatically.
- Focus on action verbs - Use "search" not "magnifying glass".



1. Search
2. Voice Search
3. More options

Accessible labels should **be succinct**, and don't need to include the control type or state. Focus on **action verbs**.

You might consider creating a mock with all of your controls labeled. This can be shared with your development team and QA team for implementation and testing.

Multiple ways to interact with and understand UI

It's easy to assume that all users interact with the page primarily using a mouse. When designing, consider how someone will interact with a control using a keyboard instead.

Plan your focus states! This means determining what a control will look like when the user focuses it with tab or presses the arrow keys. It's useful to have these states planned early, rather than trying to shoehorn them into the design later.

Finally, for any point of interaction, you want to make sure that the user has multiple ways of understanding the content. Try not to use color alone to convey information, as these subtle cues may be missed by a user with a color vision deficiency. A classic example is an invalid text field. Instead of just a red underline to signify a problem, also consider adding some helper text. That way you're covering more bases and increasing the likelihood that a user will notice the issue.

Developer

The developer's role is where focus management and semantics combine to form a robust user experience. Below are a few items a developers can keep in mind as they're working on their site or application:

- The tab order is logical.
- Focus is properly managed and visible.
- Interactive elements have keyboard support.
- ARIA roles and attributes are applied as needed.
- Elements are properly labeled.
- Testing is automated.

Logical tab order

Native elements like `input`, `button`, and `select` get opted into the tab order for free and are automatically focusable with the keyboard. But not all elements receive this same behavior! In particular, generic elements like `div`, and `span`, are not opted into the tab order. This means if you use a `div` to create an interactive control, you'll need to do additional work to make it keyboard accessible.

Two options are:

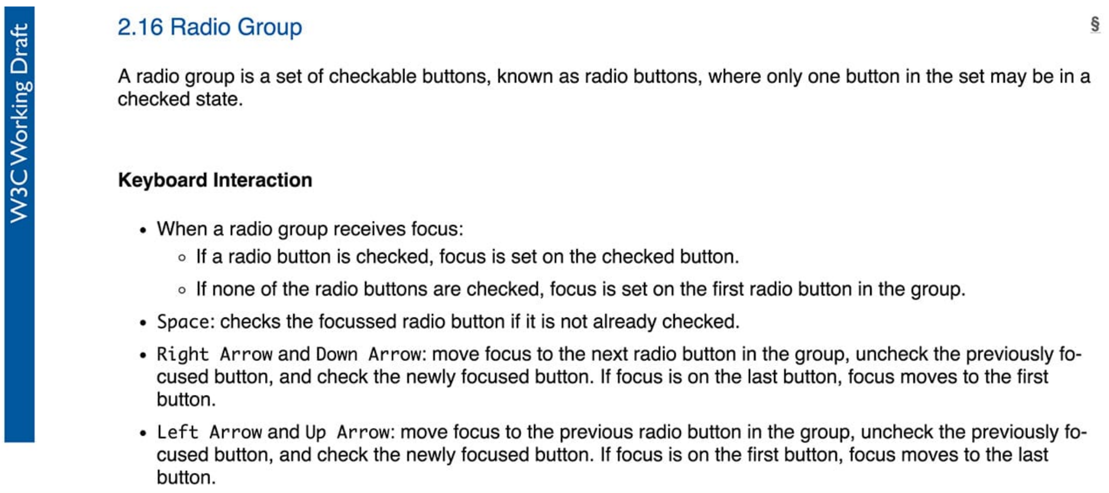
- Give the control a `tabindex="0"`. This will at least make it focusable, though you'll likely need to do additional work to add support for keypresses.
- Where possible, consider using a `button` instead of a `div` or `span` for any button-like control. The native `button` element is very easy to style and gets keyboard support for free.

Managing focus

When you change the content of the page, it's important to direct the user's attention by moving focus. A classic example of when this technique is useful is when opening a modal dialog. If a user relying on a keyboard presses a button to open a dialog and their focus *is not* moved into the dialog element, then their only course of action is to tab through the entire site until they eventually find the new control. By moving focus into the new content as soon as it appears, you can improve the efficiency of these users' experiences.

Keyboard support for interactive elements

If you're building a custom control like a carousel or dropdown, then you'll need to do some additional work to add keyboard support. The ARIA Authoring Practices Guide is a useful resource which identifies various UI patterns and the kinds of keyboard actions they are expected to support.



2.16 Radio Group

A radio group is a set of checkable buttons, known as radio buttons, where only one button in the set may be in a checked state.

Keyboard Interaction

- When a radio group receives focus:
  - If a radio button is checked, focus is set on the checked button.
  - If none of the radio buttons are checked, focus is set on the first radio button in the group.
- `Space`: checks the focussed radio button if it is not already checked.
- `Right Arrow` and `Down Arrow`: move focus to the next radio button in the group, uncheck the previously focused button, and check the newly focused button. If focus is on the last button, focus moves to the first button.
- `Left Arrow` and `Up Arrow`: move focus to the previous radio button in the group, uncheck the previously focused button, and check the newly focused button. If focus is on the first button, focus moves to the last button.

ARIA Authoring Practices. An excellent **cheat sheet** for component accessibility!

To learn more about adding keyboard support to an element, take a look at the roving tabindex section in Google's Accessibility Fundamentals docs.

ARIA roles and attributes are applied as needed

Not only do custom controls need proper keyboard support, they also need proper semantics. After all, a `div`, semantically, is just a generic grouping container. If you're using a `div` as the basis for your dropdown menu, you'll need to rely on ARIA to layer in additional semantics so the control type can be conveyed to assistive technology. Here again the ARIA Authoring Practices Guide can help by identifying which roles, states, and properties you should be using. As an added bonus, many of the explanations in the ARIA guide also come with sample code!

Labeling elements

For labeling native inputs, you can use the built-in `<label>` element as described on MDN. Not only will this help you create an onscreen visual affordance, but it also gives the input an accessible name in the accessibility tree. This name is then picked up by assistive technology (like a screen reader) and announced to the user.

Unfortunately `<label>` *does not* support giving an accessible name to custom controls (like ones created using Custom Elements or out of simple divs and spans). For these kinds of controls you'll need to use the `aria-label` and `aria-labelledby` attributes.

Automated testing

Being lazy can be good, especially when it comes to testing. Wherever possible seek to automate your accessibility tests so you don't have to do everything manually. There are a number of great industry testing tools that exist today to make it easy and fast to check for common accessibility issues:

aXe, created by Deque systems, is available as a Chrome extension and a Node module (good for continuous integration environments). This short A11ycast explains a few different ways to incorporate aXe into your development process.

Lighthouse is Google's open source project for auditing the performance of your Progressive Web Apps. In addition to checking if your PWA has support for things like Service Worker and a Web App Manifest, Lighthouse will also run a series of best practice tests, including tests for accessibility issues.

Conclusion

Accessibility is a team effort. Everyone has a role to play. This guide has laid out a few key items that each team member can use to quickly ramp up on the subject and hopefully improve the overall experience of their app.

To learn more about accessibility, be sure to check out our free Udacity course and browse the accessibility docs available here on Web Fundamentals.