

- What isn't ARIA?
- How does ARIA work?
- Why ARIA?
- Menu bar example
  - Supporting mouse clicker people
  - Making our menu bar keyboard accessible
  - Screen reader access to our menu bar
- Dealing with bugs
  - Keyboard bugs (HTML!)
  - ARIA bugs: why are there so many?
- Summary
- Addendum 1: Additional Resources
  - Hybrid reference with keyboard info and code examples
- Addendum 2: What is ARIA most used for?
- Addendum 3: What's an Accessibility API?

ARIA lets web authors create an alternative reality, seen only by screen readers

Sometimes it's necessary to expand on the truth or even downright "lie" to screen readers about what's happening in web content. For example, "focus is really over here!" or "this is really a slider!". It's like adding magical sticky notes on top of tools and widgets on your workbench. These magical sticky notes make everyone believe what's written on them.

Whenever a magical sticky note exists, it either overrides our belief about what each tool is, or something about the tool. Example: "this thing over here is a glue gun!". Even though it's still actually an empty blue box sitting there on the workbench, the magical sticky note will make us see it is a glue gun. We can also add, "and it is 30% full!". The screen reader will now report that there is a 30% full glue gun there.

The web equivalent to this is to take a plain box element (a div) with an image inside of it, and use ARIA to say it's a slider at value 30 out of 100.

What isn't ARIA?

ARIA does not affect the appearance of a web page, or the behavior for a mouse or keyboard user. Only users of assistive technologies will notice any difference from ARIA. Web developers can add any arbitrary ARIA without affecting users that aren't running an assistive technology.

You read it right: ARIA doesn't actually do anything to keyboard focus or tab order. That's all done in HTML, sometimes tweaked with bits of JavaScript.

How does ARIA work?

Browsers are asked by a screen reader or other assistive technology for information about each element. When ARIA is present on an element, the browser takes in the information and changes what it tells the screen reader about that element.

Why ARIA?

Why would we ever want to lie to our users!?

Let's say the local web store doesn't sell all the widgets we need. But, we are MacGyver, dammit. We can just invent our own widgets from other widgets! FWIW, the MacGyver's seven most used things are Swiss Army knives, gum, shoe strings, matches, paper clips, birthday candles, and duct tape. He uses them to make bombs and other things that aren't just laying around. This is pretty similar to a web author who needs to make a menu bar. Menu bars are so useful you would think they would be part of HTML, but they aren't. Oh well! You didn't think authors would be happy with links and buttons did you? So the author will cobble one together using their favorite tools: divs, images, style, click handlers, keypress handlers, spit, and ARIA.

Sometimes, rather than using ARIA to the max, we just use it as an enhancement. It can be useful to sprinkle a little ARIA on some HTML that already basically works. For example, we might want a form control to point to an error message alert that relates to some invalid input. Or we might want to indicate that a textbox is for searching. These little tweaks can make ordinary websites more usable with a screen reader.

Menu bar example

Supporting mouse clicker people

Let's make a menu bar together. We show a bunch of items in generic box elements called divs. Any time our user clicks on a div, it executes the corresponding command. Cool, it works for mouse clicker people!

Next we make it look pretty. We use CSS, i.e. styles, lining things up nicely and putting visual outlines around them. We make it look enough like other menu bars that sighties intuitively know that it's a menu bar and how to use it. Our menu bar even uses a different background color on any item that the mouse is over, giving the user some helpful visual feedback.

Some menu items are parents. They spawn child submenus. Whenever the user hovers on one of these we start an animation that slides out the child submenu.

This, of course, is all pretty inaccessible, as is the usual case for many things on the web, largely because the HTML standards wizards didn't add everything a web author needs. And even if they did, web authors would always want to invent their own special version anyway.

Making our menu bar keyboard accessible

As a first step toward accessibility, let's add keyboard accessibility. This part only uses HTML, and not ARIA. Remember that ARIA does not affect core aspects such as appearance, mouse, or keyboard for users without assistive technologies.

Just like a web page can respond to the mouse, it can also respond to the keyboard. Our JavaScript will listen to all keystrokes that occur and decide if the keypress is useful. If not, it throws it back to the page like a fish that's too small to eat. Our rules are something like:

- If the user presses an arrow key, let's look at our own internal menu bar blueprints and decide what the new active menu item should be. We will clear any current highlights and highlight the new menu item so the sighted user visually knows where they are. The web page should then call `event.preventDefault()` to prevent the browser from performing the usual action (scrolling the page, in this case).
- If the user presses the `Enter` key, we can treat it just like a click, and perform the appropriate action (or even open another menu).
- If the user presses a key that should do something else, don't eat that! Throw it back to the page as nature intended. For example, our menu bar doesn't need the `Tab` key, so throw it back! This is hard to get right, and authors often mess it up. For example, the menu bar needs arrow keys, but not `Alt+Arrow` or `Command`+Arrow. Those are shortcuts for moving to the previous/next page in the web history of your browser tab. If the author isn't careful, the menu bar will eat those. This kind of bug happens a lot, and we haven't even started with ARIA yet!

Screen reader access to our menu bar

Our menu bar was created with duct tape and divs. As a result, a screen reader has no idea what any of it is. The background color for the active item is just a color. The menu item divs are just plain objects with no particular meaning. Consequently, a user of our menu bar doesn't get any instructions about what keys to press or what item they're on.

But that's no fair! The menu bar acts just fine for the sighted user.

ARIA to the rescue. ARIA lets us pretend to the screen reader that focus is in a menu bar. If the author does everything right, our custom menu bar will look to the screen reader just like a menu bar in a desktop application.

Our first, ahem, ARIA lie, is to use the `aria-activedescendant` attribute, and set it to the ID of the currently active menuitem, being careful to update it whenever it changes. For example, `aria-activedescendant="settings-menuitem"`. This little white lie causes the screen reader to consider our ARIA active item as the focus, which is read aloud or shown on a Braille display.

Explanation of *ancestor*, *parent*, and *descendant*

The term descendant refers to the fact that an item is contained somewhere inside of another. The opposite term is ancestor, which is to say an item is contained by ancestors. For the next container up/down, these may use the more specific terms parent/child. For example, imagine a document with a paragraph that has a link inside. The link's parent is a paragraph, but it also has the document as an ancestor. Conversely, the document may have many paragraph children, each with links. The links are all descendants of the grandparent document.

Back to `aria-activedescendant`. By using it to point from the focused menu bar to a specific menu item, the screen reader now knows where the user has moved, but nothing else about the object. What is this div thing anyway? That's where the role attribute comes in. We use `role="menubar"` on the containing element for the entire thing, then we use `role="menu"` on groups of items, and `role="menuitem"` on … drumroll … the individual menu items.

And what if the menuitem can lead to a child menu? The user needs to know that right? For a sighted user, there might be a little picture of a triangle at the end of the menu, but the screen reader doesn't know how to automatically read images, at least at this point. We can add `aria-expanded="false"` on each expandable menuitem to indicate that 1) there is something that can be expanded, and 2) it currently is not expanded. As an added touch the author should put `role="none"` on the img triangle to indicate it's for prettification purposes only. This prevents the screen reader from saying anything about the image that would be redundant at best and possibly annoying.

Dealing with bugs

Keyboard bugs (HTML!)

Although keyboard access is a part of core HTML, authors mess it up all the time, either because they don't use keyboard navigation all that much, or because there is much nuance to get right.

Examples of bugs:

- A checkbox uses spacebar to toggle, but the author forgot to call `preventDefault()`. Now the spacebar will both toggle the checkbox and page down, which is the default browser behavior for spacebar.
- An ARIA modal dialog wants to trap tab navigation inside of it, and the author forgets to specifically allow `Control+Tab` through to the browser. Now, `Control+Tab` just navigates within their dialog, and doesn't switch tabs in the browser as it should. Ugh.
- An author creates a selection list, and implements up/down, but does not implement home/end/pageup/pagedown or first letter navigation.

Authors should follow known patterns. Check out the Resources section for more information.

For pure keyboard access issues, it's useful to also try without a screen reader, or with virtual browser mode off. Screen readers are not generally necessary to discover keyboard bugs, and keyboard access is actually implemented with HTML, not ARIA. After all, ARIA doesn't affect basic stuff like the keyboard or mouse behavior, it only lies to the screen reader about what's in the web page, what's currently focused, etc.

Keyboard bugs are almost always a bug in the web content, specifically in their HTML and JavaScript, not in ARIA.

ARIA bugs: why are there so many?

There are many, many places where authors can get ARIA wrong, and each will lead to either complete breakage or subtle differences. The subtle ones are probably worse, because the author won't catch most of them before publishing.

After all, unless the author is an experienced screen reader user, something is going to go wrong in the ARIA. In our menu bar example, the author could think the "option" role was to be used when "menuitem" was correct. They could forget to use `aria-expanded`, forget to set and clear `aria-activedescendant` at the right times, or forget to have a menu bar containing the other menus. And what about menu item counts? Usually menu items are presented by screen readers with something like "item 3 of 5" so that the user knows where they are. This is generally counted automatically by the browser, but in some cases, and in some browser - screen reader combinations, the wrong numbers might be computed, and the author would need to override these numbers with `aria-posinset` and `aria-setsize`.

And this is just menu bars. Think of how many kinds of widgets there are. Glance at the ARIA spec or authoring practices if you like. For each pattern, there are a dozen ways ARIA could be misused. ARIA relies on authors to know what they're doing. What could possibly go wrong, given that most authors are not screen reader users?

In other words, it is 100 percent necessary for actual screen reader users to try ARIA widgets before they're considered shippable. There's too much nuance. Ideally everything would be tried with several different browser-screen reader combinations, because of the numerous implementation quirks, in addition to a few incomplete implementations.

Summary

In summary, ARIA magic can be used to override or add to anything and everything that the HTML says. It can be used to do little fine changes to the accessibility presentation, or to create an entire experience. This is why ARIA is both incredibly powerful and yet dangerous in the hands of our friendly local web authors who don't generally use screen readers themselves.

ARIA is just a straightforward truth override markup layer. When a screen reader asks what's happening, if ARIA exists, they get the ARIA version of the truth instead of the real underlying truth.

Addendum 1: Additional Resources

Hybrid reference with keyboard info and code examples

- W3C's ARIA Authoring Practices: this documents the important keyboard navigation characteristics of each example and provides working JS/CSS/ARIA code. The examples are focused on what works today, and do not cover mobile.

Addendum 2: What is ARIA most used for?

Because ARIA can replace or supplement small or large truths, generally useful for saying stuff that the screen reader cares about.

Here are some common uses of ARIA.

- Special widgets that don't exist in HTML, like a menu bar, autocomplete, tree, or spreadsheet
- Widgets that exist in HTML, but the author invented their own anyway, possibly because they needed to tweak the behavior or appearance of the normal widget. For example, an HTML `<input type="range">` element is basically a slider, but authors want to make it look different. For most things, CSS can be used, but for `input type="range"`, CSS is awkward. An author can make their own slider, and use `role="slider"` on it with `aria-valuenow` to say what the current value is.
- Live regions tell screen readers "in this area of the page, anything that changes is worth telling the user about."
- Landmarks (HTML has equivalents now). These are somewhat like headings, in that they help screen reader users find what they want faster. However, they're different in that they contain the entire related area. Like, "this container is the main area of the page" and "this container over here is a navigation panel".

Addendum 3: What's an Accessibility API?

An accessibility API is how a screen reader or other AT knows what's in the page and what's happening right now. Examples include MSAA, IA2, and UIA. And that's just Windows! There are two parts to an accessibility API:

- A "tree" of objects that represents a container hierarchy. These are like Russian nesting dolls, but each doll can contain multiple other dolls. For example, a document can contain a bunch of paragraphs, and a paragraph can have text, images, links, boldface, etc. Each item in the object tree can have properties like a role (what am I?), a name/label, a user-entered value, a description, as well as boolean states like focusable, focused, required, checked. ARIA can override any of these properties. The screen reader uses the tree to help the user navigate in virtual buffer mode, like "go to the next heading please".
- A series of events that occur describing changes to the tree, like "focus is now over here!". The screen reader uses the events to tell the user what has just happened. When important HTML or ARIA markup changes, an event is fired to tell the screen reader that something changed.

Usually authors just use HTML, which maps nicely to these accessibility APIs. When HTML is not enough, ARIA is used and the browser overrides the HTML semantics before sending the object tree or events to the screen reader.

□□□ #3
□ □□□ □□□ 2026-10-31 21:37:12 UTC
□ cocobook-sync □□□ 2026-11-04 17:36:31 UTC