18 de febrero de 2021

Carlos Palacios Alonso

I.E.S Ribera de Castilla, 1º DAW

Práctica accesibilidad

Diseño de Interfaces Web

Índice

[1 Conceptos Básicos 2](#_Toc64649784)

[2 Diversidad de Usuarios 2](#_Toc64649785)

[3 Puntos Clave 3](#_Toc64649786)

[3.1 Foco. 3](#_Toc64649787)

[3.2 Semántica. 5](#_Toc64649788)

[3.3 Estilos (Aria). 7](#_Toc64649789)

# Conceptos Básicos

Broadly speaking, when we say a site is accessible, we mean that the site's content is available, and its functionality can be operated, by literally anyone.

Accessibility, then, refers to the experience of users who might be outside the narrow range of the "typical" user, who might access or interact with things differently than you expect. Specifically, it concerns users who are experiencing some type of impairment or disability — and bear in mind that that experience might be non-physical or temporary.

[Web Content Accessibility Guidelines (WCAG) 2.0](https://www.w3.org/TR/WCAG20/), a set of guidelines and best practices put together by accessibility experts to address what "accessibility" means in a methodical way.

WCAG is organized around four principles often called by the acronym POUR:

**Perceivable**: Can users perceive the content? This helps us keep in mind that just because something is perceivable with one sense, such as sight, that doesn't mean that all users can perceive it.

**Operable**: Can users use UI components and navigate the content? For example, something that requires a hover interaction cannot be operated by someone who can't use a mouse or touch screen.

**Understandable**: Can users understand the content? Can users understand the interface and is it consistent enough to avoid confusion?

**Robust**: Can the content be consumed by a wide variety of user agents (browsers)? Does it work with assistive technology?

# Diversidad de Usuarios

When learning about accessibility, it helps to have an understanding of the diverse range of users in the world and the kinds of accessibility topics that affect them.

Visual Impairments.

Hoy en día es inusual encontrarnos a alguien que no pueda ver por completo, aun así, hay una buena posibilidad de encontrarnos con personas que ven borroso o que tienen poca visiblidad. Tanto para unos como otros, son necesarias diferentes herramientas para poder navegar.

Lector de pantalla. Literalmente nos lee la pantalla.

Lector braille. Muestra la información del sitio web en braille.

Texto a voz. Transforma el texto a voz para poderlo oir.

Lupa. Aumenta el tamaño de una parte de la pantalla

Zoom del buscador. Aumenta el tamaño de las fuentes.

Modo de contraste alto. Como su nombre indica.

Motor Impairments

This group ranges all the way from those who would prefer not to use a mouse, because perhaps they've had some RSI or something and find it painful, to someone who may be physically paralyzed and have limited range of motion for certain parts of their body.

Este grupo puede utilizar un teclado, un switch device, control de voz o un dispositivo de eye-tracking para interactuar con el ordenador.

Hearing Impairments

This group can range from the profoundly deaf through to hard-of-hearing. And much like eyesight, our hearing tends to degrade with age. Many of us use common affordances like hearing aids to help us.

For hearing-impaired users we need to make sure that we're not relying on sound, so making sure to use things like video captions and transcripts, and providing some kind of alternative, if sound is part of the interface.

Cognitive Impairments

There's a range of cognitive conditions like ADD, Dyslexia, and Autism, which can mean that people want or need to access things differently. The accommodations for these groups are naturally extremely diverse, but we definitely find some overlap with other areas, like using zoom functionality to make reading or concentrating easier. Also, these users may find that really minimal design works best because it minimizes distraction and cognitive load.

# Puntos Clave

## Foco.

Introduccion

Focus determines where keyboard events go in the page at any given moment. For instance, if you focus a text input field and begin typing, the input field receives the keyboard events and displays the characters you type. While it has focus, it will also receive pasted input from the clipboard.

Some users operate their computer almost entirely with the keyboard or other input device. For those users, focus is critical; it's their primary means of reaching everything on the screen. For that reason, the Web AIM checklist states in section 2.1.1 that [all page functionality should be available using the keyboard](https://webaim.org/standards/wcag/checklist#sc2.1.1), unless it's something you cannot do with a keyboard, such as freehand drawing.

As a user, you can control which element is currently focused using Tab, Shift+Tab, or the arrow keys

Built-in interactive HTML elements like text fields, buttons, and select lists are implicitly focusable,

But not all elements are focusable; paragraphs, divs, and various other page elements are not focused as you tab through the page, and that's by design.

El orden de DOM importa

For example, you might have three button elements, one after the other in the DOM. Pressing Tab focuses each button in order. Try clicking the code block below to move the focus navigation start point, then press Tab to move focus through the buttons.

However, it's important to note that, using CSS, it's possible to have things exist in one order in the DOM but appear in a different order on screen. For example, if you use a CSS property like float to move one button to the right, the buttons appear in a different order on screen. But, because their order in the DOM remains the same, so does their tab order.

Offscreen content

What if you have content that isn't currently displayed, yet still needs to be in the DOM, such as a responsive side-nav? When you have elements like this that receive focus when they're off screen, it can seem as if the focus is disappearing and reappearing as the user tabs through the page — clearly an undesirable effect.

Sometimes you need to do a bit of detective work to figure out where focus has gone. You can use document.activeElement from the console to figure out which element is currently focused.

Once you know which off screen element is being focused, you can set it to display: none or visibility: hidden, and then set it back to display: block or visibility: visible before showing it to the user.

Usando tabindex

you can use the tabindex HTML attribute to explicitly set an element's tab position.

tabindex="0": Inserts an element into the natural tab order. The element can be focused by pressing the Tab key, and the element can be focused by calling its focus() method

tabindex="-1": Removes an element from the natural tab order, but the element can still be focused by calling its focus() method

tabindex="5": Any tabindex greater than 0 jumps the element to the front of the natural tab order. If there are multiple elements with a tabindex greater than 0, the tab order starts from the lowest value that is greater than zero and works its way up. Using a tabindex greater than 0 is considered an **anti-pattern**.

This is particularly true of non-input elements like headers, images, or article titles. Adding tabindex to those kinds of elements is counter-productive. If possible, it's best to arrange your source code so the DOM sequence provides a logical tab order. If you do use tabindex, restrict it to custom interactive controls like buttons, tabs, dropdowns, and text fields; that is, elements the user might expect to provide input to.

Here's a scenario where tabindex is not only useful, but necessary. You might be building a robust single page with different content sections, not all of which are simultaneously visible. In this kind of page, clicking a navigation link might change the visible content without doing a page refresh.

When this happens, you would probably identify the selected content area, give it a tabindex of -1 so that it doesn't appear in the natural tab order, and call its focus method. This technique, called *managing focus*, keeps the user's perceived context in sync with the site's visual content.

Perhaps you're working on some new [Custom Elements](https://developers.google.com/web/fundamentals/web-components/customelements) that resemble a set of radio buttons, but with your unique take on appearance and behavior.

Roving tabindex works by setting tabindex to -1 for all children except the currently-active one.

The component then uses a keyboard event listener to determine which key the user presses; when this happens, it sets the previously focused child's tabindex to -1, sets the to-be-focused child's tabindex to 0, and calls the focus method on it.

Sometimes when you're managing focus you can get into a situation you can't get out of. Consider an autocomplete widget that tries to manage focus and captures the tab behavior, but prevents the user from leaving it until it's complete. This is called a keyboard trap,

Oddly, there are times when this behavior is actually desirable, like in a modal window. Normally, when the modal is displayed, you don't want the user to access the content behind it.

In instances like this you can implement a temporary keyboard trap to ensure that you trap focus only while the modal is displayed and then restore focus to the previously-focused item when the modal is closed.

Ver como hacer modal

## Semántica.

Introduccion

Assistive technology

Assistive technology is an umbrella term for devices, software, and tools that help any person with a disability complete a task. In the broadest sense this could be something low-tech like a crutch for walking or a magnifying glass for reading, or something high-tech like a robotic arm or image recognition software on a smartphone.

Assistive technology can include something as general as browser zoom, or as specific as a custom-designed game controller. It can be a separate physical device like a braille display, or be implemented completely in software like voice control. It can be built-in to the operating system like some screen readers, or it can be an add-on like a Chrome extension.

In the context of web development, we must consider a diverse range of technologies. People may interact with your website using a screen reader or braille display, with a screen magnifier, via voice control, using a switch device, or with some other form of assistive technology that adapts the page's default interface to create a more specific interface that they can use.

 a well-designed reader to tell you all, or at least most, of the following information about the elements it encounters.

The accessibility tree

For web browsers, there's an extra step in each direction, because the browser is in fact a platform for web apps that run inside it. So the browser needs to translate the web app into an accessibility tree, and must make sure that the appropriate events get fired in JavaScript based on the user actions that come in from the assistive technology.

But that is all the browser's responsibility. Our job as web developers is just to be aware that this is going on, and to develop web pages that take advantage of this process to create an accessible experience for our users.

We do this by ensuring that we express the semantics of our pages correctly: making sure that the important elements in the page have the correct accessible roles, states, and properties, and that we specify accessible names and descriptions. The browser can then let the assistive technology access that information to create a customized experience.

Semantics in native HTML

A browser can transform the DOM tree into an accessibility tree because much of the DOM has implicit semantic meaning. That is, the DOM uses native HTML elements that are recognized by browsers and work predictably on a variety of platforms. Accessibility for native HTML elements such as links or buttons is thus handled automatically. We can take advantage of that built-in accessibility by writing HTML that expresses the semantics of our page elements.

However, sometimes we use elements that look like native elements but aren't. For example, this "button" isn't a button at all.

When we don't use an actual button element, the screen reader has no way to know what it has landed on. Also, we would have to do the extra work [of adding tabindex](https://developers.google.com/web/fundamentals/accessibility/focus/using-tabindex) to make it usable to keyboard-only users because, as it is coded now, it can only be used with a mouse.

We can easily fix this by using a regular button element instead of a div. Using a native element also has the benefit of taking care of keyboard interactions for us

owever, for input or control elements, and visual content like images, we need to make sure that we specify a name. In fact, providing text alternatives for any non-text content is [the very first item on the WebAIM checklist](https://webaim.org/standards/wcag/checklist#g1.1).

Por ejemplo, con un checkbox, lo suyo sería asociarle un label con for.

Text alternatives for images

Images are an important component of most web pages, and are of course a particular sticking point for low-vision users. We must consider the role an image plays in a page to work out what type of text alternative it should have.

Usar alt

En una caja de búsqueda, si tenemos una imagen que es para buscar, dejar vacio alt, porque si no se hace redundante al tener ya “Search”

Navigacing content

En páginas grandes es un inconveniente tener que ir elemento por elemento de arriba abajo con el screen reader, para evitar esto, se utilizan headers ocultos, los cuales les hacemos accesibles solo para screen readers y otras tecnologías de asistencia (assistive).

Otra manera: el screen reader rotor (una manera fácil de aislar y escanear una lista de títulos de una pagina) para acceder a una lista de links en la pagina.

For example, here are some common patterns that make links hard to find.

Anchor tags without href attributes. Often used in single-page applications, these link targets cause problems for screen readers. You can read more in [this article on single-page apps](http://neugierig.org/software/blog/2014/02/single-page-app-links.html).

Buttons that are implemented with links. These cause the screen reader to interpret the content as a link, and the button functionality is lost. For these cases, replace the anchor tag with a real button and style it appropriately.

Images used as link content. Sometimes necessary, linked images can be unusable to screen readers. To guarantee that the link is properly exposed to assistive technology, make sure the image has alt attribute text.

HTML5 introduced some new elements that help define the semantic structure of the page, including header, footer, nav, article, section, main, and aside. These elements specifically provide structural clues in the page without forcing any built-in styling (which you should do with CSS anyway).

Semantic structural elements replace multiple, repetitive div blocks, and provide a clearer, more descriptive way to intuitively express page structure for both authors and readers.

## Estilos (Aria).

Introducción

Lo mejor es usar los elementos HTML nativos porque te dan el foco, el soporte de teclado, etc.; pero, a veces, no hay una plantilla simple o nativa de HTML, como en un menú pop-up. Para estas situaciones, tendremos que utilizar la colección de ARIA

Que puede hacer ARIA?

ARIA puede añadir un label extra y una descripción que solo se expone a assistive technology APIs

<button aria-label="screen reader only label"></button>

express semantic relationships between elements that extend the standard parent/child connection, such as a custom scrollbar that controls a specific region.

<div role="scrollbar" aria-controls="main"></div>  
<div id="main">  
. . .  
</div>

can make parts of the page "live," so they immediately inform assistive technology when they change.

<div aria-live="polite">  
  <span>GOOG: $400</span>  
</div>

One of the core aspects of the ARIA system is its collection of *roles*. A role in accessibility terms amounts to a shorthand indicator for a particular UI pattern. ARIA provides a vocabulary of patterns we can use via the role attribute on any HTML element.

In fact, because keyboard interactions feature so prominently in screen reader usage, it's very important to make sure that, when creating a custom widget, the role attribute is always applied in the same place as the tabindex attribute; this ensures that keyboard events go to the right place and that when focus lands on an element its role is conveyed accurately.

Etiquetas y relaciones de ARIA

Labels

ARIA provides several mechanisms for adding labels and descriptions to elements. In fact, ARIA is the only way to add accessible help or description text. Let's look at the properties ARIA uses to create accessible labels.

aria-label allows us to specify a string to be used as the accessible label. This overrides any other native labeling mechanism, such as a label element — for example, if a button has both text content and an aria-label, only the aria-label value will be used.

aria-labelledby allows us to specify the ID of another element in the DOM as an element's label.

Importantly, aria-labelledby overrides **all** other name sources for an element. So, for example, if an element has both an aria-labelledby and an aria-label, or an aria-labelledby and a native HTML label, the aria-labelledby label always takes precedence.

Relationships

A relationship attribute creates a semantic relationship between elements on the page regardless of their DOM relationship. In the case of aria-labelledby, that relationship is "this element is labelled by that element".

aria-owns is one of the most widely used ARIA relationships. This attribute allows us to tell assistive technology that an element that is separate in the DOM should be treated as a child of the current element, or to rearrange existing child elements into a different order

aria-activedescendant plays a related role. Just as the active element of a page is the one that has focus, setting the active descendant of an element allows us to tell assistive technology that an element should be presented to the user as the focused element when its parent actually has the focus.

aria-describedby provides an accessible description in the same way that aria-labelledby provides a label. Like aria-labelledby, aria-describedby may reference elements that are otherwise not visible, whether hidden from the DOM, or hidden from assistive technology users.

When the size of a set cannot be determined by the elements present in the DOM — such as when lazy rendering is used to avoid having all of a large list in the DOM at once — aria-setsize can specify the actual set size, and aria-posinset can specify the element's position in the set. For example, in a set that might contain 1000 elements, you could say that a particular element has an aria-posinset of 857 even though it appears first in the DOM, and then use dynamic HTML techniques to ensure that the user can explore the full list on demand.

Como ocultar y actualizar contenido

Aria-hidden

Finally, ARIA provides a mechanism for excluding content from assistive technology that is not visually hidden, using the aria-hidden attribute. Applying this attribute to an element effectively removes it and all of its descendants from the accessibility tree. The only exceptions are elements referred to by an aria-labelledby or aria-describedby attribute.

Aria-live

aria-live lets developers mark a part of the page as "live" in the sense that updates should be communicated to users immediately regardless of the page position, rather than if they just happen to explore that part of the page. When an element has an aria-live attribute, the part of the page containing it and its descendants is called a live region.

aria-live="polite" tells assistive technology to alert the user to this change when it has finished whatever it is currently doing. It's great to use if something is important but not urgent, and accounts for the majority of aria-live use.

aria-live="assertive" tells assistive technology to interrupt whatever it's doing and alert the user to this change immediately. This is only for important and urgent updates, such as a status message like "There has been a server error and your changes are not saved; please refresh the page", or updates to an input field as a direct result of a user action, such as buttons on a stepper widget.

aria-live="off" tells assistive technology to temporarily suspend aria-live interruptions.

aria-atomic indicates whether the entire region should be considered as a whole when communicating updates

aria-relevant indicates what types of changes should be presented to the user.