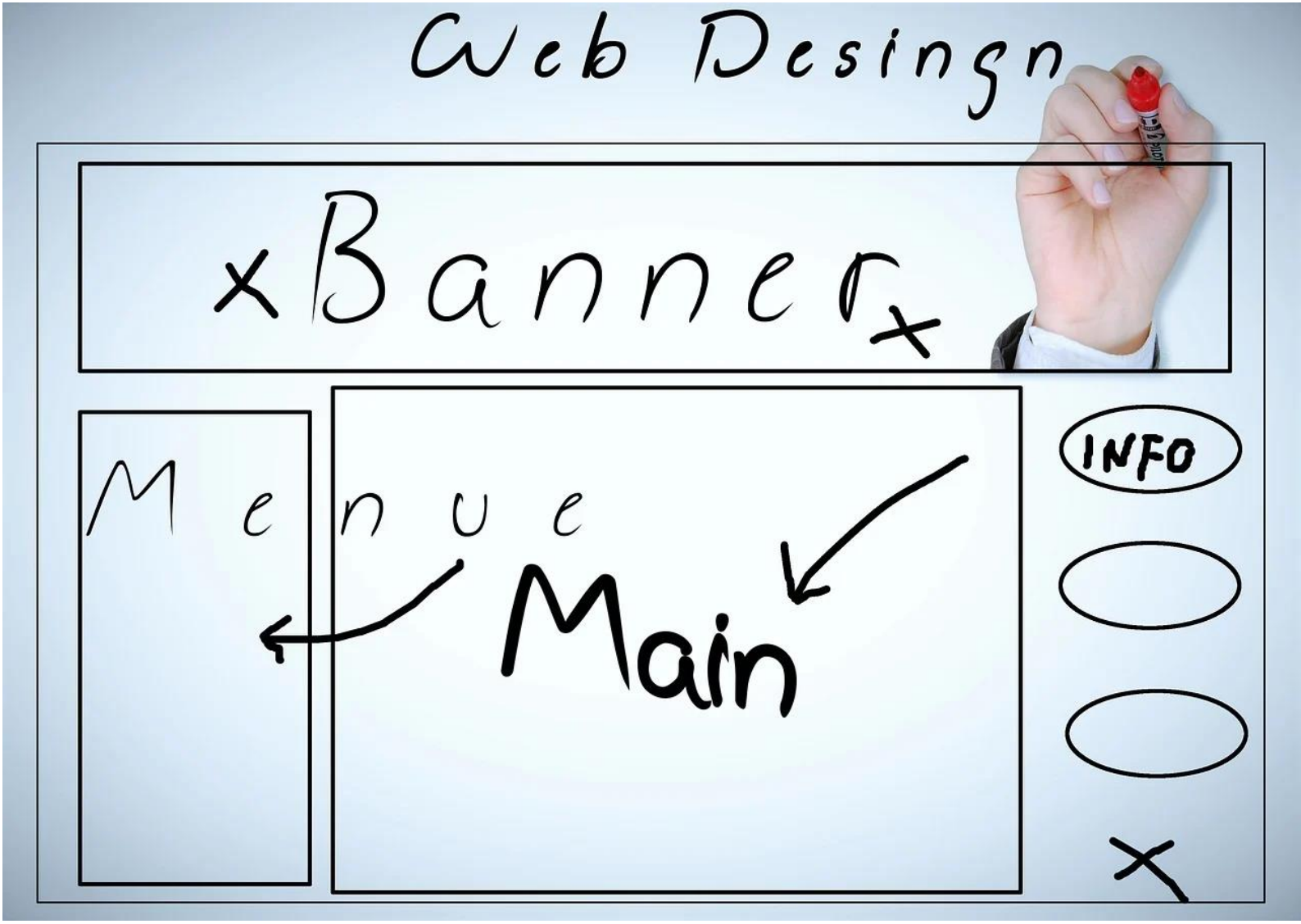

A Practical Guide to Accessible HTML

By Alexander Halser, EC Software GmbH

Text and graphics: Alexander Halser
Cover picture: Pixabay

Published in August 2024.

Accessibility has been a hot topic for quite a while, but now it's getting serious. From July 2025 it's going to be a legal requirement for all new products in the EU, and this also applies to websites and documentation. So how are we going to make HTML accessible?

As of July 2025, all new products in the EU must conform to the European Accessibility Act (EAA), and this includes websites, HTML documentation as well as hardware and software. Web designers and documentation authors are going to have to comply in time if they want to avoid some potentially eye-watering penalties.

Documentation in PDF and some other local file formats can be handled accessibly with updated software, but when it comes to HTML, authors are pretty much on their own. This article explains the basics of making HTML accessible and compliant.

Spoiler alert: *Did you know that accessible websites perform significantly better and are ranked higher in organic search on Google & Co? It's true. We'll get to that later in this article.*

Screen readers are the target

Accessibility covers a very wide range of fields, but as far as web pages are concerned, we only need to consider deaf and visually impaired users, and deaf users on the Web really only need closed captioning for videos. So the real task here is to make HTML accessible to the screen readers that visually impaired users need for accessing web pages.

Accessibility? ARIA!

ARIA (*Accessible Rich Internet Applications*) is the main W3C standard for making HTML code accessible. It achieves this with attributes that define the accessibility functions of HTML tags. One of the key attributes for this is *role*. For example, `<div role="figure">` defines the block enclosed by the `<div>` tag as an image.

However, it's best to only use ARIA attributes when no suitable HTML tag is available that implicitly provides the required role. Instead of defining a `<div>` as a figure, it's better to use a `<figure>` tag directly. The same applies for things like page headers, footers and the main content blocks of the page. The basic principle is always: As much as necessary, as little as possible.

Page structure

Basically, screen readers just search for text they can read using voice output. This includes image descriptions, which still must be entered manually in the `alt` attributes, although we will probably soon have AI-based systems that can generate these automatically. It doesn't matter if the text is enclosed in a deeply-nested sequence of `<div>` elements. Those can be ignored by the screen reader, provided you make the structure of the page clear.

Making page structure accessible to screen readers involves more than just separating the text into headings, paragraphs and body text. The blind user, and thus also the screen reader, must first be able to find the sections of the page that are worth reading.

The basic structure of a typical accessible HTML page looks like this:

```
<html>
 <head>...</head>
 <body>
  <header>
    Logo, navigation
  </header>
  <main>
    Content
  </main>
  <footer>
    Footer / end of the page
  </footer>
 </body>
</html>
```

This simple structure is already enough to give screen readers basic access to the page, eliminating the need to step through nested elements and menus. By the way: Don't confuse the `<header>` tag with the standard `<head>` tag! The first defines a visible header or heading area in accessible pages, while the `<head>` tag is the standard HTML element for the page's settings and meta data.

It may come as something of a surprise that Microsoft rather than the W3C currently has one of the best examples of accessible page structure, shown in Fig. 1. You can find it on Microsoft's website at https://learn.microsoft.com.

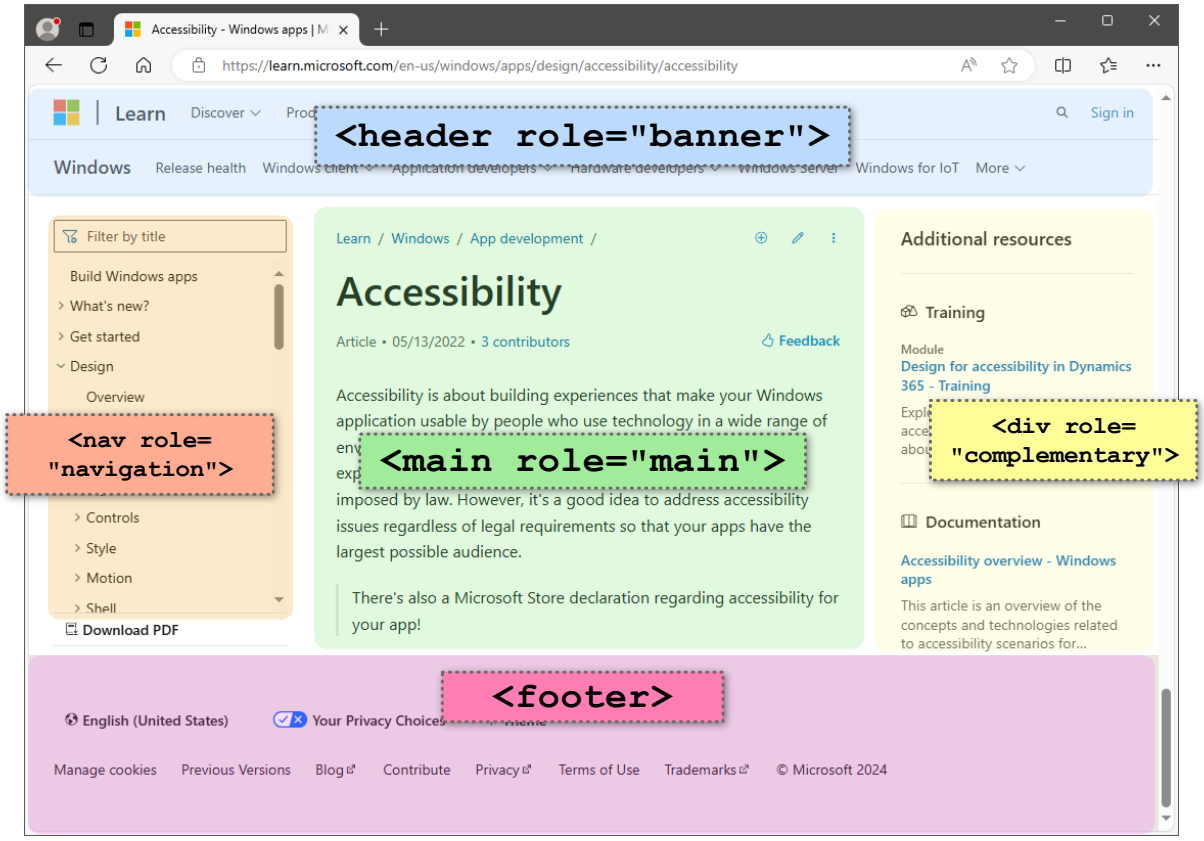

Figure 1: Page markup with <header>, <main> and <footer>

Tip: To maintain backward compatibility with older browsers, only use these accessibility tags to identify the semantic structure of your pages. Theoretically you can tell browsers how to use them with

additional CSS, but this isn't recommended. If an old browser doesn't yet recognize a tag it will simply ignore it *and* any associated CSS styling, but it won't ignore the content.

Example:

```
<main>
  This text appears in Internet Explorer 7,
  despite it doesn't know the &lt;main&gt; tag.
</main>
```

So long as you don't try to use them for CSS styling, semantic accessibility tags will work just fine in old browsers that don't recognize them. Screen readers will still see and use them.

Table of Contents

Most websites include a navigation menu, documentation almost always needs a structured table of contents, often referred to as the TOC in documentation parlance. Standard HTML `<ul>` lists are now well established for both TOCs and menus, but to identify them as navigation elements for screen readers, and to distinguish them from regular lists, you need to wrap them in an additional `<nav>` tag. This is also a good solution for link lists within the body text.

Example:

```
<nav>
 <ul>
  <li>Chapter
   <ul>
    <li>Entry</li>
   </ul>
  </li>
 </ul>
</nav>
```

The *role* attribute

Your next job is to define roles for elements on the page, where necessary. The `<header>`, `<main>`, `<footer>` and `<nav>` elements are semantic by definition. Their roles are clear to screen readers, so they don't require any further ARIA classification. In practice, however, seemingly redundant role attributes are often added. More about this further below.

These basic semantic elements are known as ARIA *Landmark Regions* (Fig. 2). Use them sparingly on your pages – multiple occurrences create more confusion than useful structure.

HTML Tag	Implicit Role	Use for
`<header>`	`role="banner"`	Page header
`<main>`	`role="main"`	Main content
`<footer>`	`role="contentinfo"`	Footer / end of the page
`<nav>`	`role="navigation"`	Menus, table of contents
`<aside>`	`role="complementary"`	Additional sidebars / columns
`<section>`	`(none)`	Content structure

Fig. 2: ARIA landmark tags and their implied roles

Though the basic landmark tags do imply a particular role, it is beneficial to label them with a role attribute. The redundant classification improves backwards compatibility with older browsers.. You can see this in the example in Fig. 1, where Microsoft explicitly writes `<main role="main">` in the source code, although the role is already implicitly defined by the tag itself. An obsolete browser like *Internet Explorer* doesn't recognize the `<main>` tag. However, this is not a problem because it still displays its content - including formatted content - as if it were a `<div>`. In this case, the additional redundant classification makes the unknown generic tag meaningful for screen readers.

Other HTML elements, like `<img>` and `<div>` tags are generic and require additional information to enable screen readers to interpret them correctly. You achieve this with the ARIA `role` attribute, which assigns defined roles to HTML tags.

Example:

```
<img role="figure" src="largeimage.png">
```

In addition to identifying elements, the *role* attribute can also be used to redefine the roles of some elements, or make them invisible to a screen reader. We will return to this further below in the discussion of tables.

ARIA defines a wide range of role classes and functions for this purpose. This ranges from basic roles for document structure to specialized roles for specific input and display elements, such as those used in interactive web applications. These can be supplemented by additional attributes for values, such as the current value of a progress bar or the open or closed status of expanding elements. However, most of these roles and additional attributes are generally irrelevant for static HTML documentation, with one important exception.

The "none" role

Assigning *"none"* as the role attribute is *not* the same as omitting the attribute, quite the contrary! Screen readers interpret the following two HTML tags very differently:

```
<img src="icon.png">
<img src="icon.png" role="none">
```

In the first example, the software will report the image to the user by reading its alt text, if present, or just mentioning its presence. In the second case the reader completely ignores the image. You can also use the value "*presentation*" as a synonym for "*none*", and this makes the purpose clearer: It defines the image as a decorative element with no meaningful functionality, and the screen reader just ignores it.

The *role* attribute can be applied to groups of elements as well as individual elements. This is particularly useful for tables used for layout rather than presentation of tabular data. More on this below.

Structuring text

If you're familiar with HTML, you will naturally structure your texts with headings and paragraphs using the standard `<h1>…<h6>` and `<p>` tags. These are already comprehensible for screen readers and normally require no further semantic description to satisfy accessibility requirements. The same applies to unordered and ordered lists defined with `<ul>` and `<ol>`.

But what about things like info boxes and warnings? Warnings in particular need to be highlighted, and the ARIA role attribute has several ways of doing this.

Example:

```
<div role="alert">Important note!</div>
```

The "alert" value identifies urgent warnings that are often also time critical. However, since notes in static text are not time critical, this value should be used sparingly. There are a number of less urgent options you can use, including "note" or "comment". In some cases, you can also use "complementary", which corresponds to the semantic <aside> tag and identifies boxes that are typically placed next to the body text. Text boxes containing helpful tips can be given the "suggestion" role. You can use the "code" role for boxes with source code examples, although it's generally better to use the standard <code> tag for this – semantically, it's the same as a tag. If all this sounds like a lot of work, don't despair. These are just examples of what is possible and available. Detailed markup like this goes well beyond the normal legal requirements for accessibility.

Important notes: The role "alertdialog" is reserved for modal dialogs in web apps and shouldn't be used for warning boxes in documentation. Similarly, it's better not to use "contentinfo", because it corresponds to the <footer> tag.

Images and figures

As the discussion above indicates, you have a wide range of options and some flexibility when applying semantic markup to body text and text boxes. The requirements are much stricter for images, since unsighted users cannot see them and must rely on descriptions to know what is on the page.

As you probably already know, images must have an alt attribute to be accessible. Often referred to simply as "alt text", this attribute contains a text alternative to the image and should generally be included for all images. Images can also have an optional title attribute that defines a caption without displaying it below the image.

In addition to this, the ARIA standard also includes some interesting additional options for images with visible captions. Such images are typically "figures", in the sense of an entry in a list of illustrations – if this is available.

For new content, the W3C recommends using the <figure> tag along with <figcaption> for the caption. However, you might prefer to use role="figure" as markup for existing content, because this doesn't require you to replace existing image tags. The result is the same.

Example:

```
<div role="figure" aria-labelledby="fig1">
 <img src="imagname.png">
 <p id="fig1">Image caption text</p>
</div>
```

In this example, the markup in the tag achieves two objectives: It defines the tag as a figure, and also links to the caption with the aria-labelledby attribute. The caption element must have a unique ID to which the figure element can refer. Alternatively, you can use the more modern version using the semantic tags directly, like this:

```
<figure>
  <img src="imagename.png">
  <figcaption>Image caption</figcaption>
</figure>
```

In this example, the caption is automatically associated with the image by the nested tags.

What about purely decorative images, such as emoticons? Depending on the context you may need to describe them, but in many cases it makes sense to hide them semantically:

```
<img src="smiley.png" role="presentation">
```

Tables

Strictly speaking, tables should only be used for tabular data in HTML and using them for layout is generally frowned on. In the real world, however, you will often have to deal with old HTML using layout tables, and you won't want to spend hours converting them to something more modern. The question is, how to make pages with layout tables accessible.

Tables that really contain tabular data are fairly straightforward. The `<table>` tag is self-explanatory for screen readers and doesn't require an ARIA role. The table's content and structure may require some additional markup for accessibility, however. Even tables presenting tabular data will usually have one or more headers, and depending on the content they can also have individual headers and footers for the columns. For sighted people, these are easy to identify with colors and text formatting, but screen readers need additional information.

To start with, we need to identify the structural components of the tables with the standard `<thead>`, `<tbody>` and `<tfoot>` tags, as shown in Fig. 3. This adequately covers the vertical structure of the table. These tags also have an implicit role and thus do not require any further semantic markup, and they are also easy to add to existing tables.

```
<table>
  <thead>
    <tr>
      <th></th>
      <th>Period 1</th>
      <th>Period 2</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        Chris
      </td>
      <td> € 1000.00 </td>
      <td> € 2000.00 </td>
    </tr>
    <tr>
      <td>
        Hans
      </td>
      <td> € 3000.00 </td>
      <td> € 4000.00 </td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td>
        Total
      </td>
      <td> € 4000.00 </td>
      <td> € 6000.00 </td>
    </tr>
  </tfoot>
</table>
```

Fig. 3: Table structure defined with <thead>, <tbody> and <tfoot>

Things get a little more complicated in tables with headers for the individual columns. However, there is already a standard HTML solution for this: The `<th>` tag with the `scope` attribute, which makes it possible to specify what the header in a specific cell refers to:

```
<th scope="col|row|colgroup|rowgroup">
```

Compare Fig. 3 above with Fig. 4 below. In the first column, the `<td>` cells have been replaced with `<th>` cells, identifying them as headers. So we now have header cells in both the first row of the table and in the first column. The `scope` attribute now clarifies their function: some are column headers, while others are row headers.

The only decision left is how to classify the very first cell in the upper left corner. However, since it is empty, screen readers will ignore it anyway.

```
<table>
  <thead>
    <tr>
      <th></th>
      <th scope="col">Period 1</th>
      <th scope="col">Period 2</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">
        Chris
      </th>
      <td> € 1000.00 </td>
      <td> € 2000.00 </td>
    </tr>
    <tr>
      <th scope="row">
        Hans
      </th>
      <td> € 3000.00 </td>
      <td> € 4000.00 </td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <th scope="row">
        Total
      </th>
      <td> € 4000.00 </td>
      <td> € 6000.00 </td>
    </tr>
  </tfoot>
</table>
```

Fig. 4: Table header cells identified with the scope attribute

Table titles

The structures shown above already make it much easier for screen readers to navigate your tables and present their data. However, if we leave it like this the user will only know whether the content is relevant *after* reading it, which would waste a lot of time. Adding table descriptions significantly improves the situation. If the table already has a `<caption>` tag, effectively providing a visible table header, this can be used directly and without additional markup:

```
<table>
 <caption>
  Revenue by period
 </caption>
  ...
</table>
```

If the heading is in a separate paragraph, you can reference it with `aria-labelledby` and a target ID, in the same way as with captions for images:

```
<div role="figure" aria-labelledby="tb1">
 <table>…</table>
 <p id="tb1">
  Revenue by period
 </p>
</div>
```

If none of these options are available, and in tables without headers, you can reference a description with the `aria-describedby` attribute. This works in the same way as `aria-labelledby`. The referenced tag and its text content can also be invisible and can be hidden, for example by using `style="display:none"`.

Layout tables

So much for real tables with tabular data. What about layout tables? In addition to bulky, they are almost never responsive and they are definitely a problem for accessibility as well. However, there they are, and getting rid of them in existing pages and layouts often requires an inordinate amount of work that you just don't have time for, particularly if the tables are also nested.

Screen readers can get hopelessly lost in pages using layout tables, making them effectively inaccessible for unsighted users. All is not lost, however. As we already indicated, existing elements can be redefined with the ARIA *role* attribute. In the case of tables you just need to change their implicit `role="table"` by adding an explicit `role="presentation"` attribute, which tells screen readers that it is just a layout element and nothing more.

Example:

```
<table role="presentation" width="100%">
  <tr><td colspan="3"> Page Header </td></tr>
  <tr>
    <td width="20%"> Left Column (Navigation) </td>
    <td> Middle Column, Main Content </td>
    <td width="20%">
      Right Column (various blocks)
      <table role="presentation">Etc., etc... </table>
    </td>
  </tr>
</table>
```

The table's own `<tr>` and `<td>` tags all inherit the table's own presentation role. Nested tables are separate elements and must have their own *role* attribute. Screen readers then ignore the table tags and only focus on the contents of the cells. If time is of the essence, you can just add `role="presentation"` to the main `<table>` tags and have done with it. For the screen readers this is the same as writing:

```
<div> Header </div>
<div> Column </div>
<div> Main Content </div>
<div> Column </div>
```

This is accessible, but it could be clearer. You can massively improve navigation with some judiciously placed `role` attributes in the relevant `<td>` tags.

Extended example with additional `role` attributes:

```
<table role="presentation" width="100%">
  <tr><td role="banner" colspan="3"> Page Header Content </td></tr>
  <tr>
    <td role="navigation" width="20%"> Left Column (Navigation) </td>
    <td role="main"> Middle Column, Main Content </td>
    <td role="complementary" width="20%">
      Right Column (various blocks)
      <table role="presentation">Etc., etc... </table>
    </td>
  </tr>
</table>
```

HTML pages in dark mode

Dark mode, also known as dark theme or night mode, is a display setting on many computers and smartphones. The default setting on most electronic devices is black text on a white background, sometimes called light mode. Switching your device to dark mode will display white text on a dark background (see fig. 5).

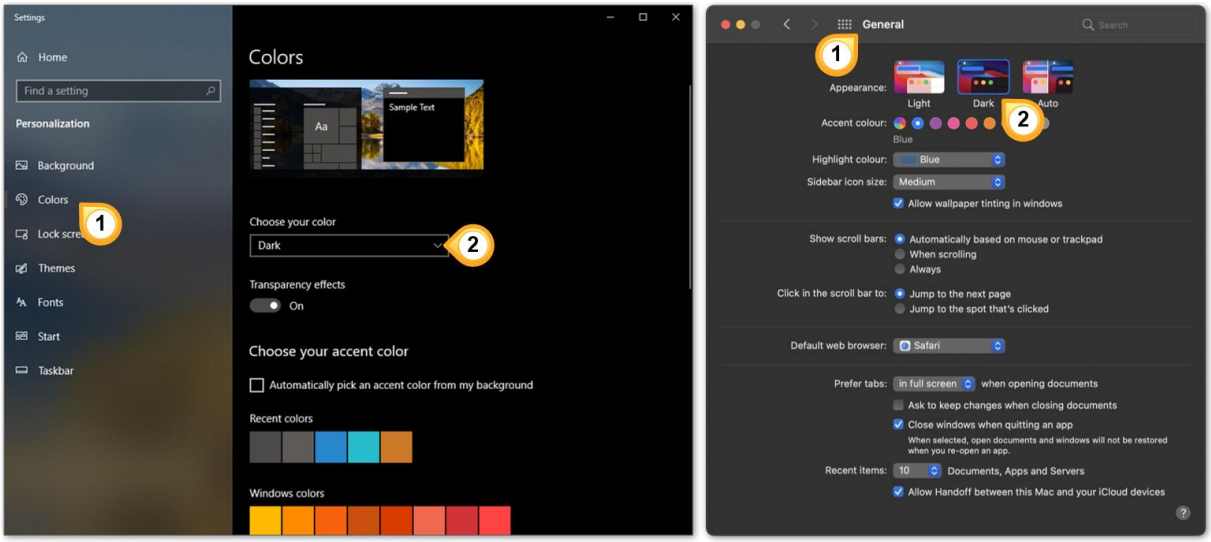

Fig. 5: Turn on dark mode in Windows (left) and macOS (right)

Dark mode is trending. It's supposed to reduce eye strain and is gaining popularity. Most macOS and ever more Windows applications support light and dark themes to adjust to the user's preferences. All recent web browsers do. Most websites, however, still adhere to the common *black-text-on-white* color scheme and do not adjust their layout based on the system settings. The typical result is a browser window with glaring bright content amidst a dark screen.

Dark mode is *not required* for accessibility with screen readers, as the software does not make a difference between light and dark colors anyway. But it creates a good first impression for your website.

When we look at the bright side of dark mode, pun intended, it is very easy to implement by a simple CSS rule that defines alternative colors for the various page elements. All you have to do is to add a **@media** section for dark mode to your CSS.

Example `style.css`:

```
html, body {
  background: #FFFFFF; /* background for light mode (default) */
  color: #000000;      /* black text for light mode */
}

@media (prefers-color-scheme: dark) {
  :root, html, body {
    background: #404040; /* dark gray background */
    color: #FFFFFF;      /* white text */
  }
  a {
    color: #7FC9FF;      /* link color for dark mode */
  }
}
```

The media query `@media (prefers-color-scheme: dark)` does the trick on all modern browsers. Within this CSS block, you define alternative colors for background, text, headings, links, you name it. When the web browser runs in dark mode, according to the system preferences, the media query becomes active and changes the color scheme of the entire page (see fig. 6).

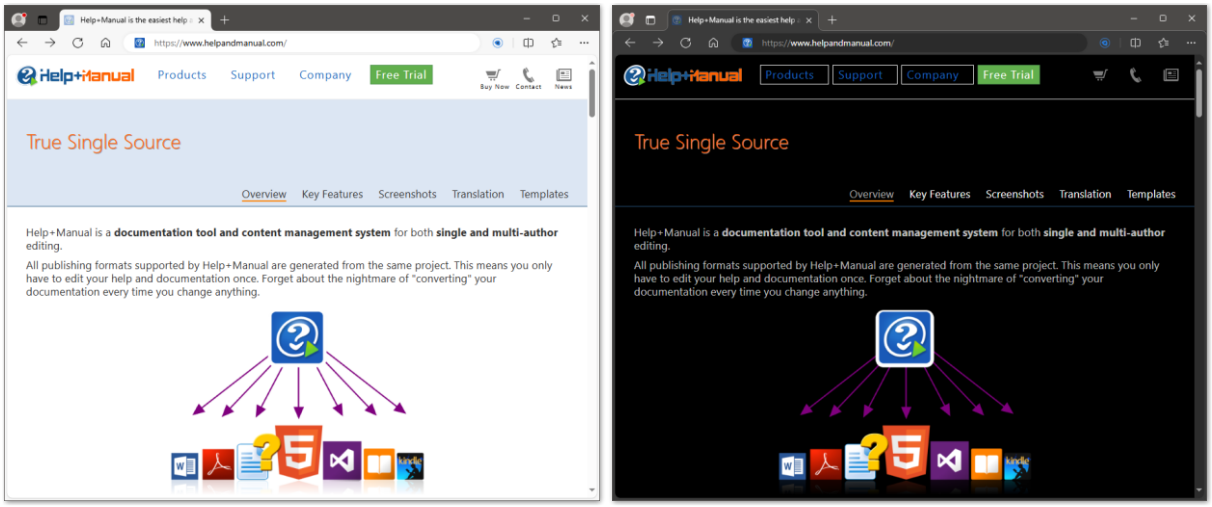

Fig. 6: Web page with light theme (default) and alternative dark theme

In spite of the simple implementation, you need to pay a lot of attention to details to get the website look good in dark mode. In practice, you need to meticulously review almost every single page. When would be a better opportunity to do this than now, when we are implementing accessibility markup?

Good to know: *Dark mode is not a requirement to meet accessibility standards. But it is quite simple to implement.*

Testing your website

Full-featured screen readers like *JAWS* and *NVDA* are certainly also useful for detailed testing of accessible web pages and web applications. However, it's probably better to get started with the evaluation tools recommended by the W3C.

My personal favorite here is the free Chrome extension *silktide*, which provides a good overview of the accessibility of HTML pages, covering everything from ARIA checks to simulation of various levels of visual impairment.

Measuring SEO effects

We have all suspected for some time, that search engines rank accessible websites higher than those which are not. Google says, they are "considering" this among many other factors, without going into details. So, the big question is: does accessibility really count?

The first thing you need to do (if you haven't already) is to register your website in Google Search Console. This is about *organic search* on Google. It has nothing to do with *AdWords* or with the cookie-based *Google Analytics* tracking, though Google may nag you about combining the services. *Search Console* is a free service and generally recommended for every website owner. After registering your web page, you get a monthly brief by email, that tells you how you are doing regarding organic search. We need this data and we'd need several months of this data, to conduct an A/B test before and after the modifications.

On our own website, we had accessibility markup in place already. Sort of. We were using `<header>` and `<footer>` tags, had a `<nav>` section and structured navigation menus as lists. But it wasn't implemented consequently and the approach was more like *"seems to be good practice to structure content somehow, let's try and see"*, without a clear focus on accessibility.

In May 2024, when this article was drafted, we eventually redesigned our website. We changed the markup only, not the page content. The Google search results for May were encouraging, yet not significant. By the end of July, however, the organic search impressions had more than doubled in comparison to the months before.

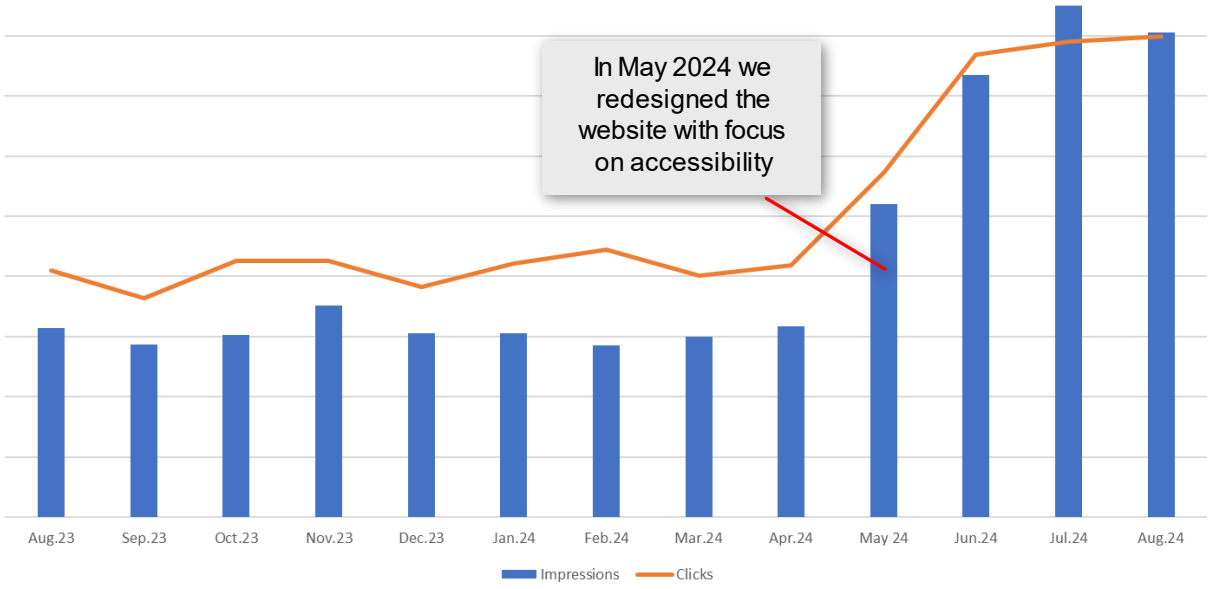

Fig. 7: Organic search impressions for www.helpandmanual.com

Is this the result of the accessibility update? Or was it influenced by some other measures that we implemented around the same time? We don't know. It's possible that the general update of all pages has pushed the search results, though most pages just received updated markup, not new content. None of that really explains such a huge impact. So we can say: yes, accessibility does count for Google & Co! We'll have to suss out to which degree.

Summary

Making websites accessible really isn't all that hard. At first glance, the ARIA specification includes an intimidatingly comprehensive toolbox for making even the most complex interactive web applications accessible to visually impaired users. However, we only need a fraction of the full spec to make a static website or technical documentation accessible.

Most authors will probably be able to manage with just half a dozen HTML tags, and ARIA *role* attributes can be safely added to legacy code with very little effort. It's important to remember that the *role* attribute is an *additional* piece of information for assistive technologies. It doesn't affect how your pages and their content are displayed in web browsers.

Last, not least: while we all probably agree that accessibility is a good thing, the incentive of being higher ranked and getting better visibility in organic search on Google & Co must not be underestimated.

Links

European Accessibility Act (EAA) | European Commission

ARIA Authoring Practices Guide (General Info) | W3C

ARIA Landmark Regions |W3C

ARIA Role Types – Overview | MDN (mozilla.org)

prefers-color-scheme | MDN (mozilla.org)

Web Accessibility Evaluation Tools List | W3C

Silktide Chrome Extension | silktide

Screen Reader JAWS® | Freedom Scientific

Screen Reader NVDA | NV Access

Google Search Console | Google

Autor

Alexander Halser is the founder of EC Software GmbH and senior developer of the documentation suite "Help+Manual". He has worked with Delphi and Web languages for more than 30 years.

Contact:
alexander.halser@ec-software.com
https://www.helpandmanual.com